An Architecture for BSD UNIX (NetBSD?) Networking

Dennis Ferguson

(Version 0.11)

Introduction

This presents a summary of possible improvements to the very old networking stack which has long been used in BSD UNIX kernels. The goals of this project are as follows:

- Provide a better framework for SMP support, and for finding operations in the network stack which may be parallelized at a cost which is justified by the benefits and which do not violate basic correctness requirements;

- Develop an arrangement for networking configuration state which makes it easier to represent configuration problems which often occur in modern networking (and not-so-modern, somewhat obsolete networking which is still more modern than the present network stack was originally designed for) like VLANs, PPP-over-something, multilink or aggregated interfaces, NAT, bridging, and so on, without unnecessarily complicating the simple, “normal” cases;

- Ensure that all networking structures are dynamic so that when, say, interface hardware is removed (perhaps a USB dongle is unplugged) all state associated with that interface can be safely deleted from the kernel, without freeing structures to which concurrent references are in progress or leaving dangling pointers behind (e.g. mbufs should not include a pointer to a struct ifnet!);

- Rethink and unify forwarding operations which were traditionally treated as separate warts. For example, if local address recognition, unicast forwarding and multicast forwarding all depend on a packet destination address, perhaps with the source address concatenated to it (i.e. multicasting) then, rather than asking the questions, “Is it a local address?”, “Is it a multicast?” and “Is it a unicast?” first and looking at the same address fields in separate places in an only very slightly different way, it would be better to just take the destination and source address keys from every packet and do a unified lookup whose result both answered the questions above and told us what to do next, all at the same time. Similarly, if packet demultiplexing to sockets is ultimately going to look at the source and destination addresses, the IP protocol number and the first 4 bytes of the transport header then just extracting those fields from every locally-bound packet and doing one common lookup to find the socket (and, simultaneously, the protocol code and session information needed to process it) is a much better arrangement than demultiplexing to protocol code first and then having each protocol implement its own separate look at the remaining fields. And if the kernel support requirements for multicast routing protocols turns out to be exceedingly similar to the kernel support requirements for ARP (which I believe is best viewed as just another routing protocol), and likely ND, then having one common mechanism which is used by all of these is better then treating them all as entirely separate problems.

- Support multiple routing instances per protocol. Multiple routing tables are required for bridging, and making support for them generic for all forwarding protocols can help provide a structure to cleanly deal with some common problems, like providing a clean configuration model for NAT or for coping with several default routes, as well as being necessary to support certain router and routing protocol configurations which now exist.

- Fix some long-standing misfeatures of the current stack. For a glaring example, a UDP server, like bind or ntpd, should be able to determine the local address on the box that an incoming packet was addressed to (knowing this is necessary to construct a proper reply) without needing to open a separate socket for each local address in the system to accomplish this; those daemons fail badly, and for no good reason, on systems with a sufficiently large interface configuration. Another example might be many of the uses of BPF which involve using it to send and receive packets, in that I believe that most of these are hack-arounds to do things that a decently functional system should accommodate directly; the fact that we need BPF to send and receive something as basic as DHCP client packets strikes me as deeply wrong.

I believe, despite the contrary impressions the above might convey, that this is a conservative plan. Viewed from outside the kernel the network stack should look quite familiar at a basic level, and it should be possible to support a backward-compatible system call interface for configuration which works right in the most common cases. Most new functionality is aimed at addressing network configuration problems that are common now but which didn't exist when the current stack was conceived. While the unification of forwarding operations may seem like it might be the cause of a requirement to undertake a rather complete rewrite of all network layer forwarding code I believe that the causality actually runs in the other direction. It is the production at an SMP-capable network stack which is clean, reliable and can be eventually made bug-free which absolutely requires a fairly complete rethink and rewrite of the network-layer code in the current kernel, since no amount of incremental modification is likely to be able to safely untangle the spaghetti that exists now. Given this, the unification of forwarding by a common paradigm built around a small number of central data structures which are used for as much as possible provides a way to structure the rewrite so it can be accomplished with a minimum of work, and reduces the incentive to leave particularly warty bits of code behind.

I guess I should point out up front that a not inconsiderable part of the design of both network stack configuration and the arrangement of forwarding operations has been derived from consideration of how to make multicast forwarding look like a fully integrated, "normal" part of the system. This is not to suggest that I have some particular interest in multicast routing (in fact the opposite is probably closer to the truth: I like the service it provides in theory, but have distaste for the routing protocols required to implement it in practice), or even that I think multicast forwarding is important, but rather was a decision arrived at from a more fundamental consideration of what is a "good" design. It turns out that most of the primitive operations you need to be able to implement to provide support for multicast forwarding are primitives which can also be put to use to implement a rather diverse range of functions a non-multicast router or a non-forwarding host is required to do (you may notice this most often when we look at next hop functions). Furthermore, there seems to be no downside for a non-multicast router or a non-forwarding host to work this way that I can discern. While there might be alternative approaches in these simpler cases (there are many alternatives for a non-forwarding host with only one interface, but constraining oneself to that is probably not so useful) they do not seem better. They are just different. That being the case, and noting that multicast forwarding is part of a full implementation of several of the layer 3 protocols we support (as well as L2 bridging, differing only slightly in the details) it seems like doing things in a way which provides support for the needs of multicast forwarding and, particularly, avoiding things which get in the way, will be the best of all possible designs of the functionality here, and that would be true even if no one gets around to implementing a multicast routing protocol which uses it.

Finally, while some of this may appear like “research”, many of the changes I’m suggesting are changes that I have seen before which I know, from experience, not only can be made to work but can be made to work very, very well. Other bits, like techniques or approaches for efficiently scheduling kernel threads to get networking work done, I have not seen done well, so for the purposes of this work I have leaned towards arrangements which seem likely to be simple and to produce correct results and left the work of finding truly efficient approaches to the same end to the future.

Interfaces and Interface Structures

The BSD networking stack is quite burdened by its origins, and no where does this constrain its operation more than in its treatment of network interfaces. The original view of a network interface treated it much like a disk controller or a serial line multiplexor. When the system was booted its hardware configuration would be polled and each bit of hardware would be matched up to a software driver for that hardware and a (usually compiled in) software state structure which the driver would use to track the hardware state. For network interface hardware this state structure, which generally included a public `struct ifnet`, was expected to hold all the state about the “interface” that was ever going to be required up to, but not including, the protocol addresses. That is, there was a built-in one-to-one correspondence between “interface” hardware, the “interface” structure and, as a result, the network layer view of the “interface”.

The difficulty with this is that term “interface” has become much more ambiguous than when the BSD networking stack was first conceived; what a network “interface” is depends on which layer of the network stack you are thinking about. If you are thinking about hardware, or perhaps the link layer, then an “interface” is an entity which probably has a connector, with media and clocking configuration, and framing (“carrier”), and an output queue (or perhaps more than one) which is used to send and receive packets. If you are thinking about a network layer protocol like IP, however, it has become common to view “interface” as being defined by multicast scope, that is as a group of two or more neighbors which all can receive a single multicast packet originated by any one of them with an “interface” on the common network (this is not to imply that there is no longer a requirement to support Non-Broadcast MultiAccess interfaces, which at one time were almost the only kind of interfaces there were, but rather just to point out that in practice NBMA configurations are seldom seen now and are usually treated as multiple point-to-point “interfaces” to retain the multicast scope definition). It is true that in many common cases, for example a conventionally configured Ethernet, there is one connector and one multicast scope so the hardware and network layer views of “interface” are the same. If VLANs are implemented on the Ethernet, however, while the single connector (and hence the single hardware “interface”) remains, one now may have multiple discontiguous multicast scopes sharing the common hardware which would really like to appear as multiple “interfaces” at the network layer, so one hardware “interface” can support several network layer “interfaces”. Conversely, link layer procedures like MLPPP or Aggregated Ethernet (or bridging, which can be seen as a network or link layer procedure depending on your point of view) can result in several hardware “interfaces” which are best viewed as a single network layer “interface”, so several-to-one arrangements exist as well. Of course if VLANs are implemented on the Aggregated Ethernet one arrives at a several-to-several relationship reflecting the orthogonality of the hardware and network layer views of “interface”.

Some uses of pseudo-interfaces arguably present another divergence. Some types of pseudo-interfaces, like the SLIP and asynchronous serial PPP, are interfaces for use with hardware which is not unambiguously “network interface” hardware. Here the relationship between the “interface” and its hardware, and the number of such interfaces required, is the topic of configuration. Many other pseudo-interfaces, however, have no relationship with hardware at all, but rather describe operations on packets like encapsulations, or NAT, some aspect of which is convenient to view as an “interface” at the network layer even if the operations being performed have nothing to do with anything one would associate with network hardware. In this case the “interfaces” are entities which represent network layer processing configuration which, like routes, we would like to add and delete as needed.

Taken together, I believe these considerations suggest the following arrangement and properties of a new design for interface structures:

- All structures should be dynamic. That is, they should be added when they are required, changed to modify their behavior and deleted when they are no longer useful. This will require some care in defining where pointers to these structures may or may not be stored.
- The representation should accommodate both the hardware view of “interface” and the multicast scope view preferred for network layer processing, by separating these into different objects which represent the relationship between the network layer entity and its underlying hardware without constraining there to be a one-to-one relationship between them.
- There should be an additional per-network-layer-protocol (e.g. IP, IP6) representation of the interface which includes protocol-specific processing information which is naturally configured on a per-interface basis, e.g. MTU, incoming and outgoing firewall filters and the route table to use to begin the lookup for incoming packets.
- The arrangement of these structures should reflect their hierarchical dependencies on other configuration (for example, network layer “interfaces” should be configured “on” the hardware layer “interface” they are associated with). Since several-to-one dependencies are common this will generally involve building tree-like configuration structures which are generally rooted at some device driver.

What follows is an overview of the structures proposed here to represent interface configuration. While it might be short on specifics, it attempts to impart understanding by describing what each structure does and how it might be used to represent configuration in particular networking situations. Note that the structures described here are intended to be the “public” face of the interface configuration, but may be embedded in larger structures which might additionally include link-specific protocol and/or hardware driver state.

The *ifdriver* Structure

The root of all interface configuration is the *ifdriver* structure. The *ifdriver* structures are the only ones compiled into the kernel, as opposed to being dynamically added, and are intended to represent the presence of driver code in the kernel to use for particular functions. When the *ifdriver* structure represents an actual interface hardware driver it may also include the intersection with hardware currently present in the system which that driver can deal with. Pseudo-device *ifdriver* structures, on the other hand, may stand alone, with their sole reason for existence being to provide an entry point for configuration to add structures on top of the *ifdriver* when the function provided by that pseudo-device is required.

Each *ifdriver* is identified by a name, like “ppp” or “fxp”, and perhaps by a brief, generic description of the function the driver serves. The *ifdriver* also provides an entry point for kernel code to call to do the driver specific operations needed to add interface structures on top of the *ifdriver* when it is desired to use the driver for something. Beyond this, I’m not yet clear on what else is specifically required to be included in the *ifdriver* structure, but the overall intent should be apparent: When a kernel is booted there should be a way for configuration applications to determine the existence of driver code, and the presence of hardware which the driver code knows how to use, but everything beyond this should be the topic of configuration and added only when it is determined that the driver will be used.

The *ifdevice* structure

The structures which can be added to an *ifdriver* structure are one or more *ifdevice* structures, or *ifd*’s. An *ifd* holds configuration and state related to networking hardware; if it has a connector, or needs configuration related to media or clocking, or has “framing” or something else which could be called a “carrier”, or if it has an input or output queue, it needs to be represented as an *ifd*. The *ifd* also identifies the “type” of hardware (maybe Ethernet, synchronous serial, ATM, etc.), and has an “encapsulation” which can be configured to select from among the variations this “type” of hardware can support (Ethernet is generally always Ethernet, so the “encapsulation” in that case might be a constant, but serial interfaces generally have many choices for the “encapsulation”). The combination of the “type” and the setting of the “encapsulation” will generally define the kinds of things which are appropriate to configure on an *ifd*.

An *ifd* may be configured on an *ifdriver* structure, or may be configured on another *ifd*. In the former case the *ifd* would be named (by default; if other variations are useful, feel free) using the format “%s%u”, where the “%s” string is the name of the *ifdriver* parent of the *ifd* while the “%u” is an arbitrary unit number. In the latter case, where the parent is another *ifd*, the *ifd* would be named “%s:%u”, where the “%s” string is the name of the parent *ifd* while the “%u” is an arbitrary unit number assigned to the child *ifd*. Besides this name string, however, each *ifd* in the system is also identified by a unique 32-bit index (the *ifd_index*), which is used as a short-form way to refer to the *ifd* in configuration.

It is probably worth explaining a situation where configuring an *ifd* on another *ifd* might be necessary to represent a hardware interface configuration. The particular case I’m aware of is in dealing with multiplexed carrier interfaces. For example, a SONET OC-12 hardware interface would require an *ifd* to represent it since it has a connector, it needs clocking and framing configuration, and it can use a “carrier” flag to indicate whether it is in frame or not. If the OC-12 is a concatenated OC-12c this single *ifd*, which will also have an output queue, might be all that is required, since the interface is receiving a single stream of packets and need only have the encapsulation set correctly (e.g. PPP, Cisco HDLC, Frame Relay, Ethernet, etc.) to define what it should send and receive. It is possible, however, that the OC-12 is instead carrying 12 multiplexed DS3 circuits (and this might in fact be a configuration option for the same hardware). In this configuration, while the OC-12 *ifd* would no longer have an output queue, the OC-12 would still have a connector and still require clocking and framing configuration which requires an *ifd* to represent, but in addition each of the DS3's, while lacking a "connector" of its own, would need to have its own independent clocking and framing configuration, and its own output queue and encapsulation configuration (if they are clear channel DS3's), so each DS3 would need representation by an *ifd* (which would look more or less the same as an *ifd* representing an actual DS3 hardware interface) as well, a relationship we would represent by configuring 12 DS3 *ifd*'s on the parent OC-12 *ifd*. Of course it is also possible for each of the DS3's to instead be carrying 28 multiplexed DS1 circuits which would again need their own clocking and framing configuration and hence their own *ifd's*, so it is possible for a hardware interface with a single OC-12 connector to be used in a way which requires 336 DS1 *ifd*'s configured on 12 DS3 *ifd*'s configured on a single OC-12 *ifd* to represent the necessary configuration. And even that might not be enough; SONET framing overhead also includes a low bandwidth signaling channel, and representing that in a way which makes it available for software might require its own *ifd* configured on the OC-12 *ifd* independent of the rest of the configuration. The ability to configure one or more *ifd*'s on another *ifd* is hence necessary to fully represent (some might call it "the horror of") carrier channelized interface configuration. That said, while the infrastructure will be designed to accommodate *ifd*-on-*ifd* configuration, I do have some doubt about whether this will ever be fully implemented since, while hardware to do channelization like this does exist, I know of none available for platforms where a Unix kernel might run and the probability of a sample becoming available in future is decreasing with time as carriers replace traditional transmission hierarchy circuits with Ethernet for Internet access (the modest likelihood of implementation also makes me less unhappy with the ugly colon-separated naming of channel *ifd*'s).

In any case, an *ifd* structure is added on an *ifdriver* by calling a procedure handle in the *ifdriver*, or added on another *ifd* by calling a procedure handle in the parent *ifd*. This will be a common theme for interface configuration; each interface object is the arbiter of good taste concerning what child structures can be configured on it, and since what is a reasonable configuration will tend to depend on the *ifd* "encapsulation" I would expect those procedure handles to change when the interface encapsulation changes. As such, changing the interface encapsulation will in general require that all previous child configuration be deleted so that configuration appropriate to the new encapsulation may be added back after the change. The *ifd* structure also includes a data link address that may be added, changed or deleted (it may have a second, non-volatile data link which can't be added, changed or deleted, to record an address associated with the physical hardware) when appropriate for the encapsulation.

It may be useful to consider the things that won't be part of an *ifd* configuration. The *ifd* won't have an `IFF_POINTOPOINT` flag as this is an attribute which is only useful when viewed from the point of view of a network layer protocol (in a hardware sense most modern Ethernet interfaces are point to point interfaces since they only connect to one other entity: an Ethernet switch port. If the "network" layer protocol is L2 bridging this fact might be relevant, but if the network protocol is IP it is not). Similarly, `IFF_BROADCAST` (a flag which is past its "best before" date) and `IFF_MULTICAST` are only useful to reflect network layer concerns, in particular to answer the question "can I send a network layer multicast(/broadcast) out this interface?", and the answer to that depends not only on the type of hardware underlying the network layer interface(s) but also on how the network layer interfaces are arranged on that hardware. These flags will hence migrate up the configuration tree into those structures which represent the network layer view of "interface".

Note that for *ifdriver* structures representing hardware-less pseudo-interface drivers the *ifd* is mostly unused since there is no hardware to configure. Never-the-less we retain the *ifd* in pseudo-interface configuration for naming purposes and to make the view of such interfaces uniform. More than this, while it is possible to organize configuration on a pseudo-interface *ifdriver* so that only one *ifd* is required (as we will see shortly) we allow driver code to choose multiple *ifd* arrangements if it makes naming or other configuration more natural or convenient. In any case there is no hard limit to the number of *ifd*'s that can be added to a hardware-less pseudo-interface *ifdriver*, while hardware drivers are limited only by the actual hardware resources available.

The *iflogical* structure

Apart from other *ifd*'s, the structure which can be added on top of an *ifdevice* structure is one or more *iflogical* structures, or *ifl*'s. The *ifl* represents a group of neighbors having layer 2 reachability via the hardware it is configured on. While a single *ifl* may represent the entire set of neighbors that can be reached via the *ifd*, there are a number of considerations dictated by interaction of the semantics of "interface" assumed by network-layer protocols with the characteristics of the underlying communications subnetwork which can make a single-network-interface view of the subnetwork inconvenient, if not unworkable. For example, both IP and IPv6 make implicit assumptions that all hosts on a subnet are fully connected and can reach each other directly but multiaccess networks like Frame Relay and ATM do not guarantee this full connectivity and only provide it if they have been (sometimes painfully) configured to do so. This means that *ifl*'s defining interfaces for IP really need to include only groups of neighbors on those subnets which are known to be fully connected. More than this, while it is not strictly required, it is often the case now that the most convenient and functional IP "interface" definition is one where all network-layer interfaces are multicast-capable. For a fundamentally NBMA communications subnet like Frame Relay or (usually) ATM this generally means configuring one point-to-point network layer "interface" for each neighbor on the subnet (since point-to-point interfaces are always multicast-capable), which translates into an *ifl* per neighbor. Similar problems of network-layer interface definition can exist on Ethernet networks in the presence of VLANs for about the same reasons. The *ifl* may hence be viewed as our "impedance-matching" mechanism between the characteristics of a communications subnet we have an "interface" on and the requirements of the network layer protocols concerning the characteristics they expect their view of "interface" to have.

An *ifl* may be added to (or "on") an *ifd*, as mentioned already, or another *ifl*. In either case the name of an *ifl* is a string defined by the format "%s.%u", where "%s" is the string naming the parent structure of the *ifl* while "%u" is an arbitrary unit number assigned to the *ifl*. For example, if an Ethernet *ifd* is named "fxp0", an *ifl* configured on it might be named "fxp0.0". To provide a modicum of backward compatibility we also adopt the convention that if a name string which lacks a "." is used in a context where it must be referring to an *ifl* we assume that it names the ".0" unit. This makes it desirable to use *ifl* unit 0 exclusively in configurations where only a single *ifl* is needed, as is likely to be the case with the vast majority of Ethernet interface configurations, since this allows all the structures representing the interface to be referred to (and perhaps displayed as) “fxp0”. While the string provides a (barely) human-readable name for an *ifl* most software will refer to an *ifl* by a second name, a 32-index *ifl_index* which uniquely identifies each *ifl* configured in the system. The *ifl_index* is used to name *ifl*'s in configuration, is carried in packets to identify the incoming “interface” both for kernel processing and user space applications, and is often used by the network layer protocol to build routing tables and make forwarding decisions. Note that the *ifl_index* number space is logically separate from the *ifd_index* number space, though it may turn out to be convenient to allow either to sometimes occupy the same field and distinguish them by the setting of the high order bit (or something). While the *ifd_index* is used only in the context of interface configuration, however, the *ifl_index* represents the identity of an “interface” for pretty much all packet processing, so an interface needs at least one *ifl* configured before it becomes useful for moving packets.

Since an incoming packet must be tagged with an *ifl_index,* clearly *ifl*'s must be distinguished in a way which can be used to classify arriving packets to identify the *ifl*. That is, while we can expect to know which hardware channel delivered the packet, and hence the incoming *ifd*, we must have a way to determine the *ifl* it logically arrived on. For an *ifd* with only one *ifl* configured this may be trivial; the incoming *ifl_index* can only be the *ifl_index* identifying that *ifl*. For multiple *ifl* configurations, however, each *ifl* must be distinguished by some field in the L2 header (i.e. the packet header that will be consumed before the packet is processed by the appropriate “network” protocol; if your “network” protocol is L2 bridging this may be different than if your “network” protocol is IP), or perhaps in metadata associated with the packet. For this reason, when multiple *ifl*'s are configured on an *ifd* each *ifl* must include an indication of the unique value in the L2 header which identifies the packet as belonging to that *ifl*. This indication, which we'll call the *L2_ID* for lack of something more creative, needs to be in a form consistent with the encapsulation of the underlying *ifd* and *ifl*'s it is a child of; that is the L2_ID needs to make sense in its context.

For a concrete example, consider Ethernet in the possible presence of VLANs and PPP over Ethernet sessions. In the simplest, most common case a single Ethernet *ifl* with no associated L2_ID, indicating that the *ifl* matched packets without a VLAN tag, would be configured on the Ethernet *ifd*. In this case the input processing would receive packets to the network protocols configured on the *ifl* (we'll get to that) having no VLAN tag, identifying their incoming interface with the *ifl_index* of that *ifl*, and would discard other packets as “not for us”. If we were expecting VLAN-tagged packets we would instead add an *ifl* for each VLAN we were interested in with an L2_ID which might include a 4 byte datum of the form 0x81000xxx, where “8100” is the VLAN tag Ethertype and “xxx” is the 12 bit VLAN ID (I just made this up; only the 12 bit value is strictly necessary), with the configuration code ensuring that the L2_ID for each *ifl* is unique. Incoming packets would then be tagged with the *ifl_index* of the *ifl* with a matching VLAN tag as their L2_ID, with other packets being discarded as “not for us”. The L2_ID hence tells us what we need to find in the L2 header to identify the packet as arriving on this *ifl*.

More complex arrangements are also possible. IEEE 802.1ad allows more than one VLAN tag to be added to the packet, with all tags but the innermost being identified by an Ethertype of 0x88a8 (there was also a non-standard form of this which used 0x9100 instead). To configure an interface for doubly-tagged packets we would add *ifl*'s to the Ethernet *ifd* with L2_ID values of the for 0x88a80xxx (or 0x91000xxx for the non-standard version), each of which would have *ifl*'s with L2_ID values of the form 0x81000xxx added on top. Incoming packet processing would find the *ifl_index* (only *ifl_indices* of the

0x81000xxx *ifl*'s would appear as incoming interfaces in packets) corresponding to the pair of tags in incoming packets, discarding the rest as "not for us". Clearly a third or fourth layer of tagging can be represented as well by simply adding a third or fourth layer of *ifl*'s with 0x88a80xxx L2_ID values to the interface.

PPPoE is a much grottier protocol to support, but its configuration also falls out rather naturally with this arrangement. PPPoE Discovery (which exchanges packets with Ethertype 0x8863) runs "on" an Ethernet *ifl* configured as described above. For each session it discovers it adds a PPPoE *ifl* on the Ethernet *ifl* it is running on. The L2_ID value includes the L2 header material needed to demultiplex the session, with the datum perhaps being a 12 byte value of the form 0x88641100xxxxaaaaaaaaaaaa (maybe arranged in a less ugly form), where "aaaaaaaaaaaa" is the Ethernet Mac address of the modem and "xxxx" is the session ID. Packets carrying the VLAN tags represented by the *ifl* configuration below the PPPoE *ifl*, having the PPPoE Ethertype (0x8864) and matching the session ID and source MAC address would be assigned an incoming interface index equal to the *ifl_index* of the PPPoE *ifl*. Note that the PPPoE *ifl* looks the same whether VLANs are involved or not; the latter is determined by the Ethernet *ifl* configuration below the PPPoE *ifl*.

Clearly the *ifl* is not just a mechanism for handling arcane Ethernet configurations; it works equally well for any L2 encapsulation I'm aware of which requires demultiplexing based on L2 header fields, or packet metadata. Frame Relay encapsulated serial interfaces, a service which some carriers still support, would have *ifl*'s configured with L2_ID values indicating each DLCI to expect to be active on the interface. ATM interfaces, which are still widely used on DSL circuits, would similarly have *ifl* L2_IDs representing the VPI/VCI values of active VCs, though the VPI/VCI is not strictly a field in the packet but rather is packet metadata (then again, many ATM NIC hardware vendors conveniently communicate this metadata by writing it to and reading it from the packet). *ifl* configuration can also be made to effect hardware configuration, as would be necessary for an ATM interface and might be useful for Ethernet cards which have some kind of hardware VLAN support, but needn't do so. Also notice that configurations which encapsulate some kind of L2 protocol in some other kind of L2 protocol kind of fall out. For example, Frame Relay, ATM and other L2 formats like PPP can carry not only layer 3 protocol packets but also Ethernet frames. This arrangement is easily (if unattractively) handled by configuring the full required Ethernet *ifl* configuration on top of the *ifl* receiving those frames.

More generally, I hope this has made it clearer what *ifl*'s actually represent. They define the "network" protocol's view of "interface", but have few of the attributes one would associate with a hardware "interface". For sending packets an *ifl* has no output queue, the output queue for an *ifl* would be the one associated with the *ifd* underlying the *ifl* configuration. Instead what *ifl* configuration defines is the L2 header which needs to be added to a packet being sent out the *ifl* "interface" before the frame is queued on the *ifd's* output queue for packet transmission. For point-to-point *ifl*'s, or when sending broadcasts or multicasts out a multiaccess *ifl*, all the L2 information required to construct the appropriate L2 header can be found by traversing the *ifl* configuration tree from the output *ifl* towards the root until the sending *ifd* structure is encountered (unicasts on a multiaccess interface require one more bit of L2 information, the destination L2 address, which, in the arrangement being described here, comes from a routing operation instead). For receiving packets the tree of *ifl* structures configured on the receiving *ifd* describe a map, keyed by L2 data extracted from the incoming frame, to find the *ifl_index* to be associated with the packet and to indicate how to further process the packet (in implementation, however, we wouldn't expect the packet forwarding path to actually look at *ifl* structures to determine this; incoming packet processing would instead look in a separate data structure, built with reference to *ifl* configuration but designed for speed, while I would expect outgoing packet processing to maintain a copy of the full, prebuilt outgoing L2 header associated with a next hop which could just be copied onto the packet). The *ifl*, in addition to providing a view of an "interface" having the properties a network protocol might require, provides a generic representation of the L2-specific data required to build and parse the L2 headers on packets.

Of course the *ifl* will hold state beyond just L2 information; it is additionally a network protocol's view of "interface" as well as a connection point for certain link layer protocols to "run on", and hence will need to represent information that these protocols require. For example the `IFF_LOOPBACK`, `IFF_POINTOPOINT`, `IFF_BROADCAST` and `IFF_MULTICAST` flags are all flags in the *ifl* since they indicate properties which either are only of interest to the "network" protocol or depend at least as strongly on the *ifl* arrangement as they do on they capabilities of the underlying hardware and communications subnet (since *ifl* configuration is almost always arranged so that `IFF_BROADCAST` and `IFF_MULTICAST` can be set I'm also of they view that they might better be eliminated in favor of their opposite, `IFF_NBMA`). I suspect there also needs to be an `IFF_LINK_DOWN` flag in the *ifl* for situations when an L2 protocol, like PPP or perhaps ATM OAM, is used to manage the *ifl* and may need to indicate when the *ifl* is unusable. And, in fact, a lot of uses of the `IFF_LINKx` appear to be things that really belong in the *ifl* since they effect the L2 encapsulation rather than the underlying hardware. A place to store a data link address, like the one configured on an *ifd*, may be necessary to support Ethernet-frames-over-X encapsulations. Entry points to add configuration on top of an *ifl* are also required in the *ifl* since, as always, code associated with the particular type of *ifl* is always the arbiter of good taste with respect to what can be added on top. Associated with this, an *ifl* "type" may also be required to make the requirements and constraints of configuration with respect to this *ifl* transparent. An *ifl* won't be a huge structure (or I hope not, anyway, since there may be many of them) but will carry enough state to satisfy the protocols running on them or configured above.

A few comments on the use of the *ifl_index* might also be in order. In packets the *ifl_index* is intended to replace the incoming interface pointer carried there now. The reason for replacing the pointer with what is essentially a name is that we expect interface structures to be dynamically configurable and it is a terrible idea to store pointers to things that can go away in independent objects (a dependent object would be one which is guaranteed to go away before the structure it points to disappears), so we use index names as an indirection instead. We do provide a way to quickly find a pointer to a structure given an index, so the *ifl_index* is in some sense nearly equivalent to a pointer, but we expect that operation to occasionally return NULL and write code which does the right thing when it does. Beyond this, however, is the fact that we also use the *ifl_index* as part of the key material for route lookups, so the index is used more frequently on its own than it is to find a pointer and making this index available in packets is actually more useful than the pointer would be. We also constrain the *ifl_index* allocation to only use non-zero values to identify *ifl*'s. A zero-valued *ifl_index* is used instead as a well-known value identifying packets originated by "this host" (i.e. which were generated locally rather than received from elsewhere via some "interface" *ifl*), so while a zero-valued *ifl_index* may appear in packets or in routes it does not (necessarily) have a corresponding *ifl* structure. I would also point out that this work would probably be easier if the *ifl_index* were only 16 bits long, since then it would fit directly in locations where the current, 16 bit `if_index` in a `struct ifnet` is carried at the user interface, in particular in (at least) `struct sockaddr_dl` structures and in **sendmsg(2)/recvmsg(2)** control messages. The reasoning behind widening this index is that, while the `if_index` names a thing which was originally thought to have a one-to-one correspondence with some bit of hardware, which would make 65,535 seem like quite a large number (though I've seen hardware configurations larger than this), the *ifl_index*, while used very similarly, names something which is unambiguously a software construct which can be created in any quantity which the software thinks it needs to get its job done. This context makes 65,535 seem like a much smaller number than it used to be, so it seems prudent to bite the bullet and widen it to 31 or 32 bits instead.

Finally, in retrospect, I fear I've made the *ifl* seem rather complex and arcane. In some sense this is unavoidable, since the purpose of the *ifl* is to provide a generic representation of L2 header state and processing which, in full generality, can be complex and arcane, and which inherently needs this complexity; the *ifl* is a tool for dealing with that. On the other hand, the interface configurations one is likely to encounter 99% of the time are just dead simple, and are as simple when implemented with the *ifdriver->ifd->ifl* arrangement as they are now. The *ifl* provides a tool to deal with the remaining 1%, but many people will never have to use it (until they do).

The *iffamily* Structure

Besides other *ifl*'s, the structures which can be added to an *iflogical* structure are *iffamily* structures, or *iff* for short. An *iff* is added via a procedure handle in its parent *ifl*, and that code will have some opinion about what is and isn't appropriate to add to this type of interface. One *iff* is added to an *ifl* for each "network layer" protocol configured to operate on the *ifl*. While we take a rather expansive view of what constitutes a "network layer" protocol, this certainly includes the layer 3 protocols like IP and IPv6 along with anything else which is brought forward (Appletalk, SMB, etc.). In many cases adding the *iff* is all that is necessary to enable a protocol on an interface; this is certainly true for IP and IP6 on point-to-point interfaces, and may also be the case for IP6 on multicast interfaces if we auto-generate a link-local address in the kernel. In other cases some additional address configuration on the *iff* may be necessary to make the protocol family fully functional.

An *iff* is named by the *ifl* it is configured on plus its own protocol family identifier. The latter can sometimes be deduced from context, so when "fxp0" is mentioned in a context which implies the IP family, like when configuring an IP address, this names the IP *iff* configured on the *ifl* named "fxp0.0". The *ifl_index* similarly finds an *iff* in contexts where a particular protocol family is specified by the protocol family's identifier, or implied by context. In particular the includes forwarding operations, which always understand the protocol from the context (the code knows what protocol it is forwarding) and often refers to the incoming or outgoing *iff* by its *ifl_index*.

The *iffamily* structure is specifically the corresponding network protocol's view of an "interface". With respect to the L2 encapsulation header description provided by the structures below, the *iff* completes it by implicitly providing the L2 protocol type in the header (when one is required in the encapsulation). More than this, the *iff* is referenced by packet forwarding operations since it contains information and procedure handles necessary to know how to process packets arriving from or being sent out this interface. In particular, I would expect the latter to include:

- A filter to be applied to incoming packets and a filter to be applied to outgoing packets. I would expect each filter would be an object independent of the *iff* and hence to be identified indirectly by a "filter index", similar to the *ifd_index* and *ifl_index* number spaces which we've acquired already. What a filter does is the subject for future development, but I would expect that it would provide a mechanism to build firewalls and/or describe nontraditional or nonstandard packet forwarding procedures.

- A routing instance identifier to indicate the routing table which packets arriving on this *iff* should begin their forwarding lookup in and where interface routes required by interface address configuration would be installed. I'm pretty sure the routing instance would also be identified by its own index space.
- A procedure handle to call to deliver broadcast and multicast packets out the interface, which would be expected to add an appropriate L2 header to the network protocol packet (if necessary; it often isn't when the "network" protocol is L2 bridging) and then deliver the fully encapsulated frame to the *ifd's* output queue for transmission.
- An MTU for the network protocol. This information is required for fragmentation and MTU discovery, and for transport protocol segmentation (if the underlying hardware isn't doing that; if it is this MTU might instead have made it down to the hardware).
- A "control" procedure handle to receive certain configuration requests from the protocol family code, for example multicast group joins or the initiation of a multicast routing protocol on the interface. When necessary the family code would translate this into requests for the hardware like modifications to an Ethernet interface's multicast MAC address filtering.
- Certain hardware capability flags, for example whether the hardware checks or generates checksums for this protocol or the above-mentioned transport protocol segmentation, are also stored here. The reason for this is that, while the capabilities may be a function of the hardware and hence the *ifd* underlying the configuration, the software which needs to adjust its behavior based on these is software that views this *iff* as the interface. Moreover, the hardware capabilities may not apply uniformly to all *iff's* configured on the *ifd* (perhaps PPPoE *iff's*, or even *iff's* configured on VLAN-tagged *ifl's*, might be on their own) so these capabilities generally require expression on each *iff*.

The *iff* also, not surprisingly, has procedure handles that are called in configuration to add dependent structure. We'll get to this when we examine that structure.

Generally only one *iff* with a particular protocol family identifier can be configured on a single *ifl* (since the protocol family identifier is the distinguishing part of the *iff* name, the naming convention itself enforces this), but as many *iff's* with unique protocol identifiers as we need to describe the network protocols we are willing to process on that (*ifl*-defined) interface might normally exist simultaneously. For *iff*s which represent true layer 3 protocols it will likely often be the case that *iff's* for different families may be intermixed freely on the same *ifl*, though this rule has its exceptions (like SLIP interfaces, which encapsulate IPv4 exclusively). To extend this, I would also expect it to be reasonable to add additional *ifl* configuration on an *ifl* which is also a parent to *iff's*, for example to add a PPPoE session to an Ethernet which is also being used for Ethernet-encapsulated IP. The broad definition of "network protocol" that we adopt for representation by the *iffamily* structure, however, creates a variety of other exceptions to this, in particular *iff's* whose configuration is meant to represent a "network protocol" which actually implements some sort of layer 2 functionality.

The best example of this is probably Ethernet bridging (or "switching", as it is now trendy to call it). I would imagine that an interface would be configured for Ethernet bridging by adding a "bridging family" *iff* to an *ifl,* which in turn is perhaps configured on an Ethernet hardware *ifd.* Note, however, that the "network header" which bridging operates on to make forwarding decisions is in fact the Ethernet header; bridged "packets" are the full Ethernet frames which arrive from a Ethernet hardware so, since there is no protocol demultiplexing layer outside this header, the configuration of an Ethernet bridging *iff* precludes the configuration of any other protocols sharing the same packet stream. I would expect this to mean that an attempt to add an Ethernet bridging *iff* to a configuration would succeed only if the Ethernet had just a single *ifl,* with no L2_ID and with no other *iff*'s configured, and once the Ethernet bridging *iff* was successfully added I would expect attempts to configure other protocol families or *ifl*'s to the interface would fail until the bridging *iff* was removed. This would work about the same way whether the Ethernet *ifl* the bridging *iff* was configured on was itself configured on an *ifd* representing a real hardware Ethernet interface, or on a PPP or Frame Relay *ifl* which was delivering encapsulated Ethernet frames.

To elaborate on the Ethernet bridging example it is interesting to consider what the contents of the bridging *iff* would be. The routing instance would, analogous to other protocols, indicate a routing table for the bridging code to do a forwarding lookup in. The interfaces whose packets were forwarded using a common routing table would be the interfaces exchanging bridged Ethernet frames with each other, so the routing instance identifier would essentially become the bridge group identifier; all interfaces with a bridging *iff* configured with the same routing instance are members of a bridge group. The filter indices could be used if someone figured out something useful for bridging filters to do, or just ignored otherwise. The procedure handle for delivering broadcast and multicast packets would be called to deliver broadcast and multicast bridged frames; this operation might be as trivial as queuing the packet to the hardware output queue unchanged, for a real Ethernet interface, or might involve adding additional L2 headers to the packet before giving it to the device when the hardware is, say, running Frame Relay or PPP. Since L2 bridging and L3 routing are very nearly the same kind of operations, differing mostly by the fact that they look at different data in the packet to make their forwarding decisions, their basic protocol family configuration requirements are likely to be analogous.

Of course the fact that a bridging family configured on an interface precludes configuration of an IP family on the interface may raise the question of what happens if one wants to run IP on that interface? The answer is that you really don't want to run IP on that particular interface. You want to run IP on the entire bridge group, which may have many hardware interfaces, so putting IP configuration on any one of them makes no semantic sense (another way to express this is that the entire bridge group represents a single multicast scope to IP which you want to be able to configure as a single "interface" for IP even if it looks like several interfaces to the bridging code). To accomplish this we use a pseudo-interface which does allow (require, actually) a bridging family to be configured along with layer 3 (including VLAN) configuration, with the pseudo-interface appearing to be another member of the bridge group to the bridging code (via the bridging family) and a vanilla Ethernet interface to layer 3 protocols. Ethernet frames passed through from the bridge would be processed through the normal Ethernet decapsulation and demultiplexing code, with a change of its incoming interface index to be the *ifl_index* of one of the pseudo-*ifl*'s, while Ethernet frames encapsulated by the layer 3 code and sent via the pseudo-*ifd* would have their incoming interface index changed to the *ifl_index* of the bridging family and would then be processed through the bridge code for transmission. Neither the bridging code nor the L3 code would need to be aware the pseudo-interface was anything different than a "normal" Ethernet hardware interface, that knowledge would be fully encapsulated in the pseudo-interface driver itself. Other link layer operations which bind several hardware interfaces in a way which requires them to be seen as a single interface at layer

3, for example Aggregated Ethernet and Multilink PPP, might be handled similarly, with an Aggregated Ethernet or Multilink PPP protocol family being configured on the hardware's *ifl* and a pseudo-interface holding layer 3 configuration, with the link layer family routing instance identifier indicating the grouping.

In summary, the *iffamily* structure is a protocol family's view of an an interface. It serves both to define the processing of that protocol's packets through this interface and to hide the details of the underlying hardware and packet demultiplexing from the protocol.

The *ifaddr* structure

The structure that can be added on an *iffamily* structure is the *ifaddr* structure, or *ifa* for short, which represents local addresses and/or addresses of directly reachable neighbors as appropriate for the *iff*'s protocol family. An *ifa* is added via a procedure handle in the *iff* it is being configured on, which is the arbiter of good taste with respect to what addresses are appropriate. For some protocol families (e.g. perhaps the link layer protocol families described above) no address configuration is appropriate or permitted. For other protocol families *ifa* configuration may be optional, or may be required for the interface to be useful. The types of addresses configured by an *ifa* are those appropriate for the protocol family they are configured on. As a general rule we consider interface address configuration to be optional to the extent permitted by the underlying protocol family, and attempt to treat the configuration of an *iff* on an interface as sufficient to make the protocol active on the interface (e.g. IP DHCP should not need to rely on addresses being configured on the interface to send and receive the packets required to learn addresses to configure), but many protocol families require address configuration to make the interface generally useful. We also permit multiple *ifa*'s to be configured when this makes sense to the protocol family.

Each *ifa* configured on an interface holds two addresses, a destination address and a local address. The destination address, which may be a prefix (i.e. have fewer bits than a host address for that protocol), identifies the address(es) of neighbor(s) directly reachable via this interface while the local address, which in general will be an address identifying an individual host for this protocol family, is an address which the system will subsequently recognize as identifying itself. An *ifa* can include both of the addresses, or just one or the other, with the suitability of the configuration for a particular interface being adjudicated by the protocol family (for example, the loopback interface is implemented as a point-to-point pseudo-interface where *ifa*'s are constrained to have local addresses only). Each *ifa* is named in configuration by its value, i.e. the two addresses it holds, plus the name of the *iff* (i.e. the *ifl*'s name string or *ifl_index*, plus the protocol family identifier) it is configured on, so the tuple of destination address plus local address is required to be unique for each *ifa* configured on an *iff*, though either address may individually appear in multiple *ifa*'s. Nonsensical configurations, for example the local address in one *ifa* appearing as the destination address in another, are strongly discouraged but may not be actively prohibited.

Beyond these general rules it is difficult to discuss the *ifa* divorced from the context of a particular protocol family, so the following considerations will apply to IP and IP6 (though we would expect other protocols to work in an analogous fashion to the extent possible). On point-to-point interfaces we take a liberal view of *ifa* configuration, and adopt the traditional semantics of Unix point-to-point address configuration. The destination address in an *ifa* is a host address which we think the system at the other end of the link will recognize as its own, while the local address is a host address we recognize as our own; a route will be installed for each address to implement this. The local address may be unique to this interface, in which case the configuration of the address will cause us to do the additional work required to inform the rest of the system that the address is "local" (beyond installing a route the address may effect local address selection state), or may be a non-unique local address that is configured on other interfaces as well, in which case the presence of the local address on this interface will solely effect local address selection for applications which send packets out the interface without pre-specifying a local address to use in those packets. The destination address may be missing, say if we don't know an address that the remote system would recognize as its own; we can still send packets to the remote system as multicasts or broadcasts, and we can always send packets through the system at the other end in any case using the *iff*'s name (*ifl_index* or name string) to specify the next hop neighbor. The local address may be missing if the default local address selection which the system implements when no local address is specified on the interface provides an acceptable result. And, in fact, we can make use of an IP or IP6 point-to-point interface to send and receive packets without an *ifa* configured at all; the name of the *iff* is sufficient to identify the system at the other end of a link as a destination both in the routing table and for multicasts. It is the case that IP6 may automatically configure an *ifa* with a link-local local address when the IP6 *iff* is added, but nothing in the kernel particularly relies on it and I suspect this will be possible to disable on a per-*iff* basis in case some user-space configuration application has different ideas about the best way to configure a link-local address.

For multiaccess interfaces, IP generally constrains *ifa*'s to subnet addressing. That is the destination address in an *ifa* must be a prefix no more than 30 bits long, and the local address must be a host address matching the destination prefix with neither the all-zeroes or all-ones host part. Furthermore, when multiple *ifa*'s are configured on an *iff* they must not have overlapping destination prefixes (i.e. destination prefixes of different lengths which match to the length of the shorter one), but multiple *ifa*'s with the same destination prefix (distinguished by unique local addresses) are possible as are multiple *ifa*'s with different, non-overlapping destination prefixes. IP6 relaxes this a bit in that, while it does do subnet addressing, it also allows *ifa*'s with only a local address to be configured (say, the link-local address). Like point-to-point interfaces, *ifa* configuration on multiaccess interfaces causes routes to be added and influences local address selection (which for IP6 is significantly more (and unnecessarily?) complex), but also may interact with address resolution procedures (say, ARP, or ND) to cause additional neighbor routes to be learned and added. Routes generated as a side effect of *ifa* configuration end up in the routing table indicated by the *iff*'s routing instance identifier. Even on a multiaccess interface, however, we consider the interface "up" for a protocol when its protocol family is configured even if no *ifa*'s have been configured, through recognizing that a multiaccess interface without addresses may be of limited use (e.g. maybe just DHCP on an IP interface).

It is worth pointing out at this point that, while there are a few constraints on addresses which can be configured on a single interface (i.e. *iff*), like avoiding overlapping subnets, we place nearly no constraints on what addresses can be configured on different interfaces. We generally presume that the person or automaton doing the configuration knows what it is doing and will do what it needs to get the result it requires. The possible existence of multiple routing instances or other arrangements even constrains us from assuming that direct neighbors having the same address but reachable through different interfaces are the same neighbor. As a general result of this, the identity of a neighbor can hence be specified unambiguously by protocol address only in the context of a particular *iff*, that is by the tuple

<*iff* name><protocol address>

where <*iff* name> might be expressed by an interface name string or the *ifl_index*. For a point-to-point interface, where there is only one neighbor, the

<*iff* name>

without a protocol address is sufficient by itself to identify the neighbour, though we might expect the semantics of this form to differ slightly from the form which includes a protocol address, in particular in the behavior when the *ifa* having the protocol address the latter depends on is deleted from the *iff*. The implication of this is, of course, that identifying a route's next hop neighbor by

<protocol address>

alone is being made increasingly ambiguous (it already was) by both new functionality and by the relaxed constraints on interface addressing. While this form will continue to be supported, the situations where it may not do what you want are increasing.

A specific problem with address configuration is also interesting to discuss here as it illustrates a more general approach to the problem of interface configuration changes which we'll adopt in all structures. As was mentioned, *ifaddr* structure configuration has the side effect of installing routes representing those addresses. Beyond this, the existence of that address configuration may prompt interface level routing protocols to discover and install routes to other neighbors on the interface, so there may potentially be a large number of routes whose presence ultimately depends on the interface address configuration. This means that should we decide to remove an *ifa* we are faced with the task of removing all routes which are dependent on that address configuration, i.e. which wouldn't have been there were it not for the *ifa* having been configured. This is fine, it is part of the kernel's task; all configuration must be dynamic. It may prompt another question about interface changes, however: since routes are installed in the table indicated by the *iff*'s routing instance, what happens if the configuration change is a change to the family's routing instance? There would seem to be two possible answers: we could remove the routes from the old table and add them to the new table, or we could refuse to make the change if there were any *ifa*'s configured on the interface, leaving it to the user space configuration application responsible for understanding that implementing the change requires first removing the *ifa*'s from the *iff*, then changing the *iff* routing instance, then adding the *ifa*'s back. I don't know which way we might end up doing this in implementation, but the implementation reserves the right to do it either way, not only in this case but in all cases where the presence of interface structures higher in the tree complicates making a change to the configuration of underlying structures. While we might argue about which way is "cleaner" or "better", for some value of those, we retain the goal of making code in the kernel no more complex than necessary. If implementing a change to a fully configured interface seems too onerous and/or unlikely to occur more than rarely in practice, we reserve the option of simplifying the kernel code in favor of leaving a more complex operation for user space applications.

Finally, I would point out that address configuration hasn't really changed much from the traditional BSD networking stack, though the way we view it may have morphed a little. We no longer depend on address configuration to tell us the corresponding protocol is "up" on the interface (we have the *iff* for that), and this has the side effect of allowing us to support unnumbered point-to-point interfaces. When there are multiple addresses configured we don't treat one *ifa* as "primary" and the others as "aliases", they are all treated as equal siblings. We deal with routing instances when they are used, but any complexity associated with that goes away if they aren't (routing instance 0 is likely to be the default when a routing instance is left unspecified). The biggest changes involve our treatment of interface routes as this allows us to, say, configure two interfaces on the same subnet without annoying interactions; we'll get to that. We'll try hard to remove any addressing restrictions which are strictly unnecessary from the kernel's point of view (even those which some standard might require), but leave configuration utilities free to observe those restrictions should they continue to be useful. Our standards compliance may not be rigorously enforced in the kernel as long as configuration utilities and other code can produce configurations which rigorously comply with all standards.

Pseudo-Interface Usage

Pseudo-interfaces are used to represent packet processing or state instantiation operations that can leverage the configuration infrastructure required by "real" interfaces to get their job done as well. While much of the above description has focussed on the per-structure semantic policies which are observed when using these structures to represent the configuration of "real" interfaces, for pseudo-interfaces we view these structures instead as solely providing mechanism, leaving policy to the pseudo-interface driver. The structures are tools to get your job done, to be used when the configuration support (if it is used; pseudo-interfaces whose upper level configuration is produced automatically by some other kernel process are certainly possible) and/or the existing interactions of interfaces with routing produce a useful result which makes that job easier. The interface arrangement described above is intended to ease its use for pseudo-interfaces by allowing *ifl*'s (or *ifl*'s/*iff*'s) to be added and deleted with abandon, and to make "interfaces" officially useful even if no addresses are configured, saving that cost if addresses aren't useful (while not prohibiting pseudo-interfaces whose whole point is address configuration; the policy is up to the pseudo-interface driver).

We've already encountered several uses of pseudo-interfaces, and while I don't think these are typical examples it still might be useful to point out the rather divergent properties these can have and perhaps expand a bit on how they might be used. There is a class of pseudo-interfaces which are actually hardware interfaces for hardware which is not unambiguously "networking" hardware, with the particular hardware example being asynchronous serial interfaces and the pseudo-interface drivers being those for SLIP or async serial PPP. We would expect these to be configured as needed by adding an *ifd* to the underlying *ifdriver* structure for each serial port we want to use for this, with the *ifd* configuration including an indication of the hardware to be used. Beyond this, however, we would expect interface configuration on the *ifd* to observe the semantic policies of "real" interface configuration since, once the *ifd* is added, the interface is a "real" hardware interface.

The second class of pseudo-interface we've seen are those which are required to represent some L2 operation involving multiple hardware interfaces as a single interface entity to protocol users of the interface, in particular as an L3 interface to an L2 bridge group or as the single L3 representation of an interface for L2 procedures like multilink PPP or aggregated Ethernet. These interfaces look and act very much like hardware interfaces carrying the corresponding types of L2 packets but the entry points provided to higher level code, which might normally be expected to queue packets to the (singular) output hardware, instead run more complex procedures to determine the hardware the packet will ultimately use to leave the system. In this case the pseudo-interface configuration allows us to reuse the common code for building interface configuration and processing packets for interfaces of this type while inserting additional code to do the processing unique to the arrangement of hardware interfaces the pseudo-interface was invented to support.

The third type of pseudo-interface we've mentioned is the ancestor of all pseudo-interfaces, the loopback interface "lo0". It is worth noting up front a rather subtle semantic shift in our treatment of the loopback interface (this should be clearer later). The current BSD networking stack sometimes uses output through the loopback interface from forwarding to indicate that the packet should be received by the local host (as opposed to sending it through an interface to someone else), though sometimes it doesn't bother with this (for example, when looking for local addresses in incoming packets, which is not considered "forwarding"). We never do this; in particular we prohibit (as semantically fraught) the installation of routes with a next hop identified by a protocol address which is a local address of the system (i.e. configured as the local address in an *ifa*) in favor of just expressly using a next hop which directly indicates that the packet is of interest to the local system. Since the loopback interface only has local addresses configured packets are never sent to the loopback interface, and since packets also never come from the loopback interface (locally originated packets come from *ifl_index* 0), the loopback interface is not an "interface" in the sense of being an object that (even logically) sends and receives packets.

Instead the loopback interface exists solely as a place to configure protocol addresses that we want the system to treat as local (i.e. which identify "us"), but which we (may or) may not want associated with interface configuration elsewhere. The IP loopback address, 127.0.0.1, and its IP6 equivalent are the obvious examples, and likely the only examples most people will ever see, but "normal" addresses that we want to be favored by local address selection procedures or which we want to exist apart from "normal" interface configuration (the latter is a common need on routers) can be added to the loopback interface as well. This is done by configuring the loopback interface with a point-to-point *ifl* and an *iff* for each protocol which needs one, and then allowing *ifa*'s which are added to only have local addresses; beyond this latter constraint it looks like a "normal" p2p interface for address configuration purposes. Note, however, that we have added additional complexity here by the addition of routing instances: since the routes installed as a side effect of address configuration are installed in the table identified by the *iff*'s routing instance, and since we will in general need to install loopback routes in each routing instance table, the presence of multiple routing instances will in general require the configuration of multiple loopback interfaces. We will probably do this by adding multiple *ifl*'s ("lo0.0", "lo0.1", ...) to the single loopback *ifd* (the alternative is adding multiple *ifd*'s, but that is more data structure for no apparent advantage) as required to configure an *iff* per routing instance per protocol. We may also use these *iff*'s to configure filters to apply to packets originated by and destined to the local machine via each routing instance, though there may be more convenient ways to do this.

The pseudo-interface applications already mentioned, however, are only a small subset of the possibilities. Pseudo-interfaces are generally useful in cases where an interface representation allows existing code reuse for route installation and packet delivery while providing a place to implement the functionally unique code to process and transform the packets it has leveraged the standard code to deliver. For example, encapsulation links are often very usefully represented via pseudo-interfaces, with different encapsulation destinations being represented at the *ifd* or *ifl* level as convenient. The delivery of the inner packets may be well represented as an "interface output and input", while the encapsulated packets might, in addition to having additional headers inserted, naturally appear instead as locally originated (i.e. having an incoming *ifl_index* of zero) and received (via an address/protocol demux) packets. A NAT implementation might be done as a variant case of this, with packets passing from one routing domain to another. That is, the NAT router might maintain two routing domains, the “public” domain and the “private” domain, with interfaces being configured into one or the other and with the public and local routes kept in different tables. The NAT operation would then appear as an interface to the private routing domain, and perhaps be configured as the default route’s output interface, but on the public routing domain’s side would appear as a source of locally originated packets and a destination for incoming packets of “local” interest, with the interface configuration providing a place to configure additional filters for packets traversing between the two domains.

Pseudo-interfaces don’t solve all problems of building descriptions of complex network processing sequences, but pseudo-interfaces plus the expressive and flexible configuration of network interface stacks can provide a natural expression for many of them, reusing existing routing and interface configuration code to the extent it is useful for getting one’s job done while providing a logical way to partition and insert new functionality into existing packet handling flows. I would expect pseudo-interface development to be a focus of future networking development.

Route Tables, Next Hops and Protocol Demultiplexing

This first section attempts to describe what the route lookup procedures adopted here attempt to achieve and how that differs from the current kernel. It will the go on to describe a particular implementation of routes and next hops which do the right thing, and continue to cover what happens to packets that are determined to be of possible interest to applications on the local system. This discussion is IP- and IP6-centric (indeed mostly IP-centric, since IP6 is often very similar; where it isn’t and the difference matters we’ll try to point that out) because the needs of different network protocols are not perfectly aligned, it is hard to get very far discussing anything outside of a particular context, and IP and IP6 are certainly the most important protocols to support well. While the details of other network protocols might differ, however, I hope the general approach to the problem will translate well.

The implementation of a network protocol having addresses fundamentally requires a facility to use the address data in a packet along with other relevant information, like where a packet came from, to determine for whom and how to process that packet. We represent this facility with two kinds of objects: *routes*, which contain addresses and other information about the packet whose subsequent processing we need to determine, and *next hops*, which inform us about what to do to further process the packet. Each route points at a next hop; we generally want these structures to be separate since in many important cases there can be many more routes than next hops (i.e. many routes can point at the same next hop) so it is often more space-efficient to split them, and also because the processes by which next hops are determined can sometimes cause them to change, a problem that is easier to deal with if there is just one next hop object effected by the change (which all routes pointing at the next hop will find out about since they all point at the same object) rather than many routes needing an independent change. The basic process, then, is to extract the address data from a packet, use this as a key to index into a map to find the matching route, and then use the next hop pointed to by that route to process the packet out to where it is going.

So far we've used the term "address data" without defining just what that is. The answer to that depends on just what we expect the facility to do for us in the most specific case (since we're extracting this data prospectively we need to get all the data the operation might require up front). For example, for a packet of interest to an application on the local machine what we're ultimately going to want to address is the socket that application is waiting for the data on and the transport protocol code we'll need to run to move the data from the packet, which would imply that "address data" might usefully require (for IP)

<destination address><source address><IP protocol><first 4 bytes of transport header>

as this, in many cases, is sufficient to get us all the way to where we want to go. A router implementing NAT might also want to support this operation to identify packets for individual applications running on other systems. While the approach of jumping all the way to the final result in one go is attractive to think about (despite its imperfections: it needs a broad exception path since, in the presence of fragmentation, the data it needs isn't even guaranteed to be in the packet we're looking at), it is a step too far for the current purposes (which you could think of as designing a very good exception path for the former scheme if you like). We instead define our route lookup to operate on the key

<destination address><source address>

as it is all that is reliably available in every packet (while not precluding the future use of a longer key if someone can figure out how to use that here), assuming that this will be sufficient to generally categorize the packet (this packet is "local", or needs a NAT operation, or needs to be sent to this neighbor) and assume that any more specific address lookup required to find a socket, or something, will be part of the processing done by the next hop for the category of packets. Note that the particular reason to include the source address in the key is that multicast forwarding routes require it (those routes include the entire destination address and a prefix of the source address), though it is always available if it is useful for something else too.

In any case, we will hence build a routing table structure which, given a key, allows us to perform lookups to find the route with the longest matching prefix for the key, and to then use the next hop that route points at to tell us how to deliver the packet where it is going. Other than the two-address keys, and the fact that we'll have one of these tables per routing instance, this probably doesn't sound a whole lot different from what the kernel does now, so we'll try to point out what is changing here.

The first thing that is changing is the structure itself. I believe forwarding lookups will get significantly cheaper than the current structure measured by cache lines required to be read to get a result (i.e. the lookup won't progress through the key doing just a single bit test per memory reference). The forwarding cache implemented by the current kernel must go for reasons far too numerous to mention, so some effort will be directed to removing any reason to want that structure by making full forwarding lookups cheap enough to remove any desire to "improve" this by adding yet another place to store routes. To reduce the overhead and ease the design of a system which allows productive concurrent use of the network stack the route table structure will allow lookups to be performed concurrently with the table being modified, without locking. That is, while we will use a lock to ensure that only one modifier of the table is active at a time, readers of the table (e.g. processes delivering packets) will continue to make progress at the same time. The guarantee we'll make to readers is that, during a change to the table (i.e. a route add, delete, or change), they will see only the pre-change state of the table or the post-change state of the table, and nothing else. Route lookups not involving the changing route will see no effect of the change at all. I would also expect the structure to be done in a way which would make it equally usable in a user space library, both because a good route lookup structure should be useful for applications as well and because this makes basic maintenance and debugging easier.

The second thing that is changing is how we use the table, that is how we arrange processing around the route lookup. Understanding this change is probably helped by understanding how the kernel works now and the difficulties it deals with. The route table which is maintained in the kernel now contains routing information in a form which is generally appropriate for packets originated by applications running in the system; you can take the address key from a packet generated by an application, look it up in that table, and get a result which tells you what to do with it. The problem with this is that routes matching some destination addresses need to do something different with a packet received from an interface then they do with a packet originated by a local application. For example, if a local application sends a packet to a broadcast address, either a subnet broadcast or 255.255.255.255, we expect the packet to be sent out the appropriate interface as a broadcast and so a route lookup whose result is "send the packet out an interface" is appropriate. If a packet to the same address comes in an interface from outside the box, however, we instead want to deliver the packet to an application on the local system (or, at least, see if any application on the local system is interested in it), so routing a received packet with one of these addresses in a routing table where the result is "send the packet out an interface" gives the wrong result. The same originated-and-received-do-different-things problem exists for multicast-addressed packets (on a non-routing host) and for packets addressed to a zero-host-bits subnet address or 0.0.0.0 in a slightly different form (received packets should be processed locally, but originated packets should be dropped with an error).

To deal with this (and ignoring IP options or the IP6 hop-by-hop header while describing this), when a packet is received from an interface and after its IP header has been verified, the current kernel tries to catch packets with these problem addresses by doing a lookup in an entirely separate structure (i.e. not the routing table) to determine if it is a "local" address (in fact calling it a "structure" is overly generous; a lot of the comparisons are inline code). If so it receives the packet locally to see if anyone is interested. If not it "forwards" the packet, which means it drops it if we're a non-forwarding host or it checks and decrements the TTL if we're a router, and if the packet survives that a second lookup is done in the main routing table (now that the problems are avoided) to figure out what to do next. Note, however, that this procedure is (of course!) broken for multicast-addressed packets in a number of ways (including the fact that a multicast router can be required to deliver the same packet to the local machine, un-forwarded, and out interfaces, forwarded, so thinking you will be doing one or the other but not both is wrong), so multicast-addressed packets are instead shunted through a bag-on-the-side with yet another lookup table and a different set of procedures. These multiple tables (which, in the presence of routing instances, multiply by the number of routing instances) and kitchen-sink code paths are problems we seek to solve.

In the current design what is proposed is building a single routing table (per routing instance) having all information required to determine the packet's destination, including multicast routes and local addresses, to make this structure efficient enough that the cost of using it to forward every packet is not significant, and to use this to correctly forward all packets no matter what their destination and where they came from. This gives us one single table, per routing instance, to store routes and do lookups in, and mostly avoids us having to look at the addresses apart from the lookup in this table. Each forwarding lookup in the table will provide the two-address key, and will receive as a result the route in the table with the longest matching prefix of the key (there will always be one since we'll always keep a 0/0 route in the table). If the process decides the packet can be forwarded with this route (there may be constraints based on the incoming *ifl_index* for the packet) it will then deliver the packet to an output function in the route's next hop. It is the responsibility of the next hop output function to do the appropriate thing with any packet given to it, including dealing with any differences in processing for originated and received packets (which we might distinguish by explicitly specifying "originated" or "received" as an argument to the output function, but which can be determined in any case by checking to see if the packet's incoming *ifl_index* is zero or non-zero).

Moving the output function to the next hop, from the interface structure where the current kernel puts it, has some advantages. It means that next hops now don't have to be associated with a particular interface when there is no particular interface (e.g. multicast and 255.255.255.255 broadcast next hops, error or discard next hops, local next hops and any type of next hop which gets invented to do a non-interface "thing" in future). Better, it avoids the loss of context which the current interface function (one of them, called by all protocols for output) suffers from, where a protocol, which knows exactly what protocol the packet is because, well, it's the protocol, calls a function which immediately does work to figure out the packet's protocol so it can build the right kind of L2 header. Here the next hop is protocol-specific and knows what kind of packets it is going to get, so a unicast next hop could conceivably operate by keeping a local, prepared copy of the full L2 header the packet needs to reach the neighbor the next hop represents, then taking each packet it is called to process (after forwarding, if the packet was "received", and maybe filtering), copying the L2 header directly onto the front of the packet and putting it on the hardware interface queue without any extraneous fooling around. The arrangement provides a considerable amount of flexibility and, in many cases, the efficiency of being able to get quite directly to the point of the operation.

There is one other problem we are changing the route table structure to address, one which might be a bit easier to explain by understanding the distinction which routing protocol developers make between a forwarding table and a routing table. A typical forwarding lookup done, say, when you have a packet whose further processing you need to determine, is keyed solely by address information in the packet. In this situation the only information you have to make a decision about which route in the table to choose is the address information and the length of the address prefix in a matching route, so there is no way to distinguish between two routes with the same prefix. Thus, in a "forwarding table", there can be only one route to any particular prefix since the key for a lookup has no information which would allow us to distinguish between more than one. Note that routing protocols often learn more than one route to the same destination, say via different neighbors or from different protocols, so in contexts where you have more information than an address key to distinguish between different routes with the same prefix it is sometimes useful to be to be able to store routes in a "routing table" which can track more than one route with the same prefix, distinguished by other data associated with the route. A forwarding table is then constructed from a subset of the routes in the routing table, generally by applying a preference function to decide which of the routes with a common prefix we like best.

The route lookup structure in the current kernel seems to be a “forwarding table” since it can only store one route per prefix. This causes difficulties in situations where there is more than one route to the same destination prefix which we would like to store even if only one of them can be used for forwarding. A particular example of this occurs when you would like to configure two Ethernet interfaces onto the same subnet; the interface configuration will generate routes to the same destination prefixes since the same destinations are reachable using either interface. While only one route per prefix can be used at a time for “normal” forwarding we would like to track all of these interface routes since they can be useful for other types of route lookups, for example to find a next hop to use for another route (next hop specifications include the interface, as we saw above, so this can be used to distinguish between routes to neighbors through different interfaces) or to route `SO_DONTROUTE` packets originated by the host which indicate an interface in packet metadata. More than this, we would like to keep all these routes in the same table that is used for forwarding since, while any particular route might not be currently used for forwarding if another route with the same prefix is preferred, we would like the route to be available should the preferred route be deleted. For these reasons our routing table data structure will store multiple routes to the same prefix, ordered by preference, and will support two kinds of lookups: a “forwarding” lookup, which takes only address data as a key and returns the most preferred route to a prefix, and a “route” lookup with takes additional data to key its search. And while the routing table is generally useful it might be interesting to note the implications of the interface route example; if interface routes corresponding to each *ifa* are always in the routing table (they almost always are now, but you can’t rely on it) then we can eliminate the need to do a linear scan of *ifa*’s to find interface routes, as is done now, which in turn eliminates a potential scaling problem if a large number of interfaces is configured.

In summary, then, we are eliminating a route cache, a “local” address lookup table, a multicast forwarding table, a unicast forwarding table and procedures which find interface routes by doing linear scans of configured interfaces in favor of storing all routes and performing all these operations in a single structure (per routing instance). We plan to support very fast forwarding lookups and efficient route lookups in the structure, and we plan to do so in a way which allows lookups to continue concurrently with an in-progress table modification. We continue by discussing routes, which are indexed in the table, and next hops, which are the results we generally perform route lookups to discover.

Routes, Route Types and Route Contents

Each route has the following contents (note that this is a work-in-progress, and may change as development requires it):

- A route type. Route types document who installed the route and/or why the route was installed. Route types are also among the attributes we use to distinguish different routes with the same prefix, and contribute to our route preference weighting.
- A prefix and a prefix length. The prefix length is the number of significant bits in the prefix and may take values between 0 and at least 256 (an IP6 prefix may be 256 bits long). The prefix itself is variable length byte data stored in network byte order, but stored in a 32 bit aligned field and padded out with zeros to a multiple of 32 bits equal to or greater than the prefix length. The 32 bit alignment allows us to make prefix comparisons 32 bits at a time.

- A 32 bit *ifl_index*. For routes installed as a result of, or otherwise dependent on, *ifa* configuration this field will be set to the *ifl_index* of the *iff* the *ifa* is configured on. For other routes this field may be set to an *ifl_index* we need to match to the incoming *ifl_index* of a packet we are forwarding (see the `RTF_IFCHECK` flag below). Note that this overloading of the field may end up biting us, so it is possible (though unlikely) that there may end up being two *ifl_index* values in the route instead.
- A flags field containing at least the following flags:
  - `RTF_IFCHECK` indicates that the incoming *ifl_index* in a packet being forwarded should be compared with the *ifl_index* in the route, with the packet being discarded if they do not match. This is used in multicast forwarding routes, and may be useful in routes representing local IP6 link-local addresses or on routes to local addresses should one wish to implement strict host behavior.
  - `RTF_LOCAL` indicates that the incoming *ifl_index* in a packet being forwarded should be compared to 0, with the packet being discarded if the *ifl_index* is non-zero. This is useful for identifying routes which should only be used for packets originated by the system, for example loopback addresses and perhaps routes to the IP6 link-local addresses of neighbors.
  - `RTF_SCOPE` may be used to indicate to IP6 that the route should only be used for packets with a source address matching the prefix's scope. This may (or may not) be useful for routes to IP6 site-local destinations.
- Perhaps a reference count (or 2) to assist in optimizing clone route operations (we'll get to that). Note that these are not incremented during forwarding operations (i.e. they won't cause cache problems for multiprocessors) but rather during route table modifications.
- A pointer to a next hop. The next hop this points to is the ultimate result of a route lookup; its output routine is called to forward the packet on.

We distinguish routes by the prefix, the route type, the route *ifl_index* for some types and the next hop owner for others. Different routes, that can be present in the table at the same time, must have different values in these fields. For routes with the same prefix we keep them sorted in preference order, with the preference being defined by the distinguishing fields just mentioned (user-defined preferences are also possible, but are for future consideration). The best preference route with a particular prefix is the route used for forwarding.

Note that, because of the routing+forwarding table design, we also support "ignored" route types, that is routes which may be present in the routing table but whose presence is ignored for forwarding operations (that is, forwarding lookups match a shorter prefix instead). We do this for two reasons. First, it allows us to keep routes in the table for purposes other than forwarding. Second, in the presence of forwarding routes which cannot be directly removed from the table (e.g. interface routes, which will remain in the table for as long as the corresponding *ifa* configuration exists), a preferred "ignored" route provides a way for user processes to remove other routes from consideration in a forwarding lookup.

These are the initial route types we will support (others are possible, as needed) in order of decreasing preference:

- `RTT_USER` routes are routes installed by "normal" user routing processes, e.g. 'route add' or a routing protocol implementation. Routes installed by a user are always preferred over other routes in the table, and hence always effect the forwarding table. We currently only allow one user route per prefix; an attempt to install a new `RTT_USER` route will replace an existing one with the same prefix. While the structure itself would allow other possibilities, existing routing applications have worked well treating the kernel table as a forwarding table (it is interface and other routes installed by kernel processes which have been problems) so we'll continue with this until someone discovers a good reason not to.

- `RTT_USER_IGNORE` routes are user routes (i.e. they replace `RTT_USER` routes with the same prefix, and vice versa) which are ignored for forwarding. They exist to allow user space processes to hide other routes in the table (e.g. interface routes) from the forwarding process. I doubt these will be used, but functional considerations suggest that we should leave user space process in full control of the forwarding table and these routes seem necessary to ensure that is possible. Note that we to make it an error to install an `RTT_USER_IGNORE` route with a 0/0 prefix to allow forwarding lookups to always return a route.

- `RTT_PERM` routes are "permanent" routes which are added to a table when it is created. These routes can't be deleted, though user space processes are free to install routes with the same prefix which will be preferred over these. The reason `RTT_PERM` routes exist is that we now do virtually all address recognition by forwarding lookups and most protocols require that some addresses (or groups of addresses) always be correctly recognized. For IP the permanent route prefixes would include 0/0, 224/4, 0.0.0.0/32, 255.255.255.255/32, 224.0.0.1/32 and, perhaps, 240/4; these routes would be present in a table which had received no other route configuration.

- `RTT_IF` routes are routes which are installed as a direct result of *ifa* configuration. One or two host routes of this type are installed as part of *ifa* configuration on a point-to-point *iff*, while 3 host routes (all-ones subnet address, all-zeroes subnet address and local address) and a shorter destination prefix route may be installed when configuring a subnet *ifa*. These routes will have their *ifl_index* set to that of the corresponding *iff*, will be reliably present when an *ifa* is configured and will be deleted if the *ifa* is deleted. Note that we may end up identifying the routes with local address host routes by a separate, higher preference type, perhaps `RTT_LOCAL`, to distinguish them for certain searches.

- `RTT_IFDEST` routes are routes to neighbors reachable directly through an interface; that is they are installed with a next hop which somehow identifies the L2 address of the neighbor so that the interface-specific code can build an L2 header to reach that neighbor. The route's *ifl_index* identifies the *iff* the routes depend on. These will generally be host routes and may (or may not) depend on *ifa* configuration, but indirectly so. They are instead installed by processes, in the kernel or in user space, which somehow learn of these neighbors. These processes clearly include ARP and ND, but may also include direct static configuration, say for NBMA-configured interfaces or as static ARP entries, which are added and deleted much as normal routes are (except for the L2-addressed next hop). These routes are deleted should they depend on an *ifa* which is deleted, and anything remaining will be deleted when the *iff* identified by the route's *ifl_index* is deleted. These are less preferred then `RTT_IF` routes, but for no particular reason (except that we wish to prefer `RTT_IF` host routes from p2p interfaces over these for forwarding to support some obscure configurations); the unique type mostly identifies them as coming from somewhere other then direct *ifa* configuration.

- `RTT_CLONE` routes are usually installed by some in-kernel process, and are treated specially during changes to the routing table. In particular, some process associated with the route is informed when the “forwarding parent” of the route changes, where the “forwarding parent” is a route with the same prefix as the clone route (if not the clone route itself) or a shorter matching prefix which is being used for forwarding. Several kinds of routes are envisaged to use this. Routes corresponding to multicast groups we have joined will likely be of this type since they need to change their behavior if we become a multicast router, an event which is implemented by installing a new route to 224/4. We may also use this (or a form of this) if we decide to track system-wide MTU discovery by installing host routes to the destination we are interested in into the routing table since these would benefit from being informed when the outgoing interface of their “forwarding parent” changes. There may be other uses.

- `RTT_IFCLONE` routes are like `RTT_CLONE` routes, but track “parent” routes of types `RTT_IF` and `RTT_IFDEST` only. In effect they track the routes that an `SO_DONTROUTE` optioned socket would use. There’s a pretty good chance these won’t be fully implemented, since I’m not sure what they’d be good for, but they exist for symmetry.

- `RTT_CLONE_IGNORE` and `RTT_IFCLONE_IGNORE` routes are like the routes above but are omitted from consideration in forwarding lookups. This form would probably be used for system-wide MTU discovery if this is done.

For preference ordering beyond the type we will likely initially prefer routes with a lower numbered *ifl_index* over higher, and order clones by data type or next hop type or something. I don’t believe that preferences will be all that important, beyond the basic type ordering above, but if that turns out to be wrong this can be revisited.

These route types represent the basic tool box. They allow the table to be maintained as a forwarding table for user space, plus a neighbor lookup to find next hops and for `SO_DONTROUTE` packets (via `RTT_IF`/`RTT_IFDEST` routes), plus a few special purpose operations using clone routes.

Next Hop Types, Functions and Contents

Next hops are the objects we do a route lookup to find. When forwarding packets our interest in the next hop is mostly limited to the output function we will call to deliver the packet, but other processing can require other functions or information which we need the next hop to provide. For example, when installing a route with a next hop through a unicast neighbor specified by an <*ifl_index*><protocol address> tuple we will process the tuple by using it to do a route lookup to find an interface route (i.e. type `RTT_IF` or `RTT_IFDEST`) for the correct interface which matches <protocol address>. The next hop we use for the route we are installing, then, will generally be a copy of the next hop pointer in the route we found in that lookup, though if using that next hop directly would be inappropriate we would expect that the next hop we found would tell us this by instead providing a function we could call which would return the appropriate next hop to use. In either case we end up with a next hop pointer to use for the route we are installing, and in ether case the output function is not interesting. Additional examples of information we might wish to find include determining the “best” local address to use for locally originated packets whose destination we know, which might be equivalent to knowing the *ifa* a next hop somehow depends on, or for making an initial estimate of an MTU to use for packets to a known destination, which might require knowing the *iff* the packet will leave the system through. We hence require next hops to provide us with this information when it is known, while recognizing that not all types of next hops would be able to provide this so we'll need to have fallback procedures to determine default values to use when better information is unavailable.

The next hop structure itself is owned by code somewhere in the system. The visible next hop structure may hence be the public face of a larger structure having function-specific next hop information required to process the packet on. For example, a unicast next hop may hold a copy of the full L2 header required to be prepended to the packet before it is queued to the output *ifd*'s transmission queue, while a multicast group next hop may maintain a list of *ifl_index* values for the interfaces this group has been joined on so that packets addressed to the group arriving on other interfaces may be discarded. The next hop output function would be written to use the information associated with the next hop to perform the processing which needs to be done. This implies that the arguments to the next hop output function will need to include not only a pointer to the packet to be processed and (perhaps) an indication of whether the packet was originated by the host or received from an interface, but also a pointer to the next hop which resulted in the output function being called.

Note that a considerable amount of processing has been deferred to the next hop output function, as it is this function which encapsulates knowledge of what needs to happen to any particular packet. Packets which need to be received by the local system will be given to a function which searches for a socket which matches the packet, perhaps after a filter run on packets destined to the local system is applied. Packets which are destined to leave the machine via an interface may need to be forwarded (e.g. have their TTL inspected and decremented if the system is acting as a router, or just dropped if the machine is a non-forwarding host) if they arrived on another interface, and may require a filter to be applied to the packet if the outgoing interface has one. The particular processing which is done will depend on where the packet is going, which the next hop and its output function will know, and where the packet came from (i.e. was it originated by a process on the system or received from an interface?), with the latter perhaps influencing the former. Note also that the processing of packets originated by local processes may be influenced by metadata associated with the packet. For example, if an application sends a packet to the 255.255.255.255 broadcast address or to a multicast group address the next hop output function may modify its action based on metadata indicating the interface the packet should be transmitted from (indicated by *ifl_index*), with a "default" interface being chosen when no packet metadata is available. This suggests that the next hop output function may also need to receive optional packet metadata, should that exist, as it may influence the function's operation.

Next hops are distinguished by the network protocol which they handle packets for (which would normally be deduced from the protocol of the routes which refer to them), by their type, which will describe the general function that a next hop's output routine will perform, and by next hop type-specific data describing the particular instance of that type of next hop. Whether next hop data is required to describe a particular next hop and, if so, what form that data may take, depends on both the next hop type and (sometimes) the type of route which is referring to the next hop. For an example of the latter, unicast next hops which are specified with `RTT_USER` routes are likely to have specification data consisting of an <*ifl_index*><protocol address> tuple, which provides an indirection to find the actual next hop via a route lookup, but `RTT_IFDEST` routes are likely to be required to specify their unicast next hops with an L2 address appropriate for the *iff* they refer to. Furthermore, we sometimes allow the type of a next hop to change during operation if this is required to achieve some necessary end; see, in particular, the `RNH_HOLD` and `RNH_DEAD` next hop types below. In any case, when a route is installed at the request of an application the next hop for the route will be specified by the next hop type and any data specific to that next hop type which is required to fully identify the particular instance of the next hop. Since next hop types provide one of the primary ways (another might be pseudo interfaces) we have to specify packet processing procedures which follow a route lookup (and perhaps a filter lookup, though we will not be specifying filter functionality here) one might expect that the number of next hop types might grow as the system acquires new functionality.

Given this, the initial types of next hops which we might expect a minimally functional system to implement are described following, along with the functions we expect them to provide and the data which might be associated with them:

- `RNH_REJECT` next hops cause packets to be discarded, with an error perhaps being reported to the sender of the packet. For IP, if the packet was originated by a process on the system the error might take the form of an error return code from the system call attempting to send the packet, while if the packet was received from an interface the error might be in the form of an ICMP Unreachable packet sent back to the sender (if the system is otherwise configured in a way which makes this appropriate; a non-forwarding host might not do this). Note that the `RTT_PERM` 0/0 route would likely have an `RNH_REJECT` next hop.
- `RNH_DISCARD` next hops cause packets to be silently discarded, and is a quieter variant of the `RNH_REJECT` next hop. No error is reported if the packet was received from an interface, though one might be reported if the packet was locally originated. The IP `RTT_PERM` 240/4 route (if we install one) might have an `RNH_DISCARD` next hop.
- `RNH_RECV` next hops cause packets originated by the host to be discarded (perhaps with an error return) but cause packets received via an interface to be received by the system, that is processed to see if any applications are interested in packets like that. Next hops of this type are used for routes to addresses which we will accept packets for if some other host sends them, but will decline to allow local applications to send to. The 0.0.0.0/32 `RTT_PERM` route is likely to have an `RTT_RECV` next hop, as would `RTT_IF` routes to all-zeroes subnet host addresses.
- `RNH_MDISCARD` next hops cause packets originated by the host to be sent out an interface (determined by packet metadata or by some default selection procedures) as a multicast, while discarding packets received via an interface. This next hop type is necessary to implement the basic operational model of IP multicast (you can send multicast packets any time you want to any group you want, but need to make additional arrangements if you want to receive multicasts addressed to a group). The `RTT_PERM` route to 224/4 will likely have an `RNH_MDISCARD` next hop.
- `RNH_MRECV` next hops cause packets originated by the host to be multicast out an interface determined in a fashion identical to `RNH_MDISCARD` next hops, but cause packets received via an interface to be received by the system, as an `RNH_RECV` next hop would do. `RNH_MRECV` next hops are used as the next hop for routes to multicast group addresses which the system is required (or is desired to) locally receive independent of the interface they arrive on. The 224.0.0.1/32 `RTT_PERM` route will have an `RNH_MRECV` next hop.

- RNH_BRECV next hops cause packets originated by the host to be broadcast out an interface, while causing packets received via an interface to be received by the system. RNH_BRECV next hops are close analogs to RNH_MRECV next hops except that they exist to deal with routes to broadcast, rather than multicast, addresses. The RTT_PERM route to 255.255.255.255/32 would have an RNH_BRECV next hop (with the outgoing interface for packets originated by the system being determined in the same way that it is used for RNH_MRECV and RNH_MDISCARD next hops), as would host routes to all-one's subnet addresses (though for these the outgoing interface is implicitly specified by the route which points to the RNH_BRECV next hop).

- RNH_LOCAL next hops cause packets to be received by the local system, that is (as in all cases where the packet is received locally) they send the packet on to perform an additional lookup to determine if there is a local process interested in the packet. RNH_LOCAL next hops are the next hop types used for routes to host addresses which the system considers local addresses, i.e. those addresses appearing as local addresses in *ifa* configuration.

- RNH_UNICAST next hops result in packets being sent out interfaces as unicasts to individual neighbors (with the particular neighbor being addressed by the L2 header of the transmitted packets on multiaccess interfaces). If the packet is originated by the system it is sent to that neighbor unchanged, while if the packet has been received by the system via an interface it is first forwarded (which may result in the packet being discarded). RNH_UNICAST next hops can be referred to by many RTT_USER routes, but will sometimes have a particular interface route (of type RTT_IF or RTT_IFDEST) which each next hop depends on (though they may not; RNH_UNICAST next hops for unnumbered point-to-point interfaces will have no corresponding interface route). Note that RNH_HOLD next hops are often RNH_UNICAST next hops for which L2 address resolution has not yet completed, while RNH_DEAD next hops are often next hops which were once RNH_UNICAST next hops which have had the interface address configuration they depended on deleted.

- Each RNH_RESOLV next hop is associated with a process which is responsible for managing the ultimate forwarding of packets which match the route pointing at the RNH_RESOLV next hop. That is, when a packet is output through an RNH_RESOLV next hop the packet is stored and a prefix of the packet key (usually the destination address alone or the full <destination address><source address> tuple) is sent to the associated process for resolution. The result of this operation is expected to be that the process will determine and install a route which matches the key but which is more specific (i.e. has a longer prefix) than the route with the corresponding RNH_RESOLV next hop, at which point the packets stored awaiting resolution which match the more specific route will be forwarded through its next hop. RNH_RESOLV next hops are used in at least two contexts (their operation is generic, but solves problems in two contexts that I know of). The first of these is as the next hop for the subnet prefix of an interface route on interfaces which require some type of address resolution procedures to be executed to find routes to neighbors not previously known, with the typical cases being ARP or ND. That is, when a packet is destined to a previous unknown neighbor it will match the subnet route and, when output through the RNH_RESOLV next hop, will cause ARP to be informed of the packet's destination address. ARP, in turn, is expected to install a route to this host whose next hop is an RNH_UNICAST next hop which reaches the neighbor host (or perhaps an RNH_HOLD next hop to serve as a placeholder until the address resolution is completed). Once ARP installs this route the packet(s) stored awaiting resolution are output through the new route's next hop. The other context a resolving next hop would be used in is when the system becomes a multicast router, an event which would be signaled (after adding a route to 224.0.0/24 with an `RNH_MDISCARD` next hop) by the installation of a route to 224/4 with an `RNH_RESOLV` next hop associated with the process responsible for implementing multicast routing protocols. In this case, when a multicast with a previously unseen <destination group><source address> tuple is routed, a notification would be sent to the multicast routing process which would be responsible for installing a route (generally with a key which includes the full destination address and a prefix of the source address) to forward the packet. Note that in either of these contexts the process receiving `RNH_RESOLV` notifications may either be in the kernel (i.e. likely ARP or ND for "normal" interfaces like Ethernet) or may be a socket opened by a user space process for this purpose (as would be highly likely for a multicast routing protocol implementation, and might be done for odd-ball address resolution procedures like those defined for ATM networks). `RNH_RESOLV` next hops hence provide a generic method to deal with situations where some subset of the kernel forwarding table needs to be treated as a route cache which is populated by some protocol and/or set of procedures based on the destinations in traffic being sent through the table.

- `RNH_MGROUP` next hops are pointed at by routes to multicast groups (i.e. /32 routes for IP, /128 routes for IP6) which are installed as a result of multicast group joins on interfaces by processes with open sockets. An `RNH_MGROUP` next hop has associated with it a list of interfaces (i.e. *ifl_index* values) which indicate the interfaces this group has been joined on by various listening sockets. Normally packets which are originated by the system and sent via an `RNH_MGROUP` next hop are sent out interfaces in the same way that they would be for `RNH_MDISCARD` or `RNH_MRECV` next hops. Packets which arrived via an interface, however, are received by the local system if the interface they arrived on is one in the `RNH_MGROUP` next hop's list, or discarded otherwise; in some sense the normal behavior of `RNH_MGROUP` next hops behave like one or the other of an `RNH_MDISCARD` or `RNH_MRECV` next hop depending on the *ifl_index* values in its associated receive list. I say "normally", however, since this is the behavior of an `RNH_MGROUP` next hop only when the system is acting as a multicast host. Should the system transition to being a multicast router (which would be detected by the change of the next hop of the forwarding parent of the route pointing to the `RNH_MGROUP` next hop from an `RNH_MDISCARD` to an `RNH_RESOLV` next hop; the route pointing at an `RNH_MGROUP` next hop will likely be an `RTT_CLONE` route so it is informed of this and can modify the behavior of its `RNH_MGROUP` next hop appropriately) both originated and received packets will instead by sent through the next hop of the forwarding parent route.

- `RNH_MULTICAST` next hops have an associated variable length list of interfaces, in the form of an *ifl_index* list, which indicates the interfaces that copies of a packet output through the next hop should be sent out of as multicasts. The interface list will have one or more entries, and may include an *ifl_index* of zero if a copy of the packet is to be delivered to (a) process(es) on the local system. Packets that arrived at the system via an interface will be forwarded before they are copied to the output interfaces (except for the copy delivered to the local system, if any, which will be in the un-forwarded state, without the TTL decrement), while packets originated by the system will be sent out interfaces in their original state. The routes which have `RNH_MULTICAST` next hops will almost certainly be multicast forwarding routes installed while the system was operating as a multicast router. A single `RNH_MULTICAST` next hop may be pointed to by multiple routes requiring the same outgoing interface distribution of matching multicast packets.

Note that multicast forwarding routes will almost certainly have the `RTF_IFCHECK` bit set with the *ifl_index* in the route indicating the interface which a multicast must arrive on if it is to be forwarded.

- `RNH_HOLD` next hops process packets output through them by simply hanging on to the packet (it need not keep more than one, or some limited number) until some change of state of the next hop allows those packets to be sent on. Typically an `RNH_HOLD` next hop will come to exist as a placeholder for an `RNH_UNICAST` next hop whose L2 address has not yet been determined by address resolution procedures. As such, a next hop may exist as an `RNH_HOLD` next hop only temporarily; at some point it would be expect that the next hop's type will be changed to `RNH_UNICAST` and its behavior to change correspondingly. In fact I would be tempted to state that `RNH_HOLD` is only used for next hops which are expected to become `RNH_UNICAST`, though if there turns out to be another process which makes `RNH_HOLD` next hop behavior useful I would expect the same type to be used for that as well.
- `RNH_DEAD` next hops come to exist when the address or interface configuration on which a next hop (generally `RNH_UNICAST` or `RNH_LOCAL`) depended on has been removed but the next hops themselves are still referenced by other (particularly `RTT_USER`) routes. That is, `RNH_DEAD` next hops were once some other type of next hop which became invalid when some independent configuration changed, but could not be deleted because of dangling references by other routes. `RNH_DEAD` next hops behave like `RNH_DISCARD` next hops with respect to packets processed through them, and will be eliminated from the system once all routes which reference them have been changed to no longer do so. Note that the primary originator of routes holding references to `RNH_DEAD` next hops will be user processes (hence the `RTT_USER` route type) since kernel processes are expected to clean up after themselves with respect to routes they maintain but it is up to user space processes to do the same for routes they have installed.

Note that this list is unlikely to be exhaustive, and I'd expect the occasional invention of new next hop types and functions to perhaps be associated with the implementation of new forwarding functionality. I believe these types are sufficient to produce a basically functional IP/IP6 host and router implementation for both unicast and multicast applications.

It is probably useful to point out again why I'm fond of the integration of multicast route operations in the same table structures as all other operations, and as an integrated part of the standard operation of the forwarding path; the primitive operations required to support multicast well are very often nearly identical to the primitive operations required to support other applications well. The decisions and procedures required for a host to send multicast-addressed packets out an interface (which requires somehow specifying the output interface since the destination address gives no hint) are the same as required to implement non-multicast broadcasts addressed to 255.255.255.255. The fact that multicast code in traditional kernels has been treated as a bag-on-the-side special for multicasts, however, has meant that those kernels have never provided decent support for sending packets to the latter broadcast address even though it is the same problem and even though there are widely used applications, like DHCP, which must address packets to 255.255.255.255. The fact that DHCP clients often use BPF to send the packets they need to is a (very typical) hint that this is broken for 255.255.255.255, even though the bag-on-the-side multicast code knows what to do. And note that processing received multicasts on a host is also very similar to the all-one's broadcast except that while the all-one’s broadcast has only one option (receive the packet locally to see if anyone wants it) multicasts have three: `RNH_MDISCARD` (don’t receive it, discard it), `RNH_MRECV` (receive it) and `RNH_MGROUP` (receive it if it came from an interface in the list, discard otherwise).

Another mechanism which finds use for multicasting and elsewhere is the `RNH_RESOLV` next hop which is generically useful for segments of the route table which need to be maintained as a cache, and hence provides a common mechanism which is equally useful for ARP and Neighbor Discovery and for multicast routing. The very close commonality of the primitive mechanisms each of these require demands that they be implemented the same way (which is also why I like maintaining the ARP cache in the routing table as first class routes instead of trying to hide them in interface-specific code, leaving other instances of the same operation to be done separately) and, while there is an argument that the mechanisms aren’t really common, since ARP is often done as a kernel process while multicast routing never is, it is the case that sometimes ARP really should be implemented in user space as well (e.g. for the portion of the subnet address space being managed by a DHCP server) and the model for user space multicast route resolution provides identically the support needed for user-space ARP too. And, while speaking of acting as a multicast router, I find that distinguishing between one’s state as a multicast host and multicast router based solely on the contents of the routing table (multicast routing is enabled by installing a new 224/4 route) to be a really clean and comfortable way to do this. This does require routes added to reflect multicast group join state to be `RTT_CLONE` routes to track this, but the route clone mechanism may itself be a useful primitive to have for other operations as well. To summarize this I would simply repeat my belief that whether you are interested in multicasting or not, of all the ways there might be to provide a mechanism to support particular operations, the choices that are “best” are those which make multicasting look like a normal, integrated part of the networking implementation. It works well.

It is also worth while mentioning an additional function required of `RNH_RESOLV` next hops on interface subnet routes. In particular, ARP (Neighbor Discovery as well, though the latter is a bit more complex) has three ways to discover neighbors which it maintains in its route table cache for the interface. The first is by the ARP protocol itself: when a neighbor on the interface sends an ARP request looking for a local address on the receiving ARP’s system it is probably a good idea for ARP, in addition to responding to the request, to also install an `RTT_IFDEST` host route back to that neighbor based on the neighbor’s information in the request since it is likely we’ll need to send packets back to that neighbor in short order. The second way ARP might learn of a neighbor is if someone (either a local process, or packets received and being forwarded as a router) sends a packet addressed to that neighbor. If ARP was not previously aware of that neighbor the packet will be forwarded through the `RNH_RESOLV` next hop of the subnet prefix, which will result in ARP being informed of the need to install a route to that neighbor and send requests to determine its L2 address; this is the mechanism in common with IP multicasting. The third way ARP can receive knowledge of a neighbor it does not know about, however, is if some other process in the system attempts to install a unicast route with that neighbor’s address as the next hop. In this case the process doing the route installation will do a lookup for the next hop address among interface (i.e. `RTT_IF` or `RTT_IFDEST` type) routes, expecting to find a next hop of type `RNH_UNICAST` or `RNH_HOLD` to point the route at, but will instead find the subnet prefix route pointing at the `RNH_RESOLV` next hop. In this case pointing the newly-installed route at this next hop would be improper; it needs a next hop for that particular neighbor, which we don’t have yet. To deal with this the `RNH_RESOLV` next hop will have an additional function handle which, when called, will return the next hop the route being installed should point to, which will likely be an `RNH_HOLD` next hop which ARP will change to `RNH_UNICAST` when it finds the information it needs to do that. That is, the function implements a “next hop resolution” operation which creates the next hop (and perhaps installs the host route to go with it?), informing ARP that it needs to fill the next hop in. The entry point handling this third case completes the generic support for interfaces requiring address resolution.

Processing “Locally Received” Packets

The action of many of the next hops described above, which are the result of route lookups, is to “receive” the packet locally, i.e. to determine if some local process is interested in the packet and, if so, to deliver the packet to the process. Here we look at what that implies, and what data structures and procedures will be necessary to implement this operation.

The broad outlines of what is implied here are clear. The first operation which needs to be performed on a locally received packet is to reassemble it, should it be fragmented. Subsequent processing depends on data which is likely to be present in only one fragment of a packet, so ensuring that all fragments of a packet are stitched together before proceeding is necessary to ensure that all fragments of a packet can be delivered to the process which wants them (note that an interesting question arises about whether a filter to be applied to locally destined packets [say, one configured on the loopback interface *iff*] should be applied before or after reassembly, but filtering issues are out of scope for the purposes of this paper). There are implementation issues concerning exactly how reassembly might be best implemented. In particular, the shared nature of the data structure used to manage incompletely reassembled packets suggests that this work is best (as in, most straight forwardly) done by queuing fragments to a single thread to do the processing, but this may become a bottle neck if reassembly is commonly required; this topic is best addressed with context gained from implementation experience.

This still leaves a correctness issue, however, in particular that in the presence of multiple routing domains reassembly, which requires that a specific address in a packet always identifies a single, specific host, must be done in a way which is sensitive to the routing domain the packet is associated with; in effect, we must implement a reassembly queue per routing domain (or, equivalently, include a routing domain identifier as part of the key, along with the packet addresses and packet ID, which identifies an individual packet). This requirement in turn raises the question of how a packet’s routing domain is determined which, given no reason to prefer something else, we answer by taking the packet’s routing domain to be the routing domain of the table that the lookup was done in to determine the packet was one of local interest. We assume, then, that we will need an instance of the packet reassembly machinery for each routing domain for this L3 protocol, with each instance being identified by the route table it is associated with.

Once a packet is successfully reassembled, or if it requires no reassembly, the next job is to determine whether a protocol session exists for which this packet is destined. This has traditionally been done for IP by using the protocol ID in the IP header to find the appropriate protocol input function which in turn, in the very common cases of TCP and UDP, would extract the addresses and port numbers from the packet and use those to determine the presence of protocol-dependent state for the particular session.

It is useful to note, however, that the vast majority of packets find their protocol session state with reference to the following packet fields, previously identified as the conceptual "key" for route lookups as well:

<destination address><source address><IP protocol><first 4 bytes of transport header>

Our design here, then, will avoid demultiplexing to transport protocol dependent code and using a transport-specific lookup to find a protocol session in favor of extracting the above fields in transport protocol independent code and using them as a key for a data structure lookup which may identify the protocol session directly. Note that the data structure we use for this will have requirements similar to the route lookup data structure; in particular we require that lookups can be performed in the data structure while a modification is concurrently in progress, so that readers of the structure need not acquire a (hotly shared) lock to use the structure to deliver packets. The lookup required in this case is not a strict prefix match, but instead will allow fields above to be fairly arbitrarily wild-carded (only <IP protocol> need be specifically matched) with a well defined strategy for determining the "best" match among several. The presence of the IP protocol in the key permits each transport protocol to continue to manage its protocol demux entries separately (and may permit finer-grained locking for concurrent modifiers of the structure) but they are managed in a single, unified structure to the extent possible rather than in separate, per-protocol structures. Note that the design does not preclude a transport protocol from being implemented which uses other packet data for its protocol session demux; that end could be accomplished by inserting an entry in the table which captures all packets to a particular IP protocol, and applying protocol-specific procedures to packets delivered to it. The intent is rather to limit the need for a lockless multiple-reader, single (or multiple) writer data structure for transport protocol processing to this one structure to the extent possible, to try to simplify the design of subsequent processing by allowing it to be primarily single threaded.

Several issues (at least) are deferred for later consideration. The first of these concerns whether the incoming interface *ifl_index* should be additionally included in the lookup key to allow (when the field is not wild-carded) protocol matches for packets which arrive on particular interfaces only. While this is not a widely useful feature (TCP, for example, needs to process packets for its connection regardless of where they arrived from) it may be required to enforce multicast join semantics in some cases and may find use for applications like interior routing protocols, which are concerned only with packets for the protocol arriving on interfaces they are configured to run on. The second issue concerns routing domains and, in particular, whether the routing domain ID should also be included as part of the lookup key or it should instead be implemented with per-routing-domain protocol demux structures. The difference between these two choices occurs with service applications which wish to accept connections in several routing domains. In the former case these servers could operate with a single listening socket with a wild-carded routing domain (when a connection is established with *accept(2)* the routing domain ID of the incoming connection would be included in the new socket's protocol demux entry) while the latter arrangement would require such servers to open a listening socket per routing domain. We will initially choose the latter implementation because it is simpler, though experience with routing domains may prove the former implementation to be more useful.

We have determined so far that a locally received packet will, if fragmented, be reassembled by a per-routing-domain reassembly facility, then a protocol demux key will be extracted from the (complete) packet and used to look up a protocol session in a per-routing-domain structure, to which transport protocols add entries to classify and separate packets which somehow require related processing but can be processed independently of others. It is intended that the result of this lookup include a procedure handle to call to further process the packet and some opaque (to the caller) state for the called procedure to use to determine what should be done to the packet. The remaining issue is, what happens inside those processing functions?

Note that we have been (or should have been) somewhat vague about what entries in the protocol demux table represent since, in fact, the answer to that depends on what the implementor's application needs it for. The protocol demux lookup might find use, for example, for identifying encapsulated packets in tunnels terminating at the local machine, in which case the processing required, perhaps stripping the outer packet header and processing the inner packet as if it just arrived on a (pseudo-) interface. A slightly heavier weight variant of this might have the entries corresponding to NAT cache state, which requires the packet be subject to a somewhat heavier weight transformation before being delivered onward. In fact, however, the most likely "normal" case is that the entry will directly correspond to an open socket in user space, along with the transport protocol state required to process, and to be updated by the processing of, that packet from its raw state into the input socket buffer.

To get to the point, then, what we expect is that generally the end result of protocol demux processing will be that the packet will be inserted into a queue to be processed further by a thread of execution separate from the thread which enqueued the packet. While a more detailed discussion of the reasons for this is a topic of the next section, it is worth summarizing the argument here. On a multiprocessor system, several processors may be concurrently busy processing packets. The packets which are associated with, say, a single TCP connection are best processed by a single thread (in particular, a thread running on the same processor core as the ultimate receiver of the data would be optimum due to cache considerations) having exclusive access to both the packets being processed and the bookkeeping state associated with the transport connection. Different packets for this one connection may, however, arrive on different interfaces and be processed through the route lookup and protocol demux by different threads running concurrently on different processor cores, so ensuring that only a single thread does the (stateful) TCP processing in general requires that, once the protocol session has been identified, the packets be inserted by the receiving thread(s) into a queue for this connection and a single, separate thread be started (if it isn't running already) to consume the packets from the queue, execute the transport protocol processing and transfer the data to the input socket buffer. In essence we split the task of delivering the data from the interfaces to the socket into two stages, with a set of threads doing the processing from the interface input packet queues either to interface output packet queues (for packets being forwarded) or to transport session packet queues (for locally received packets), with a separate set of threads processing data from transport session packet queues having queued packets to the sockets which applications will receive them from.

Stepping up a level for a better view of the bigger picture, the thread of execution which accesses both the route table and the protocol demux table makes its decisions statelessly, in that it uses these data structures to make decisions, but does not modify them; the only modifications it makes are to the packet it is processing. The goal of this processing is to find a queue to append the packet to, a goal which can be achieve without locking or other blocking operations and so will run continuously until it completes all tasks that can be performed statelessly. Sometimes the queue it finds will be an interface output queue which will be serviced by the device driver that will transmit the packet out a hardware interface, but sometimes the queue will be one which is served by an in-kernel process whose further processing involves modifying kernel state. In the latter case the arrangement of queues serves two purposes. The first is to ensure that packets whose processing will involve modifications to a common bit of state, for example a single TCP connection protocol control block, will be processed through a single queue with a single thread providing the service. The second is to identify processing which can be done independently; the processing of packets which end up in different queues will in general require modifying independent state and hence may be executed in parallel. Thus the result of the intermediate queuing stage is to allow us to find parallelism among the network tasks that need to be done. We expect that some of the threads executing these tasks may block. For example the processing of incoming TCP packets may block waiting for a thread which is sending data through the same connection to complete, or the thread processing ARP packets may wait for the lock providing it with exclusive access to modify a routing table, or any of these threads may block waiting for memory to be allocated. Because the previous processing has identified tasks which can be processed independently (i.e. there are other queues of packets awaiting processing) a suitable kernel scheduling discipline may both deploy the resources of multiple processor cores for this work and may respond to a thread that has freed a CPU because its progress is blocked by initiating another thread to service another queue's packets.

A later section will explore the work scheduling issues this arrangement raises, and present at least a preliminary proposal for an approach to deal with them.

Layer 2 Interface Identification and Protocol Demultiplexing

There is a remaining packet classification problem that will be addressed in a fashion analogous to the route lookup and locally received packet processing described above, this being the processing of an incoming packets from an interface to determine, from information in an incoming packet's L2 header, if that packet is interesting to the system and, if so, what the packet's incoming *ifl_index* should be set to and which network protocol's input routine should be called to process the decapsulated packet. It is clear that this problem is one which is closely coupled to interface configuration. In particular the event which changes the packets the interface code needs to accept and process is the addition (or deletion) of an *iffamily* structure to (from) a branch of the interface configuration tree rooted at the *ifdevice* structure which corresponds to the hardware interface the packets are arriving on.

The way this functionality is implemented in current kernels is problematic for us. In its most basic form this is implemented with code which understands the format of the L2 header for the encapsulation configured which extracts an L2 protocol identifier from that header and, for each protocol it understands, looks at interface configuration to see if the state which would require packets of that type to be processed by the system (in our case that would probably be an *iff* for the corresponding protocol) has been configured on the interface. If so it calls the appropriate protocol input routine to process that packet after discarding the L2 header, but if not it drops the packet as "not for us." This creates several problems. First, since the mapping between protocol identifier, interface configuration and input routine is performed in code, the function which does this turns into a huge `case` statement, complete with conditionally compilation `#ifdef`'s, with knowledge of every protocol identifier that the implementation might ever receive packets for. Support for an additional protocol ID requires adding new code to this function even if no code to consume such packets exists in the kernel and the only goal of receiving such packets is to deliver them to the socket of a user space application (of course applications like this generally make use of BPF instead to avoid this, a hack-around which is a reliable indication of an unfortunate underlying design). What is worse in our case, however, is the apparent need, once the packet protocol is identified, to traverse the interface configuration from the *ifd* at the root (which the driver might know) to find the *iff* whose presence would tell it that a packet of that protocol (with those VLAN tags, should it have them) is interesting to the system. The need to do this in an interrupt context implies constraints on how those interface structures can be maintained and modified if we are to avoid expensive and highly contended locking, a constraint which may make the management of an already quite complicated structure untenable. It can't work this way.

What we will do instead is to replace the coded lookup currently implemented for protocol demultiplexing with a purely data driven procedure which takes a key containing the fields from the L2 header which are used identify both the *ifl* and the protocol and uses this to perform a lookup in a data structure that returns a result which includes an *ifl_index* to use as the packet's incoming *ifl_index*, a procedure handle to call to process the packet with an opaque pointer to include as an argument, and perhaps an indication of whether the L2 header should be removed from the packet or not before calling. If no match is found the packet is not for us; otherwise the packet is prepared (the *ifl_index* is copied to the packet mbuf and the L2 header is removed if required) and then the procedure handle is called to process it on. The data structure used for this lookup will have the same characteristics as the others documented in this section, that is that it will provide consistent lookup results while a (single) modification is being performed concurrently, without locking. This data structure will start off empty (i.e. no packets are for us) and will be populated with entries to do the right thing as configuration and other operations require it. The code which demultiplexes based on protocol identifier now will be replaced with code which constructs a key based on the L2 header contents and does the lookup. If the lookup fails the packet is discarded, otherwise it is processed as dictated by the matching entry.

A simple example might clarify the above. Suppose the driver is an Ethernet driver and this Ethernet has a particularly simple configuration: IPv4 without VLANs. The lookup key extracted from each packet will generally consist of the 2-byte ethertype. Prior to any interface configuration the lookup structure will be empty, so when a packet is received from the hardware its ethertype will be extracted and used as a key for the lookup which will return nothing. All packets will be discarded. When it is desired to use the interface for IPv4 it will be configured by adding an *ifl* (with no L2_ID, since we aren't doing VLANs), an *iff* and, ultimately, an *ifa* with subnet address configuration (perhaps DHCP might do the latter). At the point where the interface has enough configuration to begin receiving packets, which is probably when the IPv4 *iff* is added, the thread doing the interface configuration will arrange for packets from the interface to begin to be delivered by adding the appropriate search entries to the lookup structure, both with the *ifl_index* of the *iff* being added. For IP two of them are required. The first will have a search key of 0x0800 (IPv4), with a procedure handle which essentially is `ip_input()` with an opaque pointer pointing at the *iff*, information from which is needed to determine how to process the packet (since the search entry is dependent on the *iff*, and will go away when the *iff* does, it is appropriate to point at it directly). The second, whose installation might be deferred until *ifa* configuration is added, will have a search key of 0x0806 (ARP) with a procedure handle to a function which might re-queue the packet to be processed by the kernel's ARP implementation. In effect, rather than having packet processing code look at interface configuration to determine which packets to accept, we have interface configuration code explicitly install search entries specifying which packets to process and how to process them as a side effect of the interface configuration which requires that. Note, however, that with this approach the packet processing code itself requires no inbuilt knowledge of the meaning of ethertypes 0x0800 and 0x0806; it is sufficient for that code handling interface configuration knows what those fields should be in packets interesting to it.

The design problem for L2 protocol demultiplexing code, then, is to extract data from the packet which can be used to construct a key containing all the information from the L2 header which is required to identify not only the packet's protocol but also the *ifl* the packet is associated with. For Ethernet, while the code constructing the key is not required to have knowledge of ethertypes for protocols, it will need to recognize the ethertypes for VLANs and for PPPoE since these indicate additional data which need to be included in the key. For example, a key constructed for a VLAN-tagged PPPoE packet (to extend examples given earlier in this paper) could conceivably have a 18 byte value of

    0x81000xxx88641100yyyyaaaaaaaaaaaapppp

where "xxx" is the 12 bit VLAN ID, "aaaaaaaaaaaa" is the source MAC address, "yyyy" is the PPPoE session ID and "pppp" is the PPP protocol type, with these values being filled in from data in the packet. If interface configuration has been built to match this PPPoE session and understands the formatting of keys build by the packet handling code it can install a lookup structure entry when an *iff* for the protocol identified by "pppp" is installed based on the protocol of the *iff* plus the L2_ID information in each of the *ifl*'s between that *iff* and the device's *ifd* which will match this. If the interface is configured with a bridging *iff*, on the other hand, a single entry which matches all packets (i.e. essentially an entry with a zero-length key, like a default route) can be installed instead. A relatively simple procedure for extracting an L2 key in the packet processing code called by the device driver to deal with incoming packets, plus a data structure populated by interface configuration to use to look up the key, can hence be leveraged to deal with arbitrarily complex interface configurations, directly providing the device's input code path with the information it needs about what has (and hasn't) been configured while avoiding any need for that code to access the interface configuration tree to determine this.

The data structure which implements this lookup will have many attributes in common with the route table data structure described above, though we might not use an identical structure since we might prefer to optimize this one for the (relatively more common) case where only a few entries exist. The structure must allow lookups to continue currently with an in-progress modification to avoid having to block the packet input thread. The entries in the structure should include keys which are variable length prefixes with a length measured in bits; an entry useful for capturing all PPP control packets, for example, might include only a few high order bits of the PPP protocol type. We need to be able to install an entry with a zero length prefix, that is the equivalent of a default route; this might be used, for example, when an Ethernet interface has been configured with a bridging *iff*. Some (perhaps limited) support for overlapping prefixes, with longest match resolution, might be required, perhaps to allow ethernet frames with a bridging protocol ethertype to be captured on an Ethernet configured for bridging (though this might not be necessary; this requires some additional thought). It seems like we need not allow multiple entries with the same prefix.

We have been associating the lookup structure itself with an *ifdevice*, and entries in the lookup structure with *iffamily* configuration, but this is not necessarily true. Consider, for example, the case where Ethernet frames are encapsulated in PPP, or are being received over an ATM VPI/VCI, a configuration which we might represent by configuring an Ethernet *ifl* on the PPP or ATM *ifd* with the remaining Ethernet *ifl*/*iff* configuration being added above the root Ethernet *ifl*. In this case we might prefer to do the L2 protocol demultiplex in two stages, with a lookup structure dealing with PPP or ATM L2 headers and protocol types being configured on the *ifd* while a structure dealing with Ethernet header demultiplexing being associated with the Ethernet *ifl*. An entry would be added to the *ifd* structure to identify those packets carrying Ethernet frames using fields from the native PPP/ATM encapsulation, with the function these packets are delivered to extracting Ethernet keys from the encapsulated Ethernet frame and looking for a result in the lookup structure associated with the Ethernet *ifl*; multilayer encapsulations might hence be dealt with by multiple L2 lookups. In this case it might be the configuration of the Ethernet *ifl* which resulted in the insertion in the *ifd* table of an entry identifying packets associated with the Ethernet *ifl*, with *iff* configuration above the Ethernet *ifl* causing the addition of entries to the Ethernet *ifl* table. And, in fact, it might be that the PPPoE key example given above would also be handled with a two stage lookup, with the "pppp" PPP protocol ID not being included in the Ethernet key. Instead the Ethernet key might include only the information required to identify the PPP(oE) *ifl*, with the PPP protocol ID being demultiplexed in a separate lookup structure associated with the latter (whose maintenance would be identical to that associated with a "normal" PPP interface *ifd*).

There is one other aspect of this approach to L2 demultiplexing which might prove useful, that being to provide support for generic "L2 protocol" sockets. That is, while we've been viewing the contents of these lookup tables as having arisen as a side effect of interface, and in particular *iff*, configuration, this is not the only possibility. A second possibility is to allow applications to open "L2 family" sockets and then bind each to an *ifl* and an L2 protocol identifier appropriate for that *ifl*. For example, the PPPoE discovery protocol (which is used to establish PPPoE protocol sessions, which in turn results in PPPoE interface configuration being added to the kernel) runs directly over the Ethernet, identifying its packets with an ethertype of 0x8863. An application which implements this protocol might arrange to receive these packets by opening an "L2 family" socket and binding it to ethertype 0x8863 on the *ifl* over which PPPoE is running, with the result of that being that an entry for ethertype 0x8863 is added to the L2 lookup structure for the Ethernet the *ifl* is associated with (if the *ifl* is VLAN-tagged the VLAN part of the key for the entry would be deduced from the interface configuration), with packets matching the entry being delivered to the application socket. A user space pppd might operate similarly, opening an L2 family socket and binding it to a PPP *ifl* with a 0x80/1 protocol type (that is, match all PPP protocol ID's with the high order bit set; these are LCP and NCP packets), and with the bind operation resulting in an appropriate entry being installed in the *ifl*'s L2 protocol lookup table. A third example where this could see use might be an implementation of the IP routing protocol IS-IS. IS-IS encapsulates packets directly over the L2 medium it is running on (on Ethernet it uses DSAP/SSAP encapsulation, which SNAP encapsulation is a variant of, so the construction of Ethernet lookup keys would need to accommodate this), and the application itself must know how to do this, so using support for "L2 family" sockets might allow it to establish the binding it needs to deliver packets from each interface it is running on without any need for code in the kernel to explicitly support this protocol. Indeed the latter attribute is an advantage shared by all the example users of "L2 family" sockets over the current kernel; these protocols can be implemented with generic, rather than protocol-specific, support for delivering their packets (though in the case of IS-IS this is especially interesting given the existence of a whole `netiso/` directory full of source code whose only remaining purpose seems to be to provide support for IS-IS). Removing explicit knowledge of protocol ID's from the device code in favor of a programmed lookup means that protocols whose implementation would have once required adding code to the kernel (or, maybe worse, using BPF as a hack-around) can perhaps now be done with a generic facility.

I would finally point out a rough, but interesting, analog between the the L2 demultiplexing facility described here and the L3 protocol demultiplexing facility described in the previous section concerning the processing of “locally received” packets. Both structures sometimes have entries inserted as a side effect of interface configuration (in the L3 case these would be encapsulation interfaces), and both structures sometimes have entries inserted to implement a socket binding. In some sense both of these structures achieve very similar purposes differing mostly in the data they examine to do so.

Packet Processing, Kernel Scheduling, SMP Issues

So far the discussion has concentrated on the representation of shared networking state in the kernel. That is, we’ve looked at interface representation and configuration structures, at the contents of the routing table and of next hops, which are the result of a route lookup, at the L3 protocol demultiplexing structure used to identify the receivers of local-bound packets (as determined by the L3 route lookup) and at the L2 protocol demultiplexing structure used to determine how to process packets arriving from an interface. The intent of this section is to begin to look at how to use these facilities to move packets from the place where they arrive at the kernel to the place they need to go, at the additional programming structures required to do this, and at how this will need to be arranged to deploy the full resources of an SMP processor complex to work on achieving this when doing so is appropriate.

We’ve already introduced the major change in kernel processing that this work imagines. In the current kernel a packet received on an interface and bound for a user socket is inserted into an input queue by the device hardware interrupt routine. That queue is subsequently serviced by a software interrupt thread which processes each packet in that queue through the appropriate network and transport protocol processing until the data ends up in an application’s socket queue. The change proposed here is to add yet another queuing stage between the device and the application, which we’ll collectively refer to as *service queues* for lack of a better term, that typically sits very roughly at the processing boundary between the network and the transport protocols. In this arrangement we have one thread doing the work to process the packet from the device input queue to an appropriate service queue with a second, separate thread of execution being deployed to process packets from each service queue to the socket. This additional queuing operation is admittedly a significant expense (and is proposed with full knowledge of another proposal by an eminent network processing designer arguing instead for the elimination of the existing queuing stages) which needs to be justified by the benefits it provides. Doing so will be a consideration of this and subsequent sections.

The arrangement of processing being proposed here is subject to and organized around a correctness constraint, that being that we must arrange our packet processing in a way which avoids reordering packets which are part of an individual “flow”. While maintaining ordering is not strictly necessary for many IP protocols doing so can improve the performance of TCP and avoid testing the corner cases in other protocols and applications, and in fact for some IP protocols (e.g. L2TP) and non-IP protocols (e.g. bridging) maintaining packet order is a requirement and failing to do so would represent a poor quality implementation. Maintaining this packet ordering constraint in general requires that all packets which may be part of a single flow be processed by a single thread of execution which operates on the packets in arrival order (as maintained by a FIFO queue). This particularly constrains the input packet path from an interface to an application; all the packets arriving from a single interface might belong to the same flow, and there is no fully general way to determine that they don't until we have done the work to classify those packets through the routing and protocol demultiplexing stages described above. To ensure ordering, then, requires that all packets arriving on a single hardware interface and queued in that interface's input queue be processed through a single thread running on a single processor (for cache efficiency the device interrupt routine which receives and inserts packets in the device's input queue and the thread which reads them out might preferably run on the same processor, though that's a scheduling detail we won't worry about now). What this means is that, while on a multihomed host or router with several hardware interfaces we can perhaps utilize an SMP system to process packets arriving on different interfaces in parallel using different processor cores, in the common case of a single homed host all traffic arriving will be constrained to be (initially) processed through a single thread running on a single processor resource. It being the case that this initial input processing must be performed using what may be a small fraction of the computing resources available within the processor complex, avoiding having this single thread become a bottleneck would seem to require that we minimize the single-threaded packet processing to the extent possible, which implies that it should continue only to the point where it has done sufficient classification work to identify the "flow" ("flow" has acquired quotes at this point since we don't define it as it might be for IP at this point; the packets belonging to the same "flow" for the purposes of service queuing are those whose processing requires modifying a common bit of shared state). At this point it should insert the packet in the service queue for the "flow" and get on with processing the next input packet from the interface. The additional queuing stage is hence being added to minimize the work done at a processing stage which must inherently be single-threaded, but we've tried to place this queuing stage at a point in the processing where the "flow"s have been identified with sufficiently fine granularity (i.e. there are potentially lots of queues for the packets to be added to) to allow subsequent processing to be done broadly in parallel with as much processing resource as we are willing to spend on it. While there may be other reasons to like those queues (which will be noted when the topic arises) this is the primary reason for their existence. It might also be interesting to note that the potential small-straw problem occurs only on the receiving side of the packet path, that is in the interface→application direction; in the application→interface direction most sending side operations are already inherently "flow" oriented and fully parallel, and hence provide no reason not to simply run them to completion.

To make subsequent discussion clearer it is probably work while to (badly) define names for the software contexts that networking-related code can be executed in. In particular, four such contexts can be identified:

- *Hardware interrupt context* refers to code running in a hardware interface's interrupt service routine, either for sending or receiving packets. Clearly such code cannot block (other than, perhaps, the necessary brief waits for spin locks) nor sleep, and beyond any mbuf allocation which might need to be done should avoid executing system code to the extent possible. Hardware interrupts are high priority, and we try to both avoid blocking them and to minimize the work done in that context (it'll be less than the current kernel).
- *Software interrupt context* is the context where processing of the packets from a hardware device's input queue is performed. It is the "narrow straw" described above, in that the number of software interrupts which can be simultaneously active may be limited to the number of hardware interfaces the system has (and systems having just one are the common cases). Code running in a software interrupt may not sleep, may not block other than briefly waiting for spin locks, may in general modify no state other than the necessary modifications to the packet it is processing (and maybe counting things) and

may not allocate memory other then the mbufs necessary for packet processing. While software interrupts may be interrupted by hardware interrupts we expect that they will otherwise run continuously until they have emptied the queue they are servicing, at which point they will stop. The goal of code running in software interrupt context is to statelessly deliver each packet to a queue (either an interface output queue or a “flow” queue) and then continue on to the next packet.

- *Kernel thread context* is the context which software dequeuing packets from a service queue runs in. Software running in kernel thread context may block. It may allocate memory (beyond just mbufs) and it is free to sleep for memory allocations or waiting for locks. Software running in this context will also be the exclusive kernel thread consumer of packets from the service queue it is processing, and hence may have exclusive access to data associated with that service queue if that data is only updated by software running in a kernel thread.
- *Process context* refers to software which is run as a result of a system call by a user space application. This software may block, may allocate memory and is free to sleep for memory allocations and to acquire locks. There is hence no difference between software running in process context and software running in kernel thread context other than the event which caused the software to run. Process context refers to operations initiated by a user space process, while kernel thread context refers to operations initiated by some event which is asynchronous to the process, like a packet arrival.

It should be apparent that the same software can run in kernel thread and process contexts since these are similarly (un)constrained. It should also be the case that the maximum number of threads required to run simultaneously in software interrupt context should be no greater than the number of processor cores in the system since the tasks which can be performed in that context must avoid blocking and hence will run continuously to completion. The number of kernel threads which might be usefully deployed to execute in kernel thread context, however, is not so clear but is likely to be larger than the number of processor cores given that software running in this context is allowed to sleep; how to optimally schedule kernel threads is a topic for future study.

There is also a priority ordering in the list above. Hardware interrupts are never blocked (except as necessary to implement synchronization primitives) and will preempt any other networking code which is currently running. When there is work to be done in software interrupt context it may preempt kernel threads and code running in process context. When a kernel thread needs to run it should definitely preempt a user space process running application code, though what should happen when the user space process is executing kernel code via a system call deserves some study (how to properly schedule kernel threads is a bit mysterious to me).

The subsequent sections will look at how the networking stack’s functionality will be distributed within this framework, and what other bits of support code might be required to successfully implement this.

Network Hardware Device Drivers

In the current network stack hardware device drivers have a curious asymmetry. For packets being output by the driver the network stack generally queues fully encapsulated frames to the device’s output queue, leaving the driver code (i.e. the code which may run in hardware interrupt context) to take these frames from the queue and arrange for the device to transmit them unchanged. On the receiving side, however, the code running in hardware interrupt context is burdened with additional tasks, in particular demultiplexing packets by protocol and then subsequently decapsulating those packets as is appropriate for the receiving protocol. That is, while the output queue is a per-device queue, the input queues are per-network-protocol queues and the hardware device driver code is given the task of classifying the packet (which in general requires to have have reasonably detailed knowledge of interface configuration) so that it can select the appropriate network protocol's queue to insert the packet into, after which it schedules a software interrupt to service that protocol's queue.

There are several reasons not to like this arrangement. First, it requires a task which can sometimes be complex (note that BPF code needs to run before packet decapsulation, so that ends up being run at interrupt level as well) to run in high priority interrupt code, something which is not aesthetically pleasing and which keeps the driver busy doing something which could be done elsewhere rather than what it must do, which is make incoming packets from the hardware available to the network software. Second, the arrangement of input queues is not very efficient. While there might be dozens of "protocols" implemented in the kernel, and hence dozens of protocol input queues to choose from, it is very common that only one (or occasionally two) of these receive a significant amount of traffic. If there is only, say, a single IPv4 input queue for all input traffic on a host or router with multiple hardware interfaces, and most of the traffic goes into this queue, then we lose the parallelism we might otherwise be able to obtain by processing traffic from different interfaces through different threads; our narrow straw becomes narrower still. We could fix this by having separate per-protocol queues for each interface but this is multiplying the data structures while still having most of it unused most of the time. If most of the traffic is to one or two network protocols, and hence ends up in only one or two queues, doing this early classification in an expensive spot has little value.

Because of this we are redefining the functionality executed at device interrupt level to be simpler, and to be more pleasingly symmetric between input and output, by having both the input and output deal in fully encapsulated frames which are placed in per-device queues. That is, much as the transmit side of the device driver picks packets out of a single per-device output queue (or perhaps a group of them if doing priority queuing; as a first cut we don't worry about this, however) and gives them to the hardware to transmit unchanged, the receive side of the device driver will take fully encapsulated frames the hardware receives and add them, unchanged, to a single per-device input queue where software can read them from, initiating a software interrupt service routine to do that work if it isn't running already. Packet decapsulation and protocol demultiplexing is deferred to software interrupt context, which might operate by taking each packet from the device input queue in turn, calling BPF is necessary, determining the packet's protocol and then calling the appropriate protocol input routine to process it. With an input queue per device a software interrupt context thread per device is possible, and may be run concurrently if the processor resources exist to do so.

By defining device driver code (i.e. that which is executed in a hardware interrupt context) to deal exclusively in raw frames we also remove most of the requirements for that code to understand the encapsulation configured for the device, and hence the need to access interface configuration data structure at hardware interrupt level. That is, for a synchronous serial interface the sole difference between being configured for Cisco HDLC, PPP or Frame Relay encapsulation may be the procedure handle the driver is configured to arrange to execute in the software interrupt to service its input queue. Similarly, an Ethernet interface may have its software interrupt procedure handle modified to reflect whether it has been configured with a bridging *iffamily* or a collection of L3 *iff*'s. An encapsulation change for the device in the interface configuration code may be implemented by changing the procedure the device driver is configured to schedule as a soft interrupt to process its packets, with any additional data structure accesses being deferred to software interrupt context. We also avoid dealing with complex encapsulation issues, like Ethernet encapsulated in PPP (or, worse, Ethernet encapsulated in PPPoE), at hardware interrupt level by pushing this work into a lower priority software interrupt.

Note, however, that we might not eliminate all interface configuration derived data parsing from the hardware interrupt service routine. For Ethernet in particular we might leave MAC address filtering as a responsibility of the device driver since this will allow drivers for hardware with very flexible MAC address filtering to avoid the software operation that drivers for less capable hardware might have to do, all while maintaining a uniform view of the driver's MAC address filtering capabilities for configuration software. Also, this simplification of driver code need not be permanent; if there is a reason to move protocol demultiplexing back into the driver (say to gain parallelism by being able to process IP and IPv6 packets through different software interrupt threads) it can be moved back later.

Software Interrupt Processing

A software interrupt is invoked to process the packets which have arrived from a hardware device and have been queued, unchanged, to that device's input queue. Only one software interrupt will be run to service any particular device input queue; if one isn't running when the device driver adds a packet to the queue the device driver will schedule one to run a procedure the device driver has been configured with, with the device driver's queue (and/or *ifd*?) as an argument. Once started the software interrupt will run until it empties the device's input queue, after which it will stop. Note that if the packet queue implementation allows the producer and consumer to run concurrently the device driver and software interrupt can run without explicit locking and, if the software interrupt routine runs until the queue is empty the device driver need only check whether it should schedule a software interrupt (which will no doubt require explicit locking against the software interrupt in case it is still running) only when the input queue it inserts the packet into was previously empty. This means that while a mutex to synchronize the software interrupt and the hardware device driver will exist that lock will only have two contenders, since it is exclusively associated with a queue with just a single producer and single consumer, and the mutex will need to be acquired less often when the load on the interface is higher (i.e. when the input queue more frequently holds more than a single packet).

In what might be a common case the software interrupt procedure might proceed as follows. Since the packets from the input queue are in on-the-wire, raw form its first action might be to extract an L2 key from the packet and look this up in a data structure like the one described above in the section titled "Layer 2 Interface Identification and Protocol Demultiplexing". This lookup with yield an incoming *ifl_index*, which it copies into the packet, a function handle to call to deliver the packet on and an indication of whether it should strip the packet's L2 header before delivery (if it doesn't find an entry the packet will be dropped as "not for us"). If the packet is to be processed as an IP packet it will strip the L2 header and then call the function, which is likely to be called `ip_input()`. That function will verify the IP header, run the packet through filter code if one is called for in the incoming *iff,* and then extract a <destination address><source address> lookup key which it uses to perform a lookup in the routing table indicated by the *iff*. The route lookup will return a matching route with a next hop, and the packet will be delivered to the next hop's output function. That function will generally do one of two things. If the packet is not of local interest it will be forwarded (or dropped if the machine is a non-forwarding host), then will be run through an output filter if required by the outgoing *iff*, and then have an outgoing L2 header (perhaps conveniently located in the next hop structure itself, fully formed) prepended, at which point the packet will be appended to the outgoing device's output queue. If this queue was previously empty the process might need to ensure that the device driver transmit software is running (which might require acquiring a mutex), but if other packets are queued it won't need to bother.

Alternatively, if the route lookup returns a next hop which understands the packet to be one of local interest, that next hop's output routine may process the packet through a filter applied to locally-bound packets and, if the packet survives that and is not fragmented, a key of the form

<destination address><source address><IP protocol><first 4 bytes of transport header>

will be extracted and used to find the best matching entry in the protocol demultiplexing table. The result of this lookup, which perhaps might belong to an established TCP connection the packet is associated with, will include a processing function the packet is delivered to. That function will append the packet to a queue stored in the entry and, if the queue was previously empty, will ensure that the queue is scheduled to be serviced by a kernel thread (if the queue was not previously empty it can assume that this was already done when the previous packet was added). At this point the software interrupt thread is done with this packet and can return to process another packet from the device input queue.

While what is described is certainly not the sole set of functions that may be performed in software interrupt context, it is useful to consider the types of operations that might be performed in a software interrupt routine, and those that might be better performed elsewhere. It is clear that the general goal of a software interrupt is to process each packet to the point where it can find a queue to insert it into for other software to service. If the system is a router the queue it finds may be the output queue of the hardware interface the packet is to be forwarded out of, but otherwise the queue will be one whose packets are served by a kernel thread. We have some reason to want to minimize the amount of work done per packet at software interrupt level since this context is a critical, single threaded resource through which all packets arriving from the hardware interface must be serially processed; once these packets have been distributed to queues served by kernel threads it may become possible to devote a much larger portion of the resources of the processor complex to perform subsequent processing. Balancing against this, however, is that fact that re-queuing a packet and initiating a separate thread to continue its processing has a significant cost associated with it, one that we might not wish to bear if we can avoid it, so as a starting point a rule-of-thumb for what is appropriate in software interrupt context is probably useful. In this case, without implementation experience as a guide, the following rule is simply proposed: software interrupt processing is limited to stateless packet processing, while stateful operations must be left to a kernel thread.

What the term "stateless packet processing" means in this context should probably be defined. All packet processing will generally result in the packet itself being modified, but our concern is with the rest of the kernel. A "stateless" process is one which, once processing of a packet is complete, leaves no indication of that packet having passed behind (beyond trivia like incrementing counters). Basic IP forwarding is a generally a stateless operation (in the absence of caches, which we eschew), which is why we might expect packets to often be processed from the input interface queue all the way to the output interface queue in software interrupt context. IP firewall filtering can also be stateless, in which case its processing would be done in software interrupt context as well, but may instead be stateful, in which case we'd expect packets to be filtered to be queued to a kernel thread instead. BPF filtering seems to be stateless, though sometimes its operation causes a copy of the packet to be made (then again, so does multicast forwarding). NAT operations are almost certainly stateful, so these would be done by kernel threads. Some kinds of packet encapsulations and decapsulations, say those involving simply adding an IP (with, say, a GRE) header and forwarding the result, or deleting one and forwarding the inner packet, are stateless and might be done in software interrupt context. Other encapsulations, for example encapsulating IP packets in a TCP stream or with certain encryption procedures, are stateful and would be performed in kernel thread context instead. Packet reassembly is stateful, so locally bound fragments of packets would be queued to a kernel thread for reassembly. The processing of packets though a resolving next hop (see `RNH_RESOLV` above) is stateful and would be done via a kernel thread (hold next hop processing may be as well, though a clever implementation might be able to avoid this by designing a "near-stateless", lockless procedure). The processing of ARP packets will very often result in stateful operations (installing routes, modifying next hops, updating timers), so interface L2 protocol demultiplexing procedures which find incoming ARP packets might be programmed to immediately deliver these to a kernel thread queue for processing (whether there is a single ARP kernel thread queue to receive all ARP packets, or a queue per *ifl* instead, is up to the implementer of that code). Additionally, we (rather arbitrarily) define any operation which results in packet data being added to an application's socket buffer to be a stateful operation, so once we determined that this is in fact where a packet's data is likely headed, which will often be what is learned from the L3 protocol demux lookup, the packet will generally be immediately appended to the appropriate queue for and further processing (based on the argument that once we know that the packet will need to be processed through a kernel thread at some point we might as well let the kernel thread, rather than the software interrupt, do as much of the subsequent work as possible). We hence expect that there will be a (at least one) kernel thread queue per open socket, plus kernel thread queues to handle stateful forwarding and other packet operations, plus whatever else is required to serialize access to whatever bit of modifiable state prompted the creation of a queue to protect it. The processing done in software interrupt context, then, is whatever is required to (statelessly) determine, with as fine a granularity as we can manage to design, what system state is going to need to be modified by the passage of that packet through the system, and then to arrange that the processing of packets needing to modify the same state be serialized through a common queue.

To step back from looking at particular trees (queues?) to get a view of the forest it is worth while to consider why it is so difficult to find an approach to make the current network stack more SMP-friendly. I think the fundamental problem is that the code in the current network stack was designed with the assumption that its job is to process each packet from where it came from (i.e. an incoming interface or a process socket) to where it is ultimately going (i.e. a process socket or an outgoing interface) with no regard at all for what data in the kernel is going to need to be changed in the process of getting that work done. This design yields a forwarding path where it is impossible to predict what data the processing of a packet is going to require changing; changing anything at all is fair game. Since we need to use locks (or some other form of serialization, like a queue) to protect data from concurrent updates to the same data, but processing a packet through the current network stack can change virtually any data it needs to to getting the packet where it is going, with no way to predict just what is going to change until it gets there, fine grain locking is essentially impossible; locks need to cover all networking data since any operation might need to change any of it.

The arrangement being proposed here in essence forces a set of constraints on the problem which require a software designer to think about the problem of implementing networking functionality a little bit differently. The data which needs to be modified by an operation needs to be explicitly identified since it needs a serializing queue in front of it through which all packets whose processing may modify this data must pass. The fact that all incoming packets whose processing will modify the data will pass through this queue first means that packets in other queues will not be modifying this data and hence may be processed concurrently. The remaining problem is to design the programming of the stateless packet forwarding core to arrange for packets whose processing requires access to this data to end up in the serializing queue; since stateless packet processing can inherently be performed concurrently as long as this does not violate the constraint against reordering packets in the same "flow" this arrangement yields a considerable amount of work that can be executed independently of concurrent operations.

Essentially the arrangement provides a scheme where fine-grained locks to protect data are avoided in favor of fine-grained queues to protect data. Rather than the thread processing a packet which needs to modify the data waiting for a lock giving it exclusive access to that data, it instead appends the packet to the end of a queue and, its work on that packet complete, continues on to work on something else. This is made efficient by collecting a core of stateless, read-only (and lockless) decision-making functionality whose basic purpose is to find the right queue to append the packet to. While many procedures may call this stateless core to deliver packets (e.g. software running in process context will use it to deliver the packets the application is generating), code running in a software interrupt context delivering packets arriving from an interface will in particular limit its processing to the stateless processing required to find the (first) queue.

Finally, by now it is probably clear that the functional distinction between "software interrupt context" and "kernel thread context" is a little bit hazy. Software running in either context consumes packets from a particular queue which have been placed in that queue by other software processes, with that processing either resulting in the packet data being inserted in some other queue or being delivered to an application's socket. Whatever processing is done by a software interrupt could be done by a kernel thread, since what software interrupts do (process packets from a queue) is what kernel threads do as well. The primary reason for software interrupts to exist as separately scheduled entities, and for us to restrict the types of operations which can be done in software interrupts to a subset of those allowed in kernel thread context, is for prioritization. Since most packets in the kernel will be passing through a relatively small number of queues, in particular those associated with hardware interfaces, and since these queues must be processed serially, it makes sense both to give the threads servicing these queues priority over other work while architecturally limiting the types of tasks which can be performed in this context to ensure that packets are promptly distributed to places which provide opportunities for greater parallelism in subsequent processing.

An additional, though minor, reason for constrained software interrupts is that if it were determined that the extra queuing stage we've added is much more detrimental to performance than I expect it to be this arrangement allows one to easily experiment with the elimination of one of the queues (in particular the device driver input queue) in favor of performing the work now done in software interrupt context directly from the hardware driver interrupt instead. I don't expect this to be necessary (and there are reasons to think it might be undesirable) but retaining the option may be useful.

Kernel Thread and Process Contexts

Kernel threads are executed in response to networking events which happen asynchronously with respect to process execution; with the architecture proposed here those events are the insertion of packets (or perhaps other messages) into service queues, events which cause kernel threads to be scheduled to process those packets. Software running in process context, on the other hand, is initiated by an application thread which executes a networking-related system call, perhaps to read data from or write data to a socket. In both contexts, however, we generally allow code to sleep for locks or memory allocations, and otherwise observe the same (liberal) constraints on what operations such code can perform. Software which can be executed in kernel thread context can also be executed in process context, and vice versa. For this reason we treat them together here.

Note that in kernel thread context the guarantee that the system must provide is that only one kernel thread at a time will be actively processing packets from any particular queue. When a packet is inserted into a service queue, however, it is necessary that the thread which inserted the packet ensure that a kernel thread has been scheduled to consume packets from the queue (if nothing is actively processing packets from the queue already) and, if not, to do what is necessary to arrange for this. This minimally requires that a thread which is inserting a packet into a queue be able to determine the state of the queue's consumer which will in general require a mutex to serialize access to this state. We hence expect a mutex to be associated with each service queue which will be acquired (briefly) before any operation to determine or update the state of the queue.

At this point we'll state some assumptions about the packet queues themselves, the associated locking protocol surrounding the state of the queue and its consumer, and the general behavior of the actors involved. The state information about the queue, protected by the mutex, will indicate whether a consumer is currently active or not, whether a kernel thread has been scheduled (but is not yet actively processing packets out of the queue) or not, and whether process context threads are waiting to become active or not. We assume that the consumer of a packet queue, once active, will always run either until it empties the queue or, if not, will schedule something to complete the job on its own. This means that a thread which has added a packet to the queue needs to schedule the queue for processing by a kernel thread only if there is no consumer currently active or scheduled (which it would determine by looking at the state of the queue after taking the queue's mutex). The queue itself will be a lockless multiple producer, single consumer implementation, with the function which enqueues a packet returning an indication of whether the queue was previously empty. The latter allows a thread which adds a packet to the queue to avoid the mutex operation altogether when the queue already has packets in it (it is guaranteed that the consumer will either be active or previously scheduled in this case) and, in the case where multiple threads add packets to an empty queue concurrently, ensures that only one of those threads will be subsequently required to contend for the mutex. We will also allow a thread operating in process context (i.e. executing a system call) to become the active packet consumer for the queue if it chooses to do so and if no consumer is already active. If a consumer is already active we will allow software in process context to sleep waiting for that consumer to complete, while if a scheduled kernel thread finds the queue already has an active consumer the kernel thread can simply skip processing the queue (after modifying the state to indicate that it is no longer scheduled) and continue by looking for other work.

The reason for this delicate dance is that we expect a queue's consumer not only to have exclusive access to the packets at the head of the queue but also to whatever state needs to be modified while processing those packets. The difficulty we seek to solve is that sometimes this state must also be modified by code executing system calls by applications, and to accomplish this that code, running in process context, must also become the queue's active consumer to acquire exclusive access to that state. In the typical case of a TCP socket the TCP connection's state must be accessed and updated not only while received packets are being processed out of the incoming packet queue, but also by system calls which send and receive data; TCP packets which are sent carry information about what data has been received and need to update the state to account for the new, unacknowledged data, and receiving data already received from a socket may require sending a packet to update the receive window to the remote end of the connection. Beyond the need to acquire exclusive access to a connection's state, however, there is also an efficiency argument to be made that system call code for sending and receiving data should process pending, unprocessed incoming packets from the queue before executing the action it was called to perform. Having the outgoing packets that the system call might generate (perhaps as a side effect, if the system call is to read data) reflect the most accurately up-to-date state possible can save sending additional packets later while having a data reader process unprocessed incoming packets to extract that data and return it to the caller can save it the trouble of sleeping on an empty socket buffer waiting for a kernel thread to run to do the work instead. As a result, not only is it necessary for code in process context to contend with a kernel thread for exclusive access to a connection's state, but it is efficient if the code in process context, once it has exclusive access to the connection state, also does the work that a kernel thread would have done had it run. This can save the cost of scheduling a kernel thread to do the work, is often protocol-efficient and is probably also computer resource efficient in that, if you need to bear the cost of reading the connection state out of memory and into the processor's cache, you might as well do as much as you can with that state once you've paid that price. For all these reasons we require process context code which needs access to the same state which a kernel thread would need to operate on to do the work which the kernel thread would otherwise need to be executed to perform.

Note, however, that there is no general need for operations on a socket to serialize themselves by acquiring the associated input queue and processing packets from it; the need to do this for TCP is a consequence of the state associated with the session which all operations on the socket will need to access and update. For sockets serving connectionless protocols, on the other hand, system calls to send packets may very well simply construct the packet they have been asked to send by adding the outer packet headers and then calling the stateless packet routing core to deliver these to the appropriate destination queue, leaving the processing of incoming packets (which the service queue is solely associated with) to kernel threads and/or system calls to receive packets. How processing of the service queue associated with a socket needs to interact with system calls performed on the socket is a design issue which needs to be resolved by the implementer given the constraints imposed by the protocol being implemented.

So far we've been considering service queues associated with open process sockets, but service queues are more generally useful than this. As mentioned previously, IP reassembly will very likely be implemented by a consumer of a queue of fragments of packets (perhaps one queue per routing domain or, to increase processing capacity, maybe multiple queues per routing domain with a hash on IP header fields being used to select which queue to add the packet to as a way to distribute the load) since packet reassembly will no doubt involve modifications to a data structure containing the fragments of incomplete packets which the thread processing a new fragment will require exclusive access to. In this case a kernel thread will be responsible for all processing of packets from the queue, and when it completes a packet, will perform a protocol demux lookup to determine the receiver of the packet and execute the associated processing routine to deliver it. ARP and ND would similarly do their work by processing incoming packets from service queues (perhaps one per *iff*, or one per routing domain) in kernel thread context, performing routing table modifications and sending responses as required, though it is possible that both ARP and ND may also have open sockets associated with user space applications (say, a DHCP server) which are using the protocol to management a part of a subnet's address space to which protocol packets associated with those addresses are delivered. A NAT implementation or a stateful firewall filter might also operate downstream of a service queue (or perhaps a group of service queues per routing domain, with packet load distributed by hashing), processing transformed packets back through the network stack for delivery. Any operation which requires stateful processing will need to be organized around a service queue which serializes the operations modifying that state, and will depend on kernel threads to do that processing.

Finally, it is probably worth pointing out a feature of this arrangement which may be useful. We have already seen that service queues can sit between the stateless forwarding path and sockets opened by applications, perhaps with processing of the service queue involving nothing more than copying packets from the service queue to the socket, or can sit in front of significant in-kernel processing functions having nothing to do with user-space applications which just require the packet serialization which the queue provides. It is the case, however, that the logic which results in a packet being added to a queue generally has no idea which type of processing is going to be done by the queue's consumer; in all cases that logic simply enqueues the packet and goes on to do something else. Because of this it seems like a very short distance to take functions which might be, or traditionally have been, implemented in the kernel and replace them with an implementation in a user-space application instead. That is, just remove the kernel processing in favor of code which simply copies the packets from the queue to a socket for an application to handle. This, plus an application interface for programming the stateless packet forwarding core delivery structures which provides applications with complete access to functionality that in-kernel code would have, might provide a way for many functions which might otherwise be implemented in the kernel to be implemented in user space instead. While there might be performance reasons not to do this for production use, this seems like a potentially very useful service to have available for the development of new networking functionality, with the core logic of the service being developed and debugged in user space where development tools are more effective, and then moved to the kernel only when it is known to work.

Packet Queues (or Message Queues?)

As we've mentioned previously (several times), packet queues are used to store packets to be transmitted through a hardware interface, which have been received from a hardware interface and which need to be processed by a kernel thread. It is intended that a single queue implementation be used for all these functions and, while much of what the implementation must do has already been described it is probably useful to reiterate and summarize it here again.

The properties we expect the packet queue implementation to have are these:

- The queue should be implemented as a lockless multiple producer, single consumer queue. That is, in general we expect that threads running on different cores in an SMP complex will be delivering packets to the same queue concurrently with a single dequeuer or, on a single processor core, that a thread in the process of operating on the queue might be interrupted at any point by a hardware or scheduler interrupt which will be adding packets to or removing packets from the same queue, and in all these cases we need the right thing to happen without the overhead of any additional mutual exclusion primitives (including blocking interrupts) if this is at all possible on the processor architecture (if it isn't, of course, we'll need to make do).
- No memory allocation should be required when adding a packet. This likely means that it must use link pointers stored in the packet mbufs.
- The queue should require only a minimal amount of data structure when empty, since there will be many of them and they'll mostly be empty most of the time.
- We want the enqueue operation to return an indication of when the operation just performed has unblocked the queue, that is that the result of this operation is that the queue's consumer should now be able to dequeue a packet (this usually means a signal which is true only when the queue was empty before the packet was queued, though the concurrency involved makes this a little fuzzier). The reason for wanting this is that if the queue's consumer is not currently running or scheduled to run (because the queue was previously empty) then we need the thread which adds a packet to the queue to also initiate the queue's consumer thread. Doing this without the indication would require every thread which added a packet to the queue to subsequently check the state of the queue's consumer to see if it was running or needed to be started, an operation which would almost certainly involve acquiring a mutex. By requiring that a queue's consumer continue to run until the queue it is servicing is empty, however, an enqueuing thread can avoid doing this check if it does not receive an indication that the operation unblocked the queue, and if multiple threads add packets to the queue concurrently we would expect that no more than one of them would receive the indication. This hence provides an optimization (whose cost is the memory interlocked and barrier instructions we'd be spending to implement the shared queue regardless) which should reduce contention for the mutex and otherwise improve efficiency under load (i.e. when queues are more likely to hold multiple packets).

The implementation I have in mind consumes about four pointer-sized objects worth of space for an empty queue, plus the associated mutex and queue state information, requires an atomic swap operation plus a memory barrier instruction to add a packet to the queue (dequeuing a packet is generally cheaper than this) and is lockless and non-blocking in operation except when the mutex must be taken. The latter should happen only when the queue is close to empty, with only a single producer thread contending with the queue's consumer for the mutex. The implementation has the desirable property of becoming more efficient under load, when queue is more likely to have multiple packets stored, since under these conditions producers and the consumer can proceed independently without a need to acquire the synchronizing mutex. It may be possible to develop a common API for all packet queues, though it is more likely that device input and output queues and kernel thread queues will require slightly different approaches because of the dissimilar procedures for initiating the queue's consumer.

We have been viewing these queues purely as queues of associated and undifferentiated packets, but for kernel thread queues in particular it may be useful to broaden that definition a bit, with two particular examples coming to mind. The first of these involves

ARP processing. We've already suggested that ARP protocol packets will be delivered to kernel thread queues (perhaps one per *ifl*) for processing, with the consumer of that queue being ensured of exclusive access to the state associated with the ARP cache. We've also noted, however, that the ARP instance maintaining an *ifl*'s routes will also have associated with it one or more resolving next hops (on the subnet route(s)) to which IP packets addressed to previously unknown destinations on the subnet will be delivered which must ultimately be processed by ARP by making modifications to the same state accessed to process ARP protocol packets. In this situation it may be convenient and desirable to queue both ARP protocol packets arriving from an *ifl* and IP protocol packets which have been output through a resolving next hop associated with the *ifl* to be added to the same queue and processed through the same kernel thread. This would require that metadata be included with the packet describing whether the packet is an ARP protocol packet or an IP packet (and, in the latter case, perhaps also indicating which resolving next hop was responsible for queuing it). Second, many of the network protocols which are processed in kernel thread context (including ARP and TCP) also require kernel timer services to provide support for various types of event timeouts. Since servicing timer events will require exclusive access to the same state protected by the serializing queue it may be efficient to provide indications of timeout events by queuing a notification of these to the same packet queue and processing these in kernel thread context as well. This may make it desirable to implement kernel thread queues as more generic message queues, with metadata associated with each queue entry indicating the nature of the message and with packets being treated as just one of several possible types of event messages that might be delivered through the kernel thread queue. Exactly how these queues should be treated is a subject for future consideration.

In summary, then, the (mostly) lockless multiple-producer, single-consumer queue implementation suggested here seems to provide a fairly powerful framework around which multiple concurrent threads can get their work done in parallel with a minimum of interaction, and in particular with almost complete avoidance of blocking of the software interrupt context threads which are likely to otherwise become the processing bottleneck and barrier to parallelism in the current arrangement. The implementation suggested here seems quite reasonable though, given the centrality of this data structure, finding a more optimal implementation for the purpose would be a valuable topic for future study.

Kernel Thread Scheduling

This topic is one which I know little about, that being the scheduling of kernel threads to process packets (or messages) from service queues. Some of the broad requirements are clear. When a packet is appended to a previously empty service queue it must be arranged that this queue come to the attention of a kernel thread which will run the code to process packets out of that queue. If no kernel threads are active when this occurs it may be the responsibility of the enqueuer of that packet to do what is necessary to start one. Since individual service queues may be processed independently and concurrently, separate kernel threads may be concurrently executed on all available processor resources to process these should the demand for their attention justify this. Once started, a kernel thread will exhaust the contents of each service queue it processes before moving on to another (or, if not, will arrange on its own that the queue will be inspected again), and will run until there are no more service queues in need of processing, at which point it will sleep until it is awakened again to provide service.

There are additional considerations which a higher quality implementation might deal with, though at this point these are guesses. The "best" kernel thread to process a particular queue might be one which is running on the same processor core as the ultimate recipient of the packet data, as this might help improve the efficiency of cache use. On the other hand, it might be better to invoke a kernel thread on a processor core other than the one running the thread which added the packet to a service queue as the queuing thread (running, say, in software interrupt context) may continue to run and running the kernel thread on a separate core may allow its processing to overlap that of the queuing thread. The number of kernel threads which are executing at once needs to somehow reflect the demand for their services, though running a smaller number threads to process a larger number of queues may be more efficient than having each single thread wake up, process a single queue and sleep again; on the other hand, if some processor cores are currently idle it might be useful to run additional kernel threads to try to complete the work more quickly while there is nothing else to spend time on. Since kernel threads are allowed to sleep for memory, locks or other I/O completion, sometimes running more kernel threads than there are processor cores to execute them may improve throughput. Implementation experience will no doubt reveal other factors (and perhaps prove some of the above to be fallacies), and it is certain that doing this well will require very considerable familiarity with both the details of the problem and the scheduling capabilities and costs associated with the host kernel, as well as considerable cleverness, all of which I lack.

For the purposes of finishing something which works correctly, if not necessarily well, a relatively simple approach is imagined for a first implementation unless someone suggests something better. When a packet is added to a previously blocked service queue, the enqueuing thread will check to see if the service queue has already been scheduled and, if not, will add the queue to a scheduler queue. The state associated with the scheduler queue will be the number of service queues scheduled and awaiting processing and the number of kernel threads currently running to process the queue. When no kernel threads are running the queuing thread will start one. The kernel thread will remove a service queue from the scheduler queue, call a processing function to consume its contents and then return to fetch another service queue from the scheduler queue. When a kernel thread dequeues a work queue from the scheduling queue it will also check to see if the number of entries in the scheduler queue exceeds the number of running kernel threads by some (programmable) factor and, if so, will wake an additional kernel thread to work on it before going on with its own work. If the kernel thread finds no service queues remain on the scheduler queue, however, it will put itself to sleep awaiting a signal to wake up and try again. If this works this leaves enqueuing thread responsible for moving the previously blocked service queue they have added a packet to to the scheduler queue, and to awake the first kernel thread if none is running at the time. Once this is done the dequeuing kernel threads will have the responsibility of starting additional threads in response to an increasing backlog of work on the scheduler queue (as measured by the number of service queues waiting there) until the work is consumed. This minimizes the work required to be done by the threads adding packets to the service queues at the expense of the kernel thread packet consumers.

For the actual implementation of the scheduler queue I know of several (imperfect) possibilities. The first is to use the same multiple producer, single consumer implementation as is used for packet queues. This works well for producers but will require kernel threads to synchronize access to the FIFO queue's output via a mutex (since the implementation allows for only a single concurrent consumer). The other alternative is to use a more complex multiple producer, multiple consumer queue implementation which I have. This eliminates most of the locking around queuing and dequeuing operations altogether, but my implementation requires that queue users be in exclusive possession of a “cookie” to use for the queuing function calls, something which complicates the queuing operation (especially since the maximum number of cookies in existence needs to be limited while the number of threads running to process packets onto a queue need not be). One or the other of these is workable, however, unless and until some better alternative can be invented.

Legacy Protocol Support

What we’ve been describing so far is an arrangement of the network functionality to yield a considerable amount of fine-grained parallelism in the network stack at the expense of a relatively extensive rearrangement of the networking code. While I believe this arrangement is general purpose and can advantageously accommodate most network protocols, it remains the case that the body of networking code for a Unix kernel is quite large, with support for a fairly broad and diverse set of network layer protocols. While none of these sees the use that IP and IPv6 do they are never-the-less sometimes useful and need to be retained, yet the task of updating all of them to operate well in the new environment is both daunting and rather unrewarding given their relatively limited use. There needs to be an approach to keeping this code running in its current form until such time (if ever) that someone gets around to refactoring the implementations.

I believe the way to do this with minimum damage to the code is to run the entire protocol stack in kernel thread context. That is, while the code will need to be modified to deal with the new interface configuration (which I believe is a fairly minor project; any problematic configuration can just be disallowed), we can retain the notion that the protocol has exclusive access to its data while it is running by queuing all arriving packets from all interfaces (identified by the generic L2 protocol demultiplexing) to a service queue consumed by a single kernel thread running the entire protocol stack. This will require application senders of packets to acquire the service queue to ensure exclusive access while processing outgoing packets, but this is no different than a TCP sender acquiring the receiving service queue to ensure exclusive access to the TCP connection state. By ensuring that only one instance of the protocol runs at a time we can retain an execution environment which meets the expectations of the protocol implementation while allowing the independent, concurrent execution of tasks for other protocols.

The intent, then, would be to refactor the major protocols to operate in the new environment but to retain single-threaded implementations of all other protocols deemed to be worth bringing forward.

Other Modifications and Bug Fixes

The changes to the BSD networking stack advocated in this paper attempt to improve the networking code in a number of dimensions. The representation of “interfaces” is partitioned to allow more flexible and natural configuration of interface arrangements commonly (and not so commonly) encountered in modern networking. The scalability of the kernel code in the presence of very large interface configurations (perhaps enabled by the new flexibiity) is significantly improved. Routing has been unified for unicast and multicast, and for locally-originated and received-from-an-interface packets. Almost all network configuration is fully dynamic; it can be added when it is needed and deleted when it is no longer useful. Multiple routing instances per protocol are supported when they usefully solve problems. The problem of moving packets from where they originate to where they need to be delivered has been divided into a stateless core, deploying data structures which need not lock out readers when being updated, and a stateful edge with serializing queues in between to reveal very fine-grained parallelism without broadly contended locks.

This section covers a number of issues which either require repair due to fallout from the above changes, or which are arguably broken in the current design, along with some ideas about how they might be repaired.

The Routing Socket

The routing socket is the primary method of configuring route and interface structures, and for determining their current state. It is clear that the routing socket protocol needs to change to accommodate the new structures which now need to be configured, about which the existing protocol has no idea. While it should be possible to retain compatibility with the existing socket protocol for simple configurations which align well with the constraints of the previous network stack, fully general configuration and monitoring of the new kernel is certain to require significant changes.

My own biases about how this should be handled going forward are as follows. The basic operations, those being ADD, CHANGE, DELETE, GET and GETNEXT are fine, though I suspect resolving next hop operations will also need to emit routing socket messages (concerning addresses in packets which have been forwarded through a resolving next hop), so a RESOLVE operation may also come to exist. I believe a drawback of the current routing socket design is the use of structure overlays for messages as these cause difficulties both when compiler type widths and alignment constraints change and by the fact that the particular variables an operation might need to include in a message may be variable while the values in a structure overlay always need to be filled in. I also think that it should be possible to include data in a routing socket message which the application and the interface code responsible for using the data in the kernel understand but which nothing else does, and which does not require the routing socket maintenance code to change every time someone invents a new configuration feature for some obscure interface type, which seems to suggest an implementation with the ability to pass raw byte strings from the message to the driver code, perhaps with syntax checking but without any need for semantic interpretation outside the driver code itself.

I hence believe that the best way forward with the routing socket may be a byte stream, Type-Length-Value encoded protocol which is defined to produce identical byte streams for identical configuration independent of the architecture on which the kernel is running (e.g. so the same routing socket code may be used by a 64 bit architecture and by its 32 bit compatibility libraries). This allows all fields in a message to become optional, allows a fairly clean separation between data which the routing socket code must interpret to determine the structure being configured (or queried with a GET or GETNEXT) and data which is only interesting to the structure-specific configuration code, but has a well enough defined common structure to allow the generic routing socket code to do basic syntax checking before presentation of the message fragment to the driver.

Note that while this may increase the costs of parsing a routing socket message a bit compared to structure overlays I believe this is irrelevant since I think the general case is that the cost of parsing a routing socket message is dwarfed by the cost of performing whatever action the routing socket message is intended to initiate. It is better to have a reliable, immutable message format with well understood rules for message construction even at the expense of a modest increase in parsing costs.

Socket Options and Socket Control Messages

Several features of the new networking stack, and misfeatures of the old networking stack, require support which will likely add socket options and extend the use of socket control messages (i.e. message control data passed via **sendmsg(2)** and **recvmsg(2)**).

The first of these problems is related to multiple routing instances. When a socket is opened on a machine configured with multiple routing instances it is necessary (given the initial implementation planned) for that socket to be associated with one of these routing instances prior to sending packets on the socket and prior to any bind() or connect() call which requires the interpretation of addresses. The reason for this is fairly straight forward. When packets are sent via a socket the code performing this work needs to know which routing table to use to route the outbound packet, while the effect of a bind() or connect() system call is to add an entry to a local protocol demux table associated with a routing instance and we need to know which routing instance's protocol demux table to insert these in, so knowing the routing instance is necessary for determining both outbound and inbound packet routing. To implement this we will likely add a socket option to specify the routing instance index (with routing instance zero being the default if another is not specified) so that subsequent system calls using the socket do the right thing.

A second issue we have involves the routing of (connectionless) packets sent to a multicast address, or 255.255.255.255 on routers and hosts with more than one logical interface. Such packets must be sent out an interface (just one, if we are a host) but the destination address gives no hint about which of several interfaces it was intended that the packet leave the box through. We decide this using the following strategy. First, for each routing domain, the kernel selects a "default" interface which multicasts and non-subnet broadcasts are sent out of if no information other than the destination address is provided. The selection procedures will likely prefer multicast-capable multiaccess network interfaces (e.g. ethernets) over others, as this is most likely to be useful, and this interface will also be the default interface for multicast group joins. If the sender wishes the packet to leave the box on another interface it will express this by either setting a socket option to indicate the *ifl_index* of the interface it prefers the packets to leave through (and, maybe, for multicast groups to be joined on by default) or it will include a control item in the control data given to **sendmsg(2)** with the packet specifying the *ifl_index* of the outgoing interface. Our preference going forward will be to name interfaces via their *ifl_index* for routing purposes (we are in fact naming an *iffamily* in this case, the network protocol is identified by the socket protocol family), perhaps providing an interface name to *ifl_index* conversion function for user interface needs.

A third issue I would like to resolve is related to the implementation of servers for connectionless protocols like UDP, with example applications being bind and ntpd. These applications have the problem of having to craft and send reply packets whose local address matches the destination address in the query they are replying to under the constraint that the current kernel provides no way to tell the destination (i.e. the address on the local machine) a received UDP message was addressed to (it does provide the source address of the sender, but this is insufficient to be able to send an unsurprising reply). To work around this these daemons generally scan the interface configuration and open a socket per local address, with each socket bound to a different local address of the machine, so that they can determine the particular local address a packet was addressed to by noting the socket it arrived on. The difficulties with this are numerous when the interface configuration is large. First, it requires that the application open a socket per local address (even if packets it must respond to will never be addressed to most of the addresses; it can't predict this) which, on a router with a huge (logical) interface configuration, can require the application to maintain a huge number of open sockets. Second, if the interface configuration is dynamic the application must go to the additional trouble of watching for interface configuration changes, say by watching for changes on the routing socket. Since there is no way to make a routing socket monitor purely reliable (the input buffer must be finite), however, the application must be prepared to rescan the interface configuration if routing socket monitor messages are lost. If the interface configuration is both huge and highly dynamic (say the box is a PPPoE concentrator) the end result can be that the application is required to open thousands of sockets to do its job and spend a huge amount of resources tracking changes to the configuration. The idiocy of this reaches epic proportions when you want to run the NTP daemon on a router to keep the router's clock in sync, configuring it with two or three peers, and discover that to send and receive a half dozen packets per minute the daemon needs to open 12,000 sockets (since you have that many interfaces) and spend a large amount of CPU time watching this huge interface configuration change via the routing socket.

To make such servers scalable I would like to add configuration to allow the destination address in a packet (along with, say, the incoming *ifl_index* and other existing data) to be reported in the **recvmsg(2)** control message, so the daemon can learn this information if it needs it. Furthermore, I would like to arrange the facility so that if the control message with this information as reported by **recvmsg(2)** is returned unchanged to a call to **sendmsg(2)** then the kernel itself would construct a reply packet with appropriate addresses and routing without the application needing to do more than record this metadata when a packet is received and return it to the kernel for the reply. In this way a UDP service can often be written to do the right thing on a multihomed host or router without any need to deal with interfaces (just like a TCP service application can do).

There are probably other issues that would be well addressed by adding socket options and/or control message configuration, but these come immediately to mind.

Receiving Packets Not Addressed To Us

Certain applications have a need to receive packets which arrive on an interface which are not, or appear not to be, addressed to the machine, i.e. packets for which a route lookup on the addresses in the packet would not yield the result that the packet is of "local interest". Most of these cases involve protocols implemented by routers, in particular PIM, IGMP and RSVP, where the packets are marked by a "Router Alert" IP option (see RFC 2113) which prompts the router forwarding path to look at them. A more important example of this, however, is a DHCP client implementation, which needs to receive packets addressed to a local address which the machine does not yet know is its own (since the packet in question is one assigning that local address to the machine). The latter is particularly interesting since it is a very commonly required application for hosts where the implementation strategy seems to be to resort to BPF, a sure sign that the forwarding path is missing required functionality.

I would like to address this by implementing a small, per-interface protocol demux style lookup, keyed with a

<IP protocol><first 4 bytes of transport header>

tuple (i.e. the same as the protocol demux lookup done on packets of local interest after the route lookup, but without addresses) which is performed prior to the address route lookup to determine if the packet might be of local interest independent of addressing. If so the packet would be queued to a service queue for delivery to the appropriate application socket. This lookup would be cost-free if there were no entries in the early protocol demux table, and would generally be very cheap since the number of entries otherwise would never be large (it would certainly be very cheap compared to invoking the BPF machinery to do the same thing; BPF needs to do the same protocol match but additionally duplicates the packet, with one of the copies continuing through the normal lookup path). The end result for a DHCP client would be that all DHCP packets arriving on the interface would be delivered to the DHCP client's socket independent of addressing.

To establish this mapping a DHCP client might open a socket, bind it to receive DHCP packets, and then execute a socket option system call to duplicate the socket binding to the early protocol demux structure for the interesting interface(s). This, coupled with a second socket option informing the kernel that it should believe any source address the application specifies for outgoing packets (see the previous section concerning source address selection for datagram protocols; this is required so that the daemon may send DHCP requests with a source address of 0.0.0.0) and the proposed networking stack's full support for sending packets addressed to 255.255.255.255 would allow DHCP client packets to be both received and sent without resorting to BPF, and with the same mechanism providing service in a slightly different fashion for Router Alert option carrying packets.

"Default" Layer 2 Protocol Demultiplexing

The notion of a layer 2 protocol socket, associated with an *ifl* and with the application registering interest in particular packets identified by an L2 protocol identifier has already been introduced, with examples being an implementation of the PPPoE discovery protocol which arranges to receive all packets with an ethertype of 0x8863 arriving on an Ethernet *ifl*, or a PPP implementation which arranges to receive all packets with a PPP protocol type with the high order bit set (i.e. a one-bit prefix of the protocol ID, capturing LCP and NCP packets) arriving on a PPP *ifl*. Matching packets would be delivered to the application socket with or without the L2 header, with the latter choice being made via a socket option, and the same socket would allow sending of similar packets out the interface. This would be implemented by adding an entry to the *ifd*'s L2 lookup structure, which is normally populated by *iff* configuration, for the *ifl*/protocol ID of interest.

This suggests another possible use of the structure for a purpose that BPF is (ab)used to do now, that being for an application which wants to receive all traffic coming in an interface and to send fully encapsulated frames out the same interface. An example of this might be the Vax emulator SIMH, which does this to connect its simulated Ethernet interface to the real word, or perhaps a network stack running in user space for development. To implement this would be quite straight forward given the proposed arrangement. An *ifl* would be configured on the interface hardware to be dedicated to this purpose and then the application would open an L2 family socket on the *ifl* and bind it to a zero-length protocol ID prefix (just a small difference from the one bit prefix required for a PPP daemon to capture its packets), which effectively matches all packets. This, plus a socket option configuring it to leave the L2 headers intact, would effectively capture all frames which arrive on the hardware interface, as well as allowing raw frames to be sent out the interface, from a relatively normal application socket without resorting to BPF.

The above works when the machine has an otherwise unused interface to devote to the application, but also when the machine has only a single interface which must be shared between the native host operating system and the simulator if bridging is implemented. In this case one would configure the single hardware interface as a bridging interface and then configure two bridging pseudo-interfaces, distinguished by the MAC addresses assigned to each, and add them to the same bridge group (i.e. routing instance). One of these pseudo-interfaces would be used for host interface configuration while the other would be dedicated to the simulator via an L2 socket opened on the pseudo-interface *ifl*, just as it would with a dedicated hardware interface. The interface configuration described here provides quite flexible arrangements which may be configured to provide applications with a common view even if the underlying piping is distinctly different.

User-Space Implementations of Kernel Protocols

A number of protocols are typically implemented in the kernel, including ARP, IPv6 ND and IGMP. This is done partly from tradition, and partly because these protocols are required to interact with other parts of the kernel in a way which has been easier to implement in the kernel than across the system call interface. For example, ARP is required to take actions (normally involving installing routes and/or updating next hops) not only when it receives ARP packets from an interface but also when packets addressed to previously unknown subnet destinations (i.e. forwarding through the subnet's RNH_RESOLV next hop) or when an application installs a route with an unknown next hop on the subnet (whose next hop lookup finds the RNH_RESOLV next hop). Similarly, IGMP's actions are prompted by multicast group joins executed on various sockets in the system which the protocol implementation must receive notification of. Each of these protocols must hence not only send and receive protocol packets but also receive notifications of related events and state changes in other parts of the system, and it has traditionally been convenient to pull these together inside the kernel. Unfortunately, in traditional kernels, once a protocol and its functionality has been implemented inside the kernel it has not been possible to get it back out even if there were some reason to prefer a user space implementation instead.

The best example of this is probably the implementation of an IPv4 DHCP server. DHCP servers already populate the routing table with routes which would otherwise be learned by ARP. Additionally, DHCP servers require a protocol to determine if hosts using addresses in the managed range are alive or not, and the very best protocol to use for this is ARP, yet on traditional BSD machines this protocol is unavailable to them (I think the standard DHCP server implementation for Unix uses ICMP ping instead, but this is a poor substitute). This is not the only example, however. IGMP can be implemented in the kernel for a multicast host, but its function needs to be assumed by an application process (implementing multicast routing protocols) should the machine transition to being a multicast router. I suspect there are similar issues involving IPv6 DHCP and ND, and I believe there are cases where it would be useful to override the kernel's automatic selection and negotiation of an IPv6 link-local address with a user space procedure to do the same.

The general approach to kernel protocol implementation taken here, then, will be to ensure that any kernel protocol implementation can be overridden by a user space implementation which assumes the same responsibilities. For some of these, like a DHCP server implementation of ARP, we will implement a demultiplexing facility so that the DHCP server can arrange that only the ARP packets involving the address range it is interested in are delivered to it, with the kernel implementation retaining responsibility for the rest. For others an application implementation may replace the entire protocol instance.

The cost of this will be not so much packet delivery but in propagating the intra-kernel signaling the in-kernel protocols use out across the system call interface so that an application implementing a protocol can receive these instead. In some instances we've talked about already we have ensured this will work by defining a common mechanism to be used by both in-kernel and user-space applications (e.g. resolving next hops used by in-kernel ARP and user-space multicast routing); replacing the in-kernel application with a user-space implementation involves deploying a mechanism which already needs to exist. In other cases it will need to be designed to work like this.

Packet Time Stamping

In a typical kernel time-of-day clocks derive their time from an underlying free-running hardware counter (or, less commonly now, a count-down timer or some other hardware time source; rarely are times of day derived by counting interrupts now), with the algorithm for obtaining a time of day being to sample the underlying hardware counter and then offer this value to code which performs the arithmetic needed to convert it to a time of day. Often the hardware counter itself can be exceedingly cheap to sample. On these machines the cost of obtaining a time of day is dominated by the cost of doing the arithmetic (in particular, the cost of accessing the constants needed to perform this computation).

Let's assume, however, that a clock implementation can be designed so that deriving a time-of-day timestamp from the raw hardware counter output can be done a considerable amount of time after the hardware counter is sampled. That is, the raw output of the underlying hardware counter (call this a `tickcount_t`) can be taken now, but the expense of converting it to a time of day only paid later should this turn out to be useful. This means that on platforms where obtaining a raw `tickcount_t` is very cheap it might not be particularly expensive to obtain a `tickcount_t` when a packet enters the kernel (via a hardware interface, or perhaps when a packet is constructed in process or kernel thread context) and record this in packet metadata. If the packet subsequently arrives somewhere which is interested in the time the packet arrived at the box the `tickcount_t` can be converted, but otherwise the only cost is obtaining the raw `tickcount_t`.

For machines where obtaining a raw `tickcount_t` has a modest cost, then, I would like to unconditionally timestamp every packet which arrives through an interface with a `tickcount_t` obtained as close to the packet arrival as is convenient, and to include this timestamp in the packet's mbuf metadata. This would serve (at least) three purposes:

- If the packet arrives at a socket which wants to know the time of the packet's arrival (an NTP implementation might want this) it can be converted to a time-of-day timestamp and provided to the application with the packet.
- If the packet arrives at a socket, or other consumer, which doesn't care about the timestamp, the raw `tickcount_t` bits can instead be stirred into the system's random number pool. This should increase the availability of high quality entropy for the pool.

- As a measure of performance for kernel network stack tuning and debugging, the timestamp carried in the packet combined with a recently taken timestamp can be used to directly measure the time it took for a packet to get from where ever it entered the kernel to where it left. This provides a useful performance measure to guide development and optimization of the networking code (i.e. to find parts that are slow), one which implements the measurement in a way complementary to (rather than duplicative of) the standard how-many-packets-per-second hero test.

I have a system clock design which permits and encourages this use, as well as correcting some misfeatures of the current clock code and providing an enhanced base for the development of new time synchronization algorithms. I would like this implemented as well.

Network Configuration Locking

While basic packet forwarding through the kernel is designed to be primarily lockless, to limit locking operations to a broad set of narrowly-contended locks around queues and to further minimize the conditions under which even those locks need to be taken, it is the case that the initial plan still calls for a “big lock” around all network configuration. That is, every process or kernel thread which wants to modify interface configuration, or install or delete a route to any table, or change a next hop, or add or delete an entry in the protocol demux table, will be required to acquire a single “network configuration” lock before doing so. Note that acquiring this lock does not block packet delivery, which can continue on concurrently, but does still block other processes which need to alter the shared network configuration structures.

Part of the reason for planning on doing it this way initially is that it is simple. It avoids the need to have to think about potential concurrent accesses to interface data structures, since these may be hard to make suitable for lockless readers (note that packet forwarding operations generally don’t need to look at interface structures other than individual *iffamily* structures and *ifdevice* output queues; any other information they need from interface configuration is made available separately in the form of data structures designed for concurrent lookups, like the L2 protocol demux structure), it generally avoids the problems of acquiring multiple locks when a particular operation requires changes to a broad swath of data structures in the forwarding core, and it particularly avoids having to think about which structures could be independently locked prior to having an implementation to look at to determine this.

On the other hand, I am aware that this lock will be broadly and frequently contended for. For example, connecting or binding a socket will require this lock to be taken to insert an entry in the protocol demux table, and closing the socket will require it to be taken again to remove that entry, so every individual socket which comes into existence requires the big lock to be taken twice. While it is clear that this will be hugely better than the current single-threaded forwarding implementation, it still may be far from optimal.

I wanted to point this out only to note that I understand this needs improvement. Identifying opportunities for finer grained locking among the various configuration data structures is a high priority for future work once a basic implementation is completed and is functional, but I believe prior implementation experience is necessary to determine the frequently executed configuration code paths which would guide the establishment of finer grained locking policies.

Summary

This paper proposes a re-architecting of the BSD networking code to provide more modern, more flexible functionality implemented within a much cleaner framework, and to enable fine grained parallelism for packet delivery with minimal locking in an SMP environment. The particular changes proposed are as follows:

- Redefine network interface configuration, and partition interface structures, to reflect the divergence between the hardware view of an “interface” and the attributes of an “interface” that define a network protocol’s view. Arrange structures in a way which makes it easy to determine layer 2 encapsulations for outgoing packets and layer 2 demultiplexing for incoming packets.

- Provide a data-driven layer 2 decapsulation lookup structure, populated by interface configuration and by socket bindings for layer 2 protocol applications, to allow incoming packets from an interface to be delivered to the appropriate protocol session processing function, with the incoming logical interface correctly identified, without reference to the interface configuration structures themselves (the data from these which an incoming lookup is required to know is instead added to the lookup structure as a side effect of interface configuration). Design this data structure so that lookups can be performed concurrently with modifications so that network configuration need not exclude packet forwarding.

- Rearrange route lookups so that the destination(s) of all packets can be determined from address information in the packet (perhaps refined by packet metadata for locally originated packets) with reference to a single table containing all the information required to classify packets by address. Deliver packets through procedure handles in route next hops whose functions are chosen to do the right thing for packets with those addresses. Design this structure so that lookups can be performed concurrently with modifications so that, again, network configuration need not exclude packet forwarding.

- Provide a consolidated protocol demultiplexing structure to directly identify protocol sessions, where possible, for packets determined to be of local interest by the previous lookup. Design this structure so that lookups can be performed concurrently with structure modifications.

- Implement multiple routing instances by essentially providing multiple, independent instances of the above two structures (a less onerous task with all lookups required to direct a packet being consolidated into a small set of data structures). Provide configuration to define when table packets arriving on an *iffamily* should be processed through and where interface routes generated as a side effect of interface address configuration should be stored. Provide mechanisms for applications to select (or have selected for them) the table instance used to route outbound packets sent on each socket.

- The above structures perform stateless forwarding procedures. Implement an intermediate queuing stage ahead of stateful packet operations so that packets whose subsequent processing requires the modification of common shared state may be serialized through a single thread of execution, while simultaneously revealing other packet flows whose processing involves modifications to independent state (because those packets are queued in other queues) and hence which can be executed concurrently. Provide a queuing implementation which allows multiple, concurrent threads to add packets to a queue without locking. Implement an initial (functional, if not optimal) scheduler for these tasks.

- Implement the functionality and procedures required to ensure scalability to large (logical) interface configurations, to eliminate (for common applications, at least) the use of BPF as a way to send and receive packets, to make multiple routing instances useful and to otherwise enhance the functionality of the network stack.

This is a large project, but not excessively so. I believe that most of this work is necessary either to make good use of the resources available in an SMP processor complex or to accommodate more modern network technologies, and that as a side effect of the latter will also provide the configuration tools and framework to build much more functional forwarding paths. I have not explicitly addressed firewall and security functionality, since these lie somewhat outside the extent of my expertise and are not strictly required for a first deliverable, but I believe the basic mechanisms provided by this framework can only enhance the ease with which this can be implemented.