Thermalcircle.de

climbing the thermals

linux, kernel, routing

Routing Decisions in the Linux Kernel - Part 1: Lookup and packet flow

In this article series I like to talk about the IPv4 routing lookup in the Linux kernel and how the routing decisions it produces determine the path network packets take through the stack. The data structures representing routing decisions are being used in many parts of the stack. They further represent the basis for route caching, which has a complex history. Thus, it is useful to know a little about their semantics. Further, the Linux kernel implements a lot of optimizations and advanced routing features, which can easily make you "not see the forest for the trees" when reading these parts of the source code. This article series attempts to mitigate that.

Articles of the series

- Routing Decisions in the Linux Kernel - Part 1: Lookup and packet flow
- Routing Decisions in the Linux Kernel - Part 2: Caching

Overview

A lookup into the routing table(s) is part of network layer handling of IP packets in the kernel. Based on the destination IP address of the packet, the lookup tries to find a best matching routing entry and thereby determines what to do with the packet. Figure 1 represents a simplified block diagram of the packet flow through the kernel and shows the two most relevant places where the routing lookup is performed. Other items like the *Netfilter* hooks[1] are shown as well to provide some orientation.

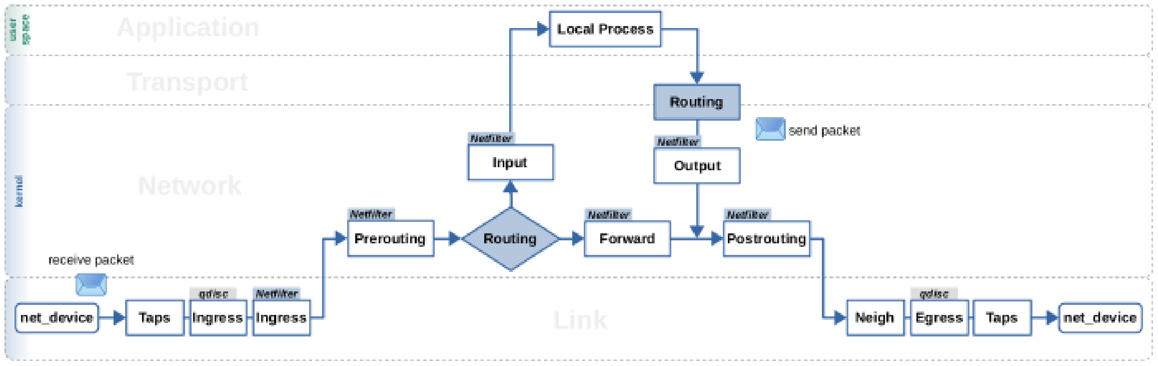


Figure 1: Simplified Overview: Routing lookup, packet flow and Netfilter hooks (click to enlarge)

As you can see, the routing lookup is performed for frames received on the network, once they are identified as IP packets, reached network layer and traversed the Netfilter *Prerouting* hook. It determines whether those packets are to be handed to transport layer to be received by a local socket or whether they are to be forwarded and then sent out again on the network. The routing lookup is further performed for locally generated packets coming from the transport layer[2], to determine whether there is a route for these packets to be sent out on the network. For forwarded as well as local outgoing packets the routing lookup determines the output network interface and the next hop gateway IP address, if existing. But how is the decision made by the lookup actually being applied to a network packet? Well, it is being attached to the packet. More precise, it is attached to the *socket buffer* (skb) representing the packet[3]. Resulting from the routing lookup, an object representing the routing decision is allocated or taken from cache and then attached to the network packet (skb) as shown in Figure 2. This object contains necessary data like output interface and next hop gateway IP address. It further contains function pointers, which lay out the path that packet takes through the remaining part of the kernel network stack. The focus of this article is on this attached "routing decision". It is often also referred to as the "destination cache".

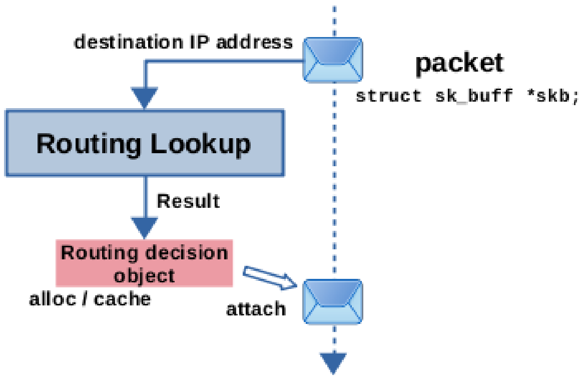


Figure 2: Resulting from Routing Lookup, a routing decision object is created or taken from cache and then attached to network packet.

Of course, the routing subsystem of the Linux kernel is heavily optimized, as well for speed to achieve high packet throughput as for keeping memory consumption at economic levels. Further, it provides advanced routing features like *policy based routing*, *virtual routing*, *multipath routing*, sophisticated caching features, support for IPv4 and IPv6, support for multicast packets, features like *receive path filtering*, and so on and so on. As a consequence, once you start looking into details you'll realize that several of my statements above are blatant oversimplifications. The routing lookup is of course not actually performed for EVERY network packet and also not necessarily only in the two places shown in Figure 1. Further it is also not necessarily just based on the destination IP address of a packet. However, when reading kernel source code, it is very easy to get lost in the details of all those advanced routing features and optimizations and "not see the forest for the trees". My intention in this article is not to try to "explain everything", but instead to explain the path which common *unicast* IPv4 packets take through the network layer part of the kernel network stack as a result of the routing decision. This should give you a lifeline to hold on to while reading these parts of the kernel source code.

The Routing Lookup

The Routing lookup brings several layers of complexity and its function call stack actually goes quite deep. I tried to illustrate some of the functions of this call stack in Figure 3. Be aware that the functions shown here are by far not all. I just selected the ones which I considered most relevant. In the receive path of IPv4 packets, shown in Figure 3 on the left, the routing lookup is done within function `ip_rcv_finish_core()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L315]. This function calls `ip_route_input_noref()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L2433], which is an intermediate function, not shown in Figure 3. Several calls deeper into the call stack, `ip_route_input_slow()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L2223] is called, which performs several checks before and also after the actual lookup, which is done within function `fib_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L364], and also takes care about creating, caching and attaching the resulting routing decision to the network packet.

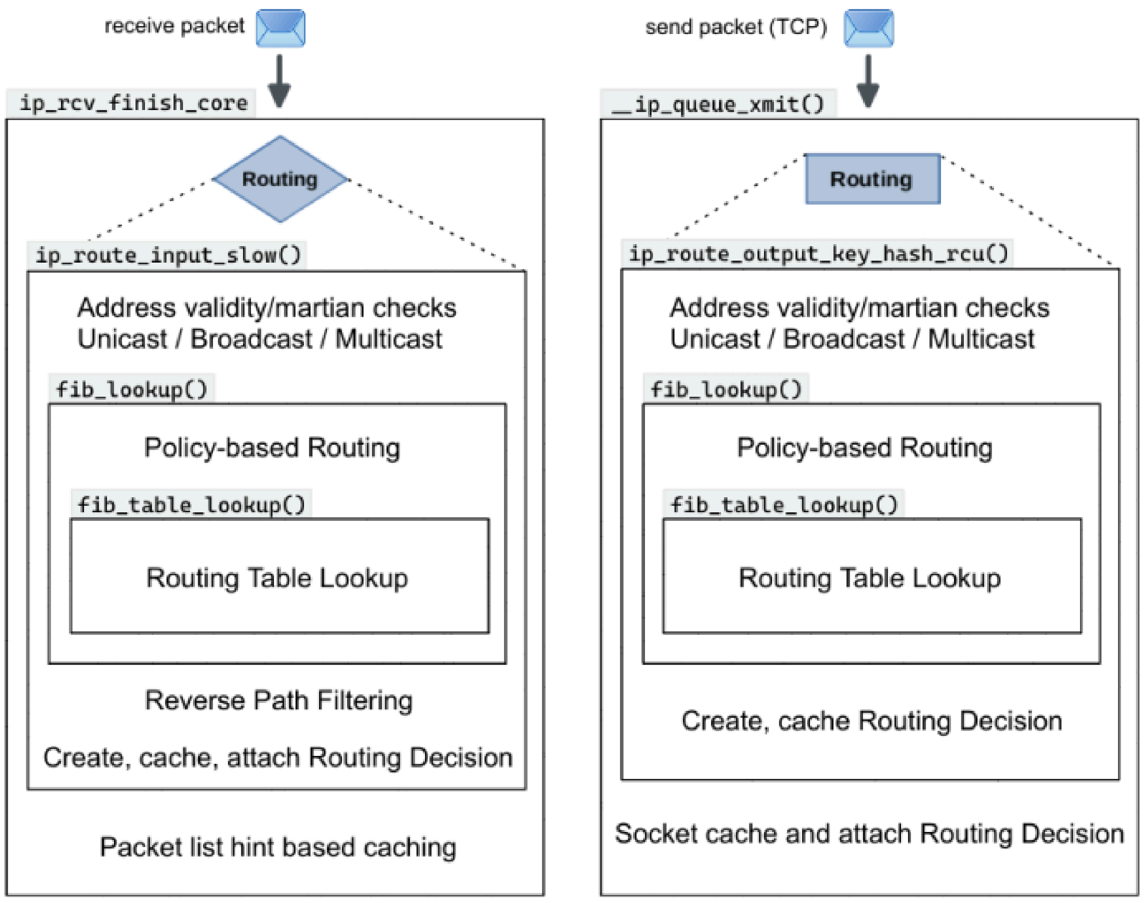
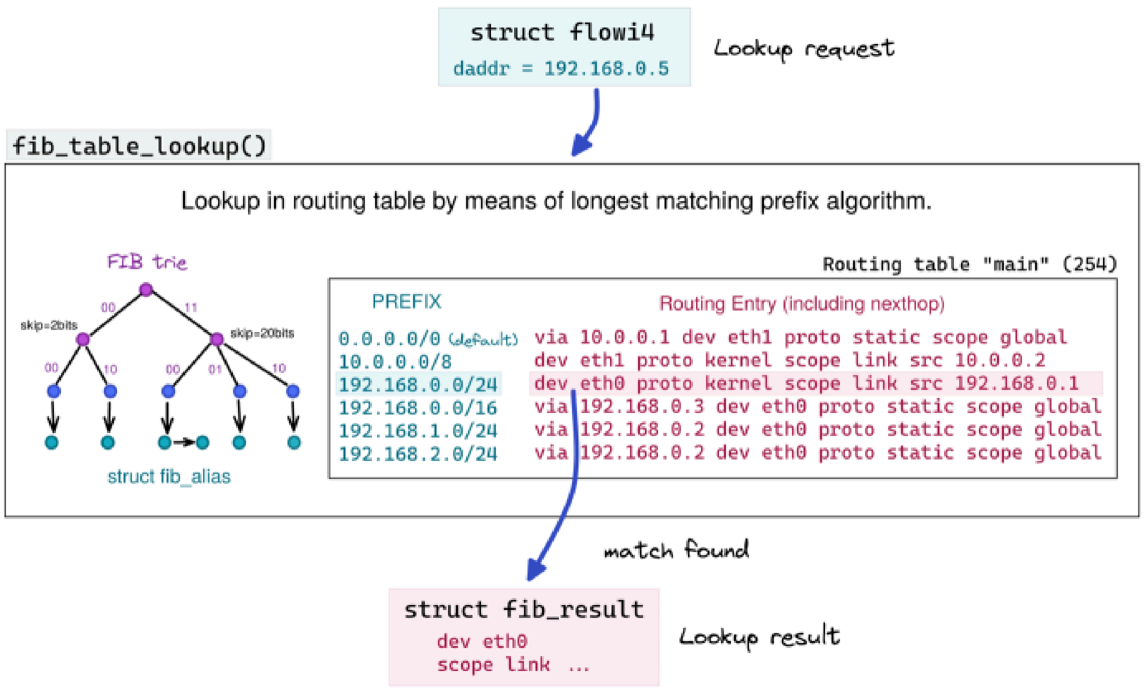

receive packet
ip_rcv_finish_core
Routing
ip_route_input_slow()
Address validity/martian checks
Unicast / Broadcast / Multicast
fib_lookup()
Policy-based Routing
fib_table_lookup()
Routing Table Lookup
Reverse Path Filtering
Create, cache, attach Routing Decision
Packet list hint based caching
send packet (TCP)
__ip_queue_xmit()
Routing
ip_route_output_key_hash_rcu()
Address validity/martian checks
Unicast / Broadcast / Multicast
fib_lookup()
Policy-based Routing
fib_table_lookup()
Routing Table Lookup
Create, cache Routing Decision
Socket cache and attach Routing Decision

Figure 3: Routing lookup as being called on the packet receive path (left side) and on the TCP local output path (right side); showing most relevant functions of its call stack (click to enlarge)

In the local output path of an already established TCP connection[4], shown in Figure 3 on the right, the routing lookup is done within function `__ip_queue_xmit()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L455]. This function calls `ip_route_output_ports()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L157], which is an intermediate function, not shown in Figure 3. Several calls deeper into the call stack, function `ip_route_output_key_hash_rcu()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L2652] is called, which performs several checks before and also after the actual lookup, which is done within function `fib_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L364], and also takes care about creating and caching the resulting routing decision. However, attaching that routing decision to the network packet is not done here in this case, but instead in the outer function `__ip_queue_xmit()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L455]. As you can see, both the receive path and the local output path got function `fib_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L364] in common. This function takes care about *policy-based routing*. Finally… a few more functions further down the call stack, function `fib_table_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_trie.c#L1432] is called, which contains the routing lookup algorithm, and thus performs the actual lookup into a routing table. In the following sections I'll explain the lookup from the inside out; thus, starting with the innermost function `fib_table_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_trie.c#L1432]. BTW: The term FIB [https://en.wikipedia.org/wiki/Forwarding_information_base] stands for *forwarding information base*, which is a more generic term for routing tables and MAC tables.

fib_table_lookup()

Function `fib_table_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_trie.c#L1432] performs the actual routing table lookup, more precisely the lookup into one single routing table. As systems usually got several routing tables, the code calling this function needs to specify which routing table shall be consulted. In the most common case this of course is the *main* routing table, which is the one that is shown when you enter command `ip route` without further arguments. Function `fib_table_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_trie.c#L1432] needs to be fed with an instance of `struct flowi4` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/flow.h#L69]. This instance represents the "question" or "request" you like to give to the routing subsystem. The struct contains several member variables. However, the only one relevant for function `fib_table_lookup()` in practice is member `daddr`, which holds the IP address for which the lookup shall be performed. If the lookup finds a match, then the function spits out an instance of `struct fib_result` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L165], which – as the name suggests – represents the lookup result. If no match is found, an error code is returned. This can e.g. happen, if no routing entry matches and no *default route* `0.0.0.0/0` is specified within that routing table, which else would function as a "catch all" match.



PREFIX	Routing Entry (including nexthop)
0.0.0.0/0 (default)	via 10.0.0.1 dev eth1 proto static scope global
10.0.0.0/8	dev eth1 proto kernel scope link src 10.0.0.2
192.168.0.0/24	dev eth0 proto kernel scope link src 192.168.0.1
192.168.0.0/16	via 192.168.0.3 dev eth0 proto static scope global
192.168.1.0/24	via 192.168.0.2 dev eth0 proto static scope global
192.168.2.0/24	via 192.168.0.2 dev eth0 proto static scope global

Figure 4: The actual lookup into the *main* routing table (click to enlarge).

Figure 4 shows the *main* routing table with some example content. At the surface level you can think of a routing table as a list of *key* + *value* pairs. The *keys* are network *prefixes*. In other words, they are IP addresses + subnet masks. The *values* are the *routing entries*, which contain data like the output interface, the next hop gateway address (if existing) and so on. What the lookup does is in principle a bitwise comparison of the network part of the *prefixes* in the table to the corresponding bits of `flowi4` member `daddr` to find the *longest matching prefix*. Figure 4 shows an example lookup for IP address `192.168.0.5`. You can see that *prefix* `192.168.0.0/24` matches. *Prefix* `192.168.0.0/16` and the *default route* `0.0.0.0/0` both also match, but `/24` is the longest match among these three candidates and thereby wins. The mentioned instance of `struct fib_result` is initialized with data from that matching *routing entry* and given to the calling function. Of course, as you probably can imagine, this lookup does not simply loop through all *prefixes* of the routing table to perform a bitwise comparison operation for each one. That would by far be too inefficient. After all, Linux is not only used in small embedded systems, PCs and edge routers. It is also used in core routers within the backbone of the Internet. In the latter case, routing tables can have several hundred thousand entries. Thus, an algorithm is required, which scales well with these numbers and still provides efficient lookups even within huge routing tables. Since kernel 2.6.39 [https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3630b7c050d9c3564f143d595339fc06b888d6f3], the *longest matching prefix* lookup algorithm used for IPv4 is the so-called *FIB trie* algorithm[5], which is also referred to as *LC-trie*. I'll not attempt to describe that algorithm here. It is a complex topic, which would deserve its own article (probably several ones). However, I'll give you some hints and references, in case you like to dig deeper: The algorithm places the *prefixes* of the routing table in a *binary trie* data structure and applies optimization techniques like *path compression* and *level compression* to the *trie*. The blog article IPv4 route lookup on Linux (Vincent Bernat, 2017) [https://vincent.bernat.ch/en/blog/2017-ipv4-route-lookup-linux] describes in an excellent way what that means and how it is implemented in the Linux kernel. The *FIB trie* algorithm itself is fully specified in paper IP-address lookup using LC-tries [https://doi.org/10.1109/49.772439], published 1999 by S. Nilsson and G. Karlsson in the *IEEE Journal on Selected Areas in Communications*. Some good background knowledge related to that is e.g. provided in chapter 2 of book High Performance Switches and Routers [https://onlinelibrary.wiley.com/doi/book/10.1002/0470113952], published 2006 by H.J. Chao and Bin Liu. Further, the official kernel documentation provides some implementation notes [https://www.kernel.org/doc/html/latest/networking/fib_trie.html]. Finally, file `/proc/net/fib_trie` visualizes the *trie* structures of the routing tables on your system and file `/proc/net/fib_triestat` shows some statistics and counters.

fib_lookup()

This function represents the next higher layer of abstraction on the way up the call stack. As you can see in Figure 5, it is also being fed with an instance of `struct flowi4 [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/flow.h#L69]` and spits out an instance of `struct fib_result [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L165]`. Those actually are the very same instances explained above. Their pointers are simply forwarded to the function `fib_table_lookup()` when calling it. There are two implementations of this function: `fib_lookup() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L306]`^{1} and `fib_lookup() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L364]`^{2}. Kernel build-time config `IP_MULTIPLE_TABLES [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/Kconfig#L63]` decides which implementation is used. If it is set to `n`, then only one routing table named *main* exists on the system and function `fib_lookup() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L306]`^{1} is used, which doesn't do much more than simply calling `fib_table_lookup()` for that table[6]. If it is set to `y`, which is a common setting for kernels in most modern Linux distributions, then multiple routing tables exist in your system. By default, the kernel here creates three tables named *local*, *main* and *default*. Function `fib_lookup() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L364]`^{2} is used here and implements the concept of *Policy-based routing (PBR) [https://en.wikipedia.org/wiki/Policy-based_routing]*. Let's do a short recap on what that means: A so-called *Routing policy database (RPDB)* is implemented, which is a set of rules to be evaluated before the actual routing lookup. These rules specify which routing table(s) to consult in which case. This way, more packet characteristics than just the destination IP address can be taken into account when doing the routing lookup. These can be things like the source IP address, `skb->mark`, TCP/UDP source/destination port, and so on. This is why `struct flowi4` has several member variables. While the lower layer function `fib_table_lookup()` mostly only cares about member `daddr`, the *PBR* implementation in function `fib_lookup()`, depending on the set of rules, makes use of the other ones. For example, a rule can specify to consult a different routing table for packets with a specific source IP address. You can use command `ip rule` to list/add/delete rules in the *RPDB*. Consult the man-page [https://manpages.debian.org/bullseye/iproute2/ip-rule.8.en.html] of that command for details on rule syntax and semantics. The following shows the default set of rules in the *RPDB*:

```
# PRIORITY:  SELECTOR   ACTION
0         :  from all   lookup local
32766     :  from all   lookup main
32767     :  from all   lookup default
```

Each rule consist of three parts: *PRIORITY*, *SELECTOR* and *ACTION*. Integer *PRIORITY* specifies the order in which the rules are evaluated during lookup. *SELECTOR* specifies for which packets the *ACTION* of the rule shall be executed. In the default case shown above, `from all`, means that *ACTION* shall be executed for packets containing any possible source IP address (= for all packets). In case you are adding your own non-default rules to the *RPDB*, then *SELECTOR* is the part of a rule where you can specify filters based on additional characteristics of the network packet and thereby e.g. create a rule which specifies that its *ACTION* shall only be executed for network packets with source IP address `10.0.0.1`. Part *ACTION* specifies which table is to be consulted; thus, `lookup local` means that the routing table named *local* shall be consulted. In summary, this default set of rules means that three routing tables named *local*, *main* and *default* are to be consulted and what `fib_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L364] does then is simply looping through these three tables and calling `fib_table_lookup()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_trie.c#L1432] for each one of them; see Figure 5. If a routing entry matches, then the lookup stops at this point and uses this match as the final result, meaning the potentially remaining table(s) are not consulted in that case.

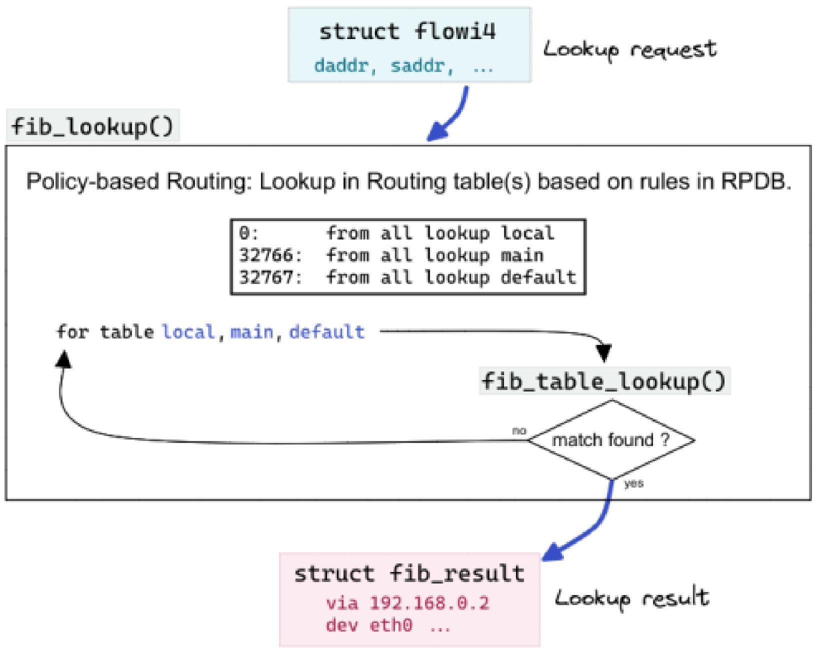

struct flowi4
daddr, saddr, ...
Lookup request
fib_lookup()
Policy-based Routing: Lookup in Routing table(s) based on rules in RPDB.
0:      from all lookup local
32766: from all lookup main
32767: from all lookup default
for table local, main, default
fib_table_lookup()
match found ?
no
yes
struct fib_result
via 192.168.0.2
dev eth0 ...
Lookup result

Figure 5: The Policy-based Routing lookup based on *default* set of rules in *RPDB*.

Remember, that a routing entry with *prefix* `0.0.0.0/0` (*default* route) would represent a "catch all" match and in this setup it thereby would prevent any remaining routing table(s) to ever be consulted. So, what's the purpose of these three routing tables *local*, *main* and *default*? Table *local* is the first one to be consulted. All its routing entries are created automatically by the kernel and there intentionally never is a "catch all" *default* route present, so the following table *main* will still be consulted, if there is no match in *local*. As mentioned above, the routing lookup, among other things, is used to determine which network packets are meant for local delivery on this system. Let's say you are assigning address `192.168.2.100/24` to your local network interface `eth0`. The kernel then not only assigns this address, but also adds routing entries to table *local*[1]:

```
broadcast 192.168.2.0   dev eth0 proto kernel scope link src 192.168.2.100
local     192.168.2.100 dev eth0 proto kernel scope host src 192.168.2.100
broadcast 192.168.2.255 dev eth0 proto kernel scope link src 192.168.2.100
```

The first column specifies the *type* of the routing entry, which is either *local* or *broadcast* for the entries added to table *local*. This is the most important field of a routing entry, because it specifies what to do with a matching packet. *Type=local* means that the packet is meant for local delivery. *Type=broadcast* means the same for packets with the specified *broadcast* destination addresses. Field `proto …` specifies who created the routing entry. Thus, `proto kernel` means that these entries have been created automatically by the kernel. See man-page [https://manpages.debian.org/bullseye/iproute2/ip-route.8.en.html] of command `ip route` for more details. The second table to be consulted is *main*. This is the "normal" routing table where all the routing entries which you create with command `ip route` go, if you do not explicitly specify otherwise. The kernel also auto-creates an entry here, when you assign IP address `192.168.2.100/24` to `eth0`:

```
unicast 192.168.2.0/24 dev eth0 proto kernel scope link src 192.168.2.100
```

Type=unicast here means that a matching packet is to be forwarded and sent out on `eth0` as a normal *unicast* packet and that the next hop is located in the local broadcast domain of `eth0` (`scope link`) and that it is the endpoint and not a next hop gateway. The third table to be consulted is *default*. This table is usually empty and seems it merely still exists for historic reasons.

The kernel itself does not care or know about routing table *names*. It identifies each routing table with a *u32 tb_id;* *[https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L250]* integer. The integer values identifying the default tables *local* (255), *main* (254) and *default* (253) are hard-coded. Iproute2 commands like *ip rule* and *ip route* can work with *names* as well as integers. File */etc/iproute2/rt_tables* serves as mapping between both.

Don't get confused by this highly optimized code. The source code of fib_lookup() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L364][2] might give you the impression, that in the default case just tables *main* and *default* are consulted, but table *local* is omitted. That is actually not the case. There are 2 possible ways this works: In case you already modified [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L371] the set of rules in the *RPDB*, then the whole rule evaluation and subsequent calls to *fib_table_lookup()* are all performed within __fib_lookup() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_rules.c#L80] and functions below. However, in case the *RPDB* still contains the default set of rules, then fib_lookup() simply directly calls fib_table_lookup() two times… first for table *main* and then for table *default*. This is done as a runtime optimization. But where is the lookup into table *local*? This commit message [https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0ddcf43d5d4a03ded1ee3f6b3b72a0cbed4e90b1] answers that. In case of an unmodified set of rules, the routing entries of table *local* are simply merged into the FIB trie of table *main*. This stays invisible to the user, as commands *ip route show table main* and *ip route show table local* will still give you the fake impression that those tables exist separately. However, a look at file */proc/net/fib_trie* can confirm that both tables are actually merged. It will still show you both tables *local* and *main*, but if you look closely, you'll see that both FIB tries are identical and both contain all entries of *local* and *main* together.

The surrounding functions

Now let's move up the call stack once more. The function on the packet receive path calling `fib_lookup()` is `ip_route_input_slow()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L2223]` and on the local output path it is `ip_route_output_key_hash_rcu()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L2652]`. Please take another look at Figure 3. What I am about to describe here involves several more layers up the call stack and a whole bunch of helper functions. What happens on the receive path compared to the local output path is not identical, but quite similar. The code performs several checks on the destination IP address of the network packet to identify and sort out potential martian addresses and so on. A distinction is done to handle multicast packets. An instance of `struct flowi4` is prepared, `fib_lookup()` is called and spits out an instance of `struct fib_result`. The code on the receive path, depending on sysctl setting, further performs *Reverse Path Filtering* (explanation in box below). Based on the `fib_result`, an object representing the "routing decision" is allocated, initialized and finally attached to the network packet.

Reverse Path Filtering (RPF):
This feature is specified in RFC3704 [https://datatracker.ietf.org/doc/html/rfc3704] and is also referred to as *Reverse Path Forwarding* or *Route Verification*. It is meant as countermeasure against potential IP address spoofing which is often done in the scope of DDoS attacks. It cannot fully prevent spoofing, but it can limit it. *RPF* is implemented in form of function fib_validate_source() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_frontend.c#L421] which is called on the packet receive path shortly after *fib_lookup()*; see Figure 3. This feature, if fully activated, performs yet another routing lookup, but this time for the source IP address of the network packet. The general idea is to check whether a route exists for a potential reply packet and whether that route would actually go out on the network interface where the original packet has been received. It not, the packet would be dropped. See Figure 6. *RPF* is switched off by default. It can be switched on via sysctl [https://www.kernel.org/doc/html/latest/networking/ip-sysctl.html]. It will automatically activate in case you got non-default rules in your *RPDB*.

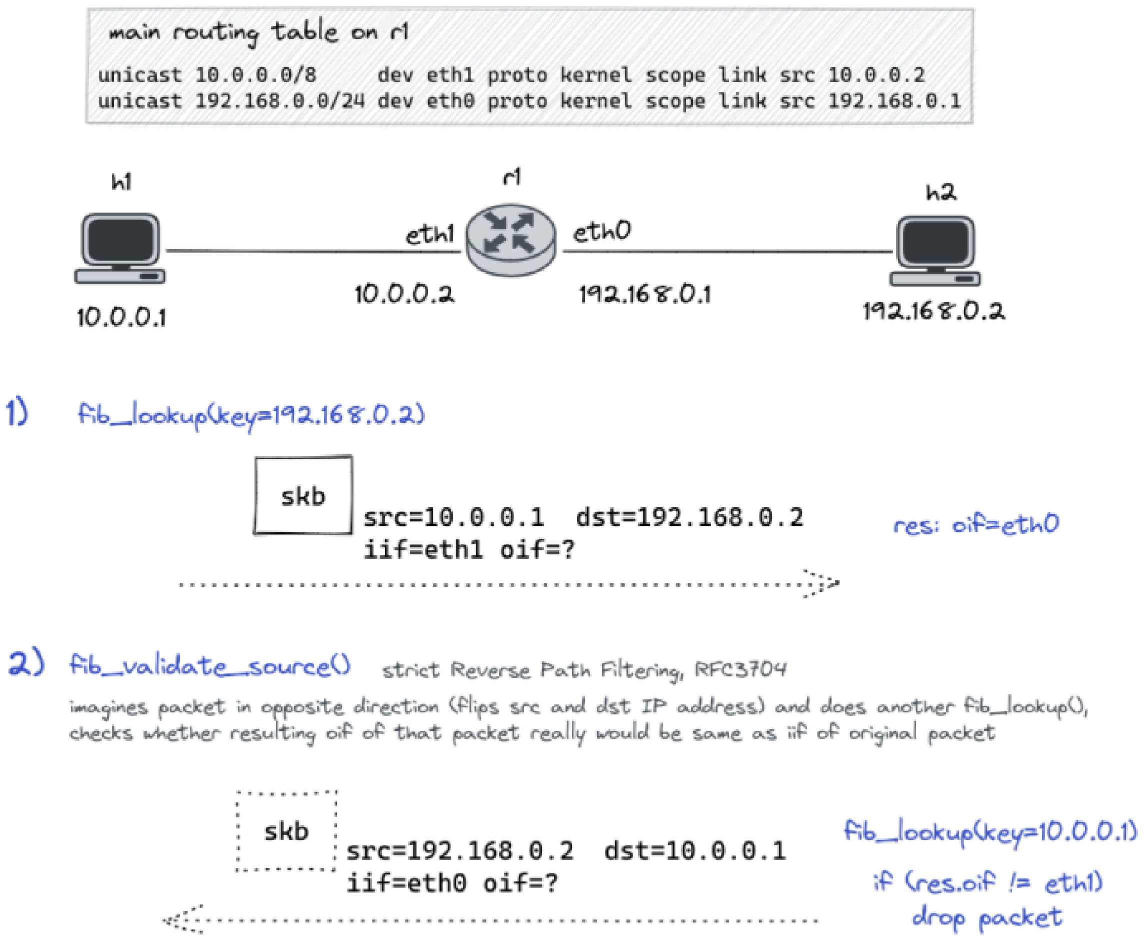
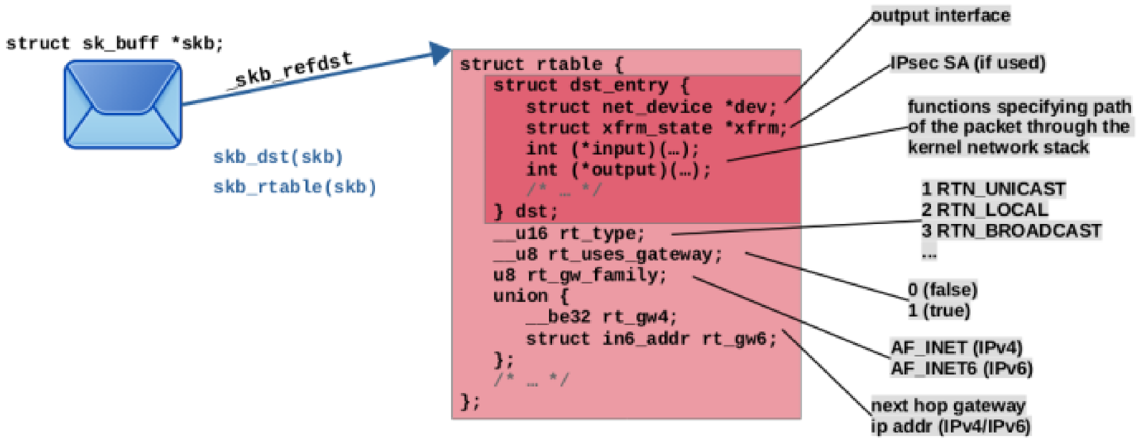

main routing table on r1
unicast 10.0.0.0/8   dev eth1 proto kernel scope link src 10.0.0.2
unicast 192.168.0.0/24 dev eth0 proto kernel scope link src 192.168.0.1
h1
r1
h2
eth1
eth0
10.0.0.1
10.0.0.2
192.168.0.1
192.168.0.2
1) fib_lookup(key=192.168.0.2)
skb
src=10.0.0.1 dst=192.168.0.2
iif=eth1 oif=?
res: oif=eth0
2) fib_validate_source() strict Reverse Path Filtering, RFC3704
imagines packet in opposite direction (flips src and dst IP address) and does another fib_lookup(), checks whether resulting oif of that packet really would be same as iif of original packet
skb
src=192.168.0.2 dst=10.0.0.1
iif=eth0 oif=?
fib_lookup(key=10.0.0.1)
if (res.oif != eth1)
drop packet

Figure 6: Example setup demonstrating *Reverse Path Filtering* (click to enlarge).

The routing decision object

This object consists of two structs, the outer struct rtable [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L49] and the inner struct dst_entry [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L25]; see Figure 7. It is being allocated and partly initialized in function rt_dst_alloc() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1633]. Further member initialization happens after this function call. Alternatively, this object can also be taken from cache. Caching is a complex topic, which deserves its own article. I'll skip it for now, but will explain it in detail in the second article of this series. Attaching this object is done either by function skb_dst_set() [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/skbuff.h#L991] or skb_dst_set_noref() [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/skbuff.h#L1006][8]. The network packet provides member unsigned long _skb_refdst [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/skbuff.h#L759] as pointer to that attachment.



```
struct rtable {
    struct dst_entry {
        struct net_device *dev;
        struct xfrm_state *xfrm;
        int (*input)(…);
        int (*output)(…);
        /* … */
    } dst;
    __u16 rt_type;
    __u8 rt_uses_gateway;
    u8 rt_gw_family;
    union {
        __be32 rt_gw4;
        struct in6_addr rt_gw6;
    };
    /* … */
};
```

Figure 7: Routing decision object attached to a network packet, showing most relevant member variables (click to enlarge).

struct rtable

`struct rtable` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L49] is the outer struct of the routing decision object attached to a network packet. Variables of this type are often named `rt`. Most relevant member variables are:

- `struct dst_entry dst;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L50]
  Details in the following section. First member variable; thus, its memory address is identical to the address of the instance of `struct rtable` containing it. The kernel code provides the two comfort functions `skb_rtable(skb)` [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/skbuff.h#L1025] and `skb_dst(skb)` [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/skbuff.h#L972], which return a correctly casted pointer to either one of these two structs attached to a given skb; see Figure 7. Those functions are used in countless places throughout the kernel network stack.

- `__u16 rt_type;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L54]
  This member contains the *type* of the matching routing entry, which led to the creation of this routing decision object. The *type* of a routing entry determines how to handle a network packet. All possible values are defined in this unnamed enum [https://elixir.bootlin.com/linux/v5.14.7/source/include/uapi/linux/rtnetlink.h#L244]. However, only a subset of these values will end up being used in `rt_type`. The following table lists those values and their meaning.

enum	int	ip route[9]	meaning
RTN_UNICAST	1	unicast	(default type of a routing entry; used, if not explicitly specified otherwise) packet shall be forwarded, gateway or direct route
RTN_LOCAL	2	local	packet is for local host
RTN_BROADCAST	3	broadcast	accept locally as broadcast, send as broadcast
RTN_MULTICAST	5	multicast	multicast route
RTN_UNREACHABLE	7	unreachable	packet dropped and ICMP message *host unreachable* is generated

- `__u8 rt_is_input;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L55]
  Boolean variable. It is set to true (1), if the routing lookup and creation of this routing decision has been performed on the packet receive path. It is set to false (0), if same has been performed on the local output path. If this is checked later in other places of the kernel code, then one of the wrapper functions `rt_is_input_route()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L75] or `rt_is_output_route()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L80] is used to access this variable.

- `__u8 rt_uses_gateway;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L56]
  Boolean variable. It is set to true (1), if the next hop of the matching routing entry is a gateway (if the routing entry contained some statement like `via 10.0.0.1`). In this case, member `rt_gw4` contains the gateway IP address (see below). It is set to false (0), if the next hop is not a gateway. It is the final destination host.

- `u8 rt_gw_family;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L60]
  Set to 0, if there is no gateway. Set to `AF_INET` [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/socket.h#L179] (2), if gateway IP address is IPv4. Set to `AF_INET6` [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/socket.h#L187] (10), if gateway IP address is IPv6.

- `union { __be32 rt_gw4; struct in6_addr rt_gw6; };` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L62]
  If there is a gateway, its IP address is stored here. `rt_gw_family` specifies, whether it is an IPv4 or IPv6 address and thereby, whether member `rt_gw4` or `rt_gw6` of this union is being used.

struct dst_entry

`struct dst_entry` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L25] is the inner struct of the routing decision object attached to a network packet. It is sometimes referred to as the *destination cache*. Variables of this type are often named `dst`. Most relevant member variables are:

- `struct net_device *dev;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L26]
  Pointer to the output network interface, on which this packet shall be sent out (relevant of forwarded and outgoing packets). This has been specified by the matching routing entry.

- `struct xfrm_state *xfrm;` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L31]
  Reference to an IPsec SA. In a normal situation this is *NULL*. However, in case you are using IPsec and a lookup into the IPsec SPD (Security Policy Database) has been performed, then, in case this network packet matches an IPsec policy, the routing decision object is replaced by a whole "bundle" of structs referencing each other. In that case, member `xfrm` points to the IPsec SA which is to be applied to this packet. You'll find details on that in my blog article Nftables - Netfilter and VPN/IPsec packet flow.

- `int (*input)(struct sk_buff *skb);` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L35]
  Based on the result of the routing lookup, the *type* of the route and several other factors, this is set to point to one of several possible functions. At a later point, it will be dereferenced and the function will be called, taking the *skb* of the packet as parameter. This will thereby determine the path the network packet takes through the network stack. It represents a demultiplexing mechanism, which directs the packet to e.g. local delivery, to be forwarded, to error handling, or other possible paths… The following table lists some functions this member can point to (not a complete list):

`dst_discard()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L381]	Default value; set when instance of this struct is first created and initialized; see function `dst_init()`

	`[https://elixir.bootlin.com/linux/v5.14.7/source/net/core/dst.c#L60]`.
`ip_local_deliver()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L240]`	For received *unicast* and *broadcast* packets destined to local host; this means in case matching routing entry of type *local* or *broadcast* has been found, but also in case daddr of packet is `255.255.255.255` (in latter case routing lookup is omitted).
`ip_forward()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_forward.c#L86]`	For received *unicast* packets that shall be forwarded.
`ip_mr_input()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ipmr.c#L2071]`	For received multicast packets (under some conditions).
`ip_error()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L937]`	E.g. set, if no route has been found; thus, for type *unreachable*. Packet is dropped and ICMP *host unreachable* message is generated.
`lwtunnel_input()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/core/lwtunnel.c#L383]`	In case of light weight tunnels like MPLS are to be used on this packet; see `CONFIG_LWTUNNEL` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/Kconfig#L390]`.

- `int (*output)(struct net *net, struct sock *sk, struct sk_buff *skb);` `[https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L36]`
  Based on the result of the routing lookup, the *type* of the route and several other factors, this is set to point to one of several possible functions. At a later point, it will be dereferenced and the function will be called, taking the *skb* of the packet as parameter. This will thereby determine the path the network packet takes through the network stack. It represents a demultiplexing mechanism, which directs the packet to e.g. be sent out as a normal *unicast* packet, to be output handled by the Xfrm framework (= to get IPsec encrypted), or other possible output paths… The following table lists some functions this member can point to (not a complete list):

`dst_discard_out()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/core/dst.c#L30]`	Default value; set when instance of this struct is first created and initialized; see function `dst_init()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/core/dst.c#L60]`.
`ip_output()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L423]`	For packets on the local output path, to be sent out as *unicast* packets.
`ip_rt_bug()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1246]`	For received *unicast* packets destined to local host; bug handling, should never be called.
`xfrm4_output()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/xfrm4_output.c#L31]`	For forwarded packets or packets on the local output path, which yet need to be transformed by Xfrm (get IPsec encrypted). See my other article Nftables - Netfilter and VPN/IPsec packet flow for details.
`ip_mc_output()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L361]`	For packets on the local output path to be sent out as *multicast* packets.

Packet Flow

Figure 8 visualizes the flow of IPv4 *unicast* packets with focus on network layer and on the attached routing decision. In general there are a lot of places in the network stack where packets are being demultiplexed based on certain criteria and where that is implemented based indirect function calls or, in other words, based on C function pointers. There are two demultiplexing points on the network layer which operate based on the routing decision object attached each network packet. They are shown in green color in Figure 8. For completeness, the figure further shows two more demultiplexing points for received packets. The first is used to distinguish between different network layer protocols like IPv4, IPv6, ARP, … and the other to distinguish between different transport layer protocols like TCP, UDP and so on.

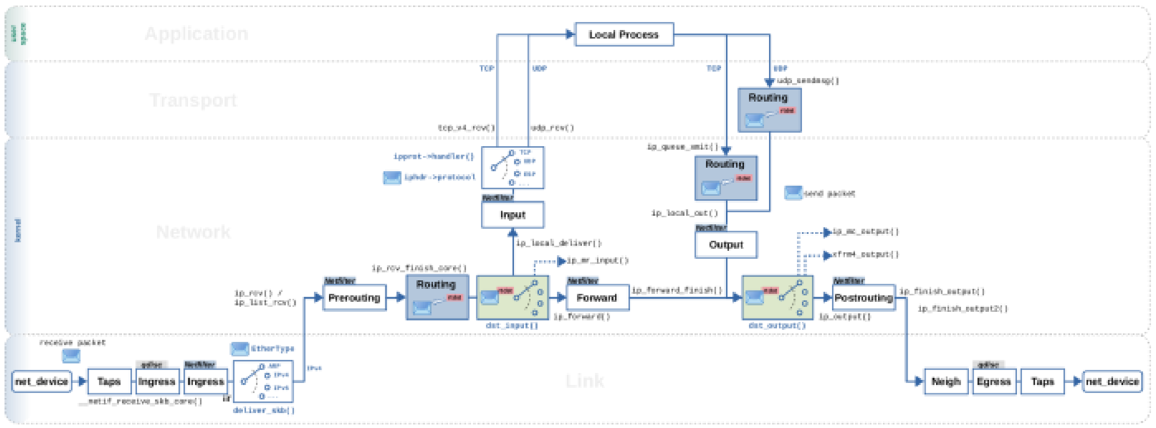


Figure 8: Packet flow, focus on *unicast* IPv4 packets on network layer, packet demultiplexing based on attached routing decision object *rtdst* (click to enlarge).

Let's take a look at what this means for *unicast* IPv4 packets, which are received in form of Ethernet frames on link layer. Thus, we are starting in the lower left corner of Figure 8. The first thing you need to know here is that packets can cross the line from link layer to network layer and undergo several of the initial steps of network layer handling either as individual packets (one by one) or as lists of packets. Thereby, to cross the line from link layer to network layer, either function `deliver_skb()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/dev.c#L2332] or its list-based alternative `__netif_receive_skb_list_ptype()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/dev.c#L5530] is called. Either one here performs an indirect call to a registered network protocol receive handler function. Network layer protocol implementations like IPv4 register their receive handler functions to be called here for incoming packets. Protocol distinction is done by means of a hash table lookup, based on the EtherType [https://en.wikipedia.org/wiki/EtherType] of the frame. In case of IPv4, function `ip_rcv()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L531] or its list-based counterpart `ip_list_rcv()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L613] is called. These functions represent two parallel alternative paths for IPv4 packets through the network layer. However, semantically both do the same to each packet, the only difference being that `ip_list_rcv()` handles lists of packets while `ip_rcv()` handles a single packet. Once on network layer, the received packets traverse the Netfilter *Prerouting* hook. If they are not dropped here, then function `ip_rcv_finish()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L415] or its list-based counterpart `ip_list_rcv_finish()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L564] is called. Among other things, both variants call `ip_rcv_finish_core()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L315]. This function has no list-based counterpart. It handles individual packets. As you might recall from the sections above, this is where the routing lookup happens; see again Figure 3. There are several optimizations and caching mechanisms in place here, which influence the way the routing lookup is performed or even skip it completely in certain cases. I'll describe those cases in the second article of the series. No matter in which exact variant the lookup is performed here, the end result is always the same: A routing decision object, shown as `rtdst` in Figure 8, is being attached to the packet (skb). Shortly after that, function `dst_input()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L456], shown in green color in Figure 8, is called, This function does packet demultiplexing by performing an indirect call of member `(*input)()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L35] of the attached `struct dst_entry` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L25] instance[10]. There is no list-based handling here. Each call handles a single packet (skb), as you can see in the function parameter list. For a packet destined to be locally received on the system, `dst_input()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L456] calls function `ip_local_deliver()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L240], which handles IP fragmentation, leads the packet to traversal of the Netfilter *Input* hook and ultimately to a transport layer protocol handler, to be potentially received by a listening socket. As you can see, transport protocols like TCP and UDP also register their receive handler functions, so there is yet another demultiplexing step happening here, based on indirect function calls. For a packet to be forwarded, `dst_input()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L456] calls function `ip_forward()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_forward.c#L86], which leads the packet to traversal of the Netfilter *Forward* hook and finally calls `ip_forward_finish()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_forward.c#L65][11]. This function ends in calling `dst_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L446], which is the second point of demultiplexing based on the routing decision, shown in green color in Figure 8. It does an indirect call on member `(*output)()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L36] of the attached `struct dst_entry` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L25] instance. In the most common case `dst_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L446] thereby calls function `ip_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L423], which leads the packet to traversal of the Netfilter *Postrouting* hook and finally calls `ip_finish_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L311]. What follows is a call to `__ip_finish_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L290] and then `ip_finish_output2()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L187], which hands the packet over to the *neighbouring subsystem* and thereby finally to link layer, so that it can be sent out on the network. Based on the attached routing decision, `dst_output()` can alternatively call other functions like e.g. `xfrm4_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/xfrm4_output.c#L31] which puts the packet on a path to be encrypted via means of IPsec or function `ip_mc_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L361], which is used in case of *multicast* packets.

Now, let's take a look at packets which are to be sent out, starting from a local TCP or UDP socket on the system. Depending on the socket/protocol state, different functions might be involved here. However, commonly function `ip_queue_xmit()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L546] is called here at some point in case of TCP. It in turn calls `__ip_queue_xmit()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L455], which you might recall from the sections above; see again Figure 3. This is where the routing lookup is performed. In case of UDP, this is commonly done in function `udp_sendmsg()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/udp.c#L1040]. There are optimizations and caching in place here. The routing decision is e.g. being cached within the socket, so that the routing lookup doesn't have to be repeated for every packet sent out by the same socket (more on that in the next article). Nevertheless, the end result is the same: The routing decision object is being attached to the packet (skb). Shortly after that, function `ip_local_out()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L120] is called, which in turn calls `__ip_local_out()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L99], which lets the packet traverse the Netfilter *Output* hook. If the packet is not dropped here, then function `dst_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L446] is being called. Thus, we again reached the point of packet demultiplexing based on the routing decision object. This is where the paths of forwarded packets and packets on the local output path merge in Figure 8.

In case you like to deepen your understanding of the routing subsystem and the packet flow, then I recommend the book Linux Kernel Networking: Implementation and Theory [https://ramirose.wixsite.com/ramirosen], published 2014 by Rami Rosen, Apress. This book, in combination with reading source code (there's no way around that), has been my starting point on this topic and it has been very helpful. This article is not suited to replace reading source code and books like that. It is merely meant as an addendum to shed a little more light on the functions `dst_input()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L456] and `dst_output()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L446] and their background.

A comment on the mentioned list-based handling of received packets: Seems most of that has been implemented with this patchset [https://lore.kernel.org/netdev/bec738bd-c3fa-ba54-9473-84c8366c5699@solarflare.com/] 12). It is a runtime optimization, which among other things aims at reducing the number of indirect calls. The idea is that only one indirect call, e.g. to *ip_list_rcv()*, is required for a whole list of packets, instead of one indirect call to *ip_rcv()* for each individual packet.

Network namespace

Routing tables and the set of rules of the *RPDB* are all stored locally within a *network namespace* instance. In other words, each *network namespace* has its own individual set of routing tables and *RPDB* rules, same as it is with other components of the network stack like network interfaces, sockets, Netfilter hooks and so on.

Context

The described behavior and implementation has been observed on a Debian 11 (bullseye) system with Debian *backports* on *amd64* architecture, using Kernel version 5.14.7.

Feedback

Feedback to this article is very welcome! Please be aware that I did not develop or contribute to any of the software components described here. I'm merely some developer who took a look at the source code and did some practical experimenting. If you find something which I might have misunderstood or described incorrectly here, then I would be very grateful, if you bring this to my attention and of course I'll then fix my content asap accordingly.

Continue with next article

Routing Decisions in the Linux Kernel - Part 2: Caching

published 2022-07-04, last modified 2022-10-30

1)
I describe those in my other article Nftables - Packet flow and Netfilter hooks in detail.
2)
Figure 1 intentionally shows the routing lookup on the borderline between transport layer and network layer. While the lookup itself is clearly part of network layer, depending on transport layer protocol and state, it can be called directly from transport layer code (UDP/TCP). Thus, it is debatable where to put it exactly.
3)
Network packets and their meta data are represented by instances of that data structure, `struct sk_buff` `[https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/skbuff.h#L720]`, within the kernel.
4)
I am showing the TCP established case here as an example. The lookup is done in similar ways, however within other functions, in case of UDP, ICMP protocols …
5)
the algorithm used for IPv6 is slightly different
6)
The entries of table *local* would in this case be merged into the *main* table (more on that below…).
7)
The following shows those entries in the output format of command `ip -d route show table local`.
8)
The difference between both of these functions is merely setting a flag which indicates reference counting, which is only done in certain cases here.
9)
The `ip route` command uses this syntax for specifying the *type*. When you e.g. use command `ip -d route` (`-d` for "details") to list the routing entries of your *main* table, then *type* is shown in the first column of each line of output.
10)
Do not let this highly optimized code confuse you. The function does exactly that. The macros you see in the code, like `INDIRECT_CALL_INET()` `[https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L458]`, are merely a runtime optimization, taking into account the branch prediction [https://en.wikipedia.org/wiki/Branch_predictor] feature of instruction pipelining CPUs.
11)
BTW: In case you are using IPsec, then before traversal of the Netfilter *Forward* hook, `ip_forward()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_forward.c#L86]` calls `xfrm4_policy_check()` `[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_forward.c#L104]` and `xfrm_route_forward()`

`[https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_forward.c#L121]`, which respectively handle the IPsec forward policy (`dir fwd` SP) and output policy (`dir out` SP) lookups. I describe that in detail in my other article Nftables - Netfilter and VPN/IPsec packet flow.

12) The commit message of the merge commit of this patchset contains a detailed description on why this has been implemented.

blog/linux/routing_decisions_in_the_linux_kernel_1_lookup_packet_flow.txt · Last modified: 2022-10-30 by Andrej Stender