Thermalcircle.de

climbing the thermals

linux, kernel, routing

Routing Decisions in the Linux Kernel - Part 2: Caching

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

In the previous article I talked a lot about the *routing decision*, represented by 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], which is attached to a network packet *skb* as result of the routing lookup. However, I intentionally left out one very essential topic… *caching*. When you look at how a packet is being handled as a result of its attached *routing decision*, then it is not relevant (at least not at first glance) to know where this attached decision actually came from. Maybe it had been allocated specifically for the network packet at hand or it originated from some kind of cache… it does not matter. The resulting packet handling is all the same. However, this does not make the topic *caching* less relevant. In fact it is very relevant, especially regarding performance, and due to its complex history, I found that it deserved its own article. You've probably heard the term *routing cache* here and there, right? An actual full-blown *routing cache* was present in older kernels and had been removed with v3.6. However, that does not mean that caching thereby vanished completely. Other caching and optimization mechanisms regarding routing decisions have been added to the kernel since that removal. Further, some already existing ones have been kept. The Internet is full of articles and documentation which mention some kind of *routing cache*. However, due to the vivid history, it is very easy to get confused by all this, partly contradicting, information. At least that was my problem at the beginning. This article is an attempt to get that information sorted and to put into context what has been the case in older kernels and what actually is the case in a modern kernel like v5.14.

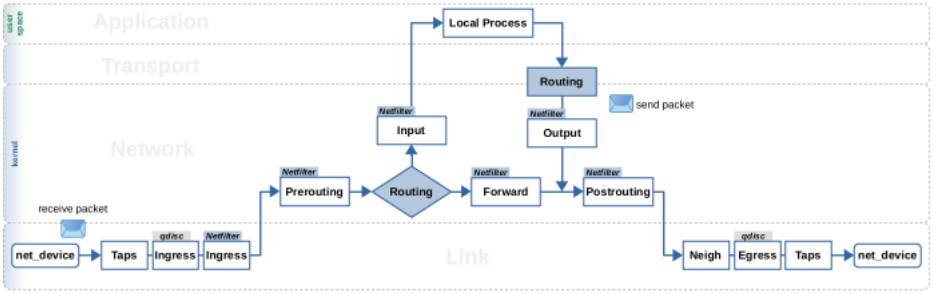

Figure 1: Simplified Overview: Routing lookup, packet flow and Netfilter hooks (click to enlarge)

There is one thing all those caching mechanisms, historic or not, got in common: They are all based on the same mentioned data structures. This is why `struct dst_entry` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L25] is also referred to as *destination cache* [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L3]. However, caching works a little differently when you compare the receive path with the local output path. This is why I show Figure 1 from the previous article here again. It gives a rough orientation on where the routing lookup is commonly performed on both paths. I strongly recommend that you read the previous article of this series first to fully understand what I explain in the sections to follow.

The "old" Routing Cache

Linux kernels prior to v3.6 contain a *routing cache*, which is consulted before the actual *routing lookup*. Only if the lookup into this cache does not produce a match, then the actual routing table lookup, including policy-based routing, is performed. The cache is implemented as global hash table `rt_hash_table` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/route.c#L299]. A lookup into this table is performed based on a *key* constructed from protocol and meta data of the network packet in question, like source and destination IP address and more. The details vary here a little on the receive path compared to the local output path. A positive cache match returns an instance of the mentioned routing decision, represented by the outer `struct rtable` and its inner `struct dst_entry`, which is then attached to the network packet. If the cache produces no match, then the normal routing lookup, represented by function `fib_lookup()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/route.c#L2731][1], is performed. In case that produces a matching route, a new instance of a routing decision is allocated and initialized based on that result, then added to the cache and finally attached to the packet. The *routing cache* implementation comes with a garbage collector, which is executed based on a periodic timer and is also triggered in case of certain events (e.g. when the number of cache entries exceeds a certain threshold).

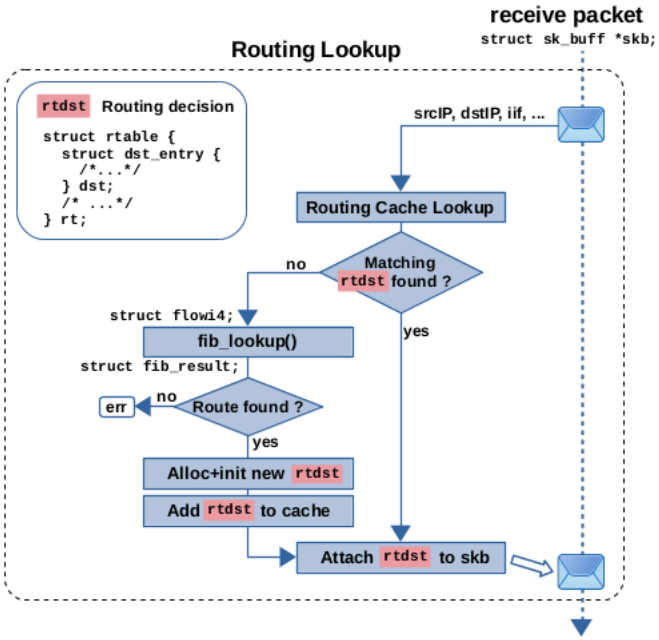

Figure 2: Routing Cache Lookup and routing table lookup on the receive path in kernels prior to v3.6.

Let's take a look at the source code of kernel v3.5, the final release version before the cache got removed: Figure 2 illustrates the behavior on the receive path. There, function `ip_rcv_finish()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/ip_input.c#L317] calls `ip_route_input_noref()` [https://elixir.bootlin.com/linux/v3.5/source/include/net/route.h#L178], which in turn calls `ip_route_input_common()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/route.c#L2427]. This is where the cache lookup is implemented. On the receive path, the key used for cache lookup is based on source and destination IP address of the packet, its ingress network interface index, IPv4 TOS field, `skb->mark` and the *network namespace*. In case of a cache hit, the cached routing decision object is attached to the network packet and all is done. In case of a cache miss, function `ip_route_input_slow()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/route.c#L2246] is called. It calls `fib_lookup()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/fib_rules.c#L57] and based on the result then allocates and initializes a routing decision object. This object is added to the routing cache and then attached the network packet. Now you know, why function `ip_route_input_slow()` is named "slow". It is only called in case of a cache miss and executes the full-blown routing table lookup including policy-based routing. In newer kernels, with the cache removed, this function is actually always called, however its name hasn't been changed.

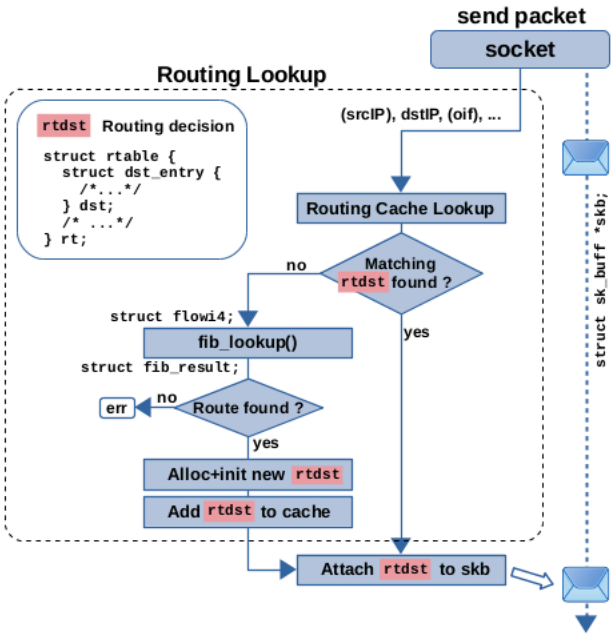

Figure 3: Routing Cache Lookup and routing table lookup on the local output path in kernels prior to v3.6.

Now let's take a look at the local output path in kernel v3.5, illustrated in Figure 3. The cache and routing lookup nearly work the same way than on the receive path; however, there are some minor differences. Like in the previous article, I again take function `ip_queue_xmit()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/ip_output.c#L335] as an example, which is used when a TCP socket likes to send data on the network. It calls `ip_route_output_ports()` [https://elixir.bootlin.com/linux/v3.5/source/include/net/route.h#L140], which in turn calls `ip_route_output_flow()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/route.c#L2931], which in turn calls `__ip_route_output_key()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/route.c#L2809]. This is where the cache lookup is implemented. The lookup key is generated here based on data provided by the sending socket, like source and destination IP address, egress network interface index, IPv4 TOS field, `skb->mark` and the *network namespace*. Obviously locally generated packets which are to be sent out do not possess an ingress network interface. Thus, instead the egress network interface is being used here. But wait a minute. Isn't the egress network interface actually being determined by the routing lookup? So, how can it be an input parameter for the routing cache lookup? Same goes for the source IP address, which would be implicitly determined once the egress interface got determined. Well, both parameters can be predetermined in case the socket which sends this packet is bound to a specific network interface or to an IP address which is assigned to one of the network interfaces on this system. In all other cases it is of course the routing lookup itself which determines these parameters. Ok, let's get back to packet processing: In case of a cache miss, function `ip_route_output_slow()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/route.c#L2616] is called, which in turn calls `fib_lookup()`, same as on the receive path. Based on its result then a routing decision object is allocated and initialized and then added to the routing cache. However, no matter whether the routing decision object had just been allocated or came from the routing cache, its attachment to the network packet is actually not handled as part of this whole routing lookup. That happens a few function call layers up in function `ip_queue_xmit()` [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/ip_output.c#L380]. This might seem like an irrelevant detail right now, but it will become relevant later in the sections below, you'll see.

So, why has the *routing cache* been removed with v3.6? Its problem was, that it was susceptible to DoS attacks. As described, it cached the routing decision for every individual flow; thus, roughly for every source+destination IP address pair. As a consequence, you could easily fill the cache with entries basically by sending packets to random destinations. Removal of the cache of course came with a performance penalty, because now the full-blown routing lookup, represented by `fib_lookup()`, would have to be executed each time instead. However, a few kernel versions before the removal, the so-called *FIB trie* algorithm[2], became the default routing table lookup algorithm used in the kernel. That happened with kernel v2.6.39 [https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3630b7c050d9c3564f143d595339fc06b888d6f3]. That algorithm provided a faster lookup, especially for routing tables with lots of entries, and thereby made removal of the *routing cache* feasible. The following references provide further details on the *routing cache* and its removal:

- Tuning Linux IPv4 route cache (Vincent Bernat, 2011) [https://vincent.bernat.ch/en/blog/2011-ipv4-route-cache-linux]
- Removing The Linux Routing Cache (David S. Miller, 2012) [http://vger.kernel.org/~davem/columbia2012.pdf]
- Routing cache is dead, now what ? (David S. Miller, 2013) [https://home.regit.org/2013/03/david-miller-routing-cache-is-dead-now-what/]

FIB Nexthop Caching

The removal of the *routing cache* does not mean that no caching of routing decisions happens at all in modern kernels. On the contrary, there are several mechanisms in place here. The one I describe in this section can probably be referred to as caching in the *FIB Nexthop* entry. What does this mean? First of all, this mechanism is not meant to replace the normal routing lookup. That lookup happens in the way I described in the previous article. However, there is no need for a new instance of a *routing decision* to be allocated and initialized after each routing lookup. Roughly spoken, if two routing lookups both produce the same result; thus, in case the same routing entry (route) is chosen in both cases, then this also means that the *routing decision* object will have the same content. So this mechanism, roughly spoken, caches one *routing decision* object for each entry in your routing table. A new object is only allocated and initialized at the very first time a certain routing entry is being used. Once that entry is used again, the cached version of this routing decision is used and attached to the network packet. I intentionally said "roughly spoken", because I simplified things a little, to explain the basic principle. For example, not just one, but actually two *routing decision* objects need to be cached, because you need to distinguish between the receive path and the local output path; I'll get to that. Let's dive in a little bit deeper: If you look at how the entries of a routing table are saved, then it is not as simple as there being a single data structure whose instances are used to hold routing entries in memory. The routing lookup works based on the *FIB trie* algorithm and is highly optimized for speed and economic memory consumption[3]. In addition to the *LC-trie* itself, several data structures like `struct fib_info` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L131], `struct fib_alias` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_lookup.h#L10], `struct fib_nh` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L103] and `struct fib_nh_common` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L80] are involved here. If you issue command `ip route` to list the entries of your routing table, the content of each routing entry (each line) is actually collected from several instances of these data structures. The only data structure whose instances seem to have a 1:1 relation to routing entries is `struct fib_alias` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/fib_lookup.h#L10][4]. All that data is saved in a very memory efficient way, so if e.g. part of the content of two routing entries happens to be identical, then it is probably saved only once and not twice. Things are a little complex here. What is relevant for caching is `struct fib_nh` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L103] and its member `struct fib_nh_common` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L80]. Here *nh* stands for *nexthop*. Thus, instances of these data structures hold the part of routing entries, which describes the next hop, like `via 192.168.0.3 dev eth0`. In Figure 4 I show the entries of an example routing table in the syntax of command `ip route`. The parts of the entries which belong to the *nexthop* are shown in red color.

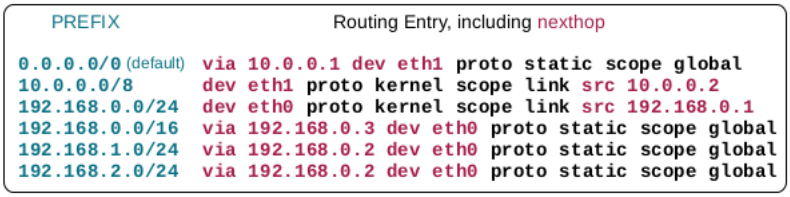

Figure 4: Example routing table; *nexthop* data shown in red color.

However, as I said, there is not necessarily a 1:1 relation between routing entries and instances of these *nexthop* data structures. Several routing entries can have the same *nexthop* data and thereby this then might be saved in just one instance. On the other hand, in case of *multipath routing*, a single routing entry can have more than one *nexthop* instance. Nevertheless, the data structure `struct fib_nh_common` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L80] has two member variables for holding cached instances of a *routing decision*; thus, pointers to type `struct rtable`; see Figure 5. Those are `nhc_pcpu_rth_output` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L98] and `nhc_rth_input` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L99]. The former one is per-cpu data and is used in case of routing lookups on the local output path. The latter is just a single pointer and is used in case of routing lookups on the receive path. It is necessary to have separate caches for the receive path and for the local output path, because, even if a routing lookup has the same result, the resulting *routing decision* contains those function pointers `(*input)()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L35] and `(*output)()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L36], which determine the path the network packet takes through the network stack, remember? As I described in the previous article, those point to different functions on the receive path compared to the local output path.

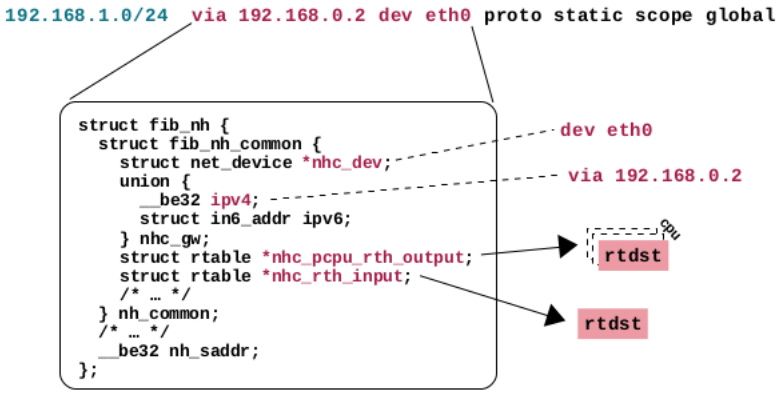

Figure 5: *Nexthop* data of a routing entry, as it is stored in `struct fib_nh_common` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L80]. Further member variables are used for caching *routing decisions* on the receive path and the local output path.

So how is this used? Let's say a routing lookup is performed for a network packet on the receive path. This is illustrated in Figure 6. Function `ip_route_input_slow()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L2223] is called. It prepares the `flowi4` request and calls `fib_lookup()`, which returns the routing lookup result in form of the usual `struct fib_result` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L165]. This struct contains pointers to all parts of the matching routing entry, including a pointer to an instance of `struct fib_nh_common` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L80]. If the matching routing entry here matches for the very first time, then the pointer of member `nhc_rth_input` is still `NULL` and so a new *routing decision* object needs to be allocated and initialized. Once this is done, `nhc_rth_input` is set to point to this new object, thereby caching it. Then the object is being attached to the network packet. Thus, when the same routing entry matches for yet another lookup of yet another network packet, then the routing decision object cached in `nhc_rth_input` is used and attached to that packet and no new object needs to be allocated. This is done here [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L2358] for locally received packets and here [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1854] for forwarded packets.

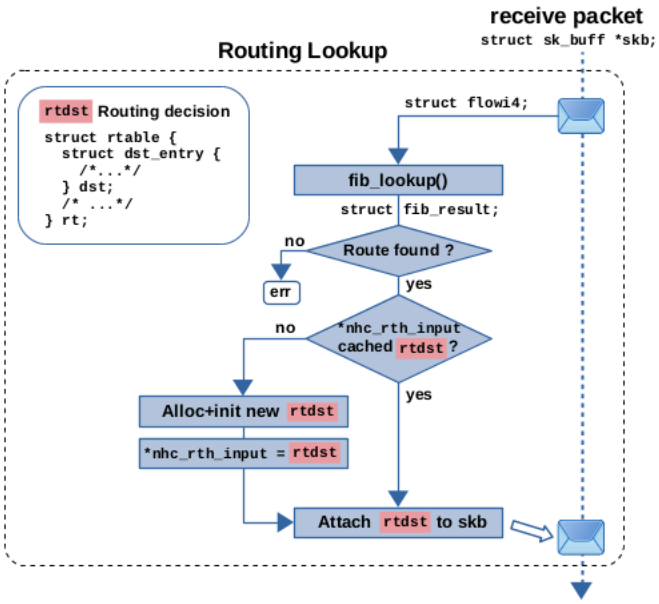

Figure 6: Routing table lookup and *nexthop* caching on the receive path.

Now let's take a look at the routing lookup on the local output path. This is illustrated in Figure 7. In case of TCP, function `__ip_queue_xmit()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L455] is called. It calls `ip_route_output_ports()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L157] to do the routing lookup. 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 roughly does the same things as `ip_route_input_slow()` does for the receive path. The only things which are different here are, that `struct fib_nh_common` member `*nhc_pcpu_rth_output` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L2582] is used for caching and that the actual attachment of the routing decision object to the network packet happens inside the calling function `__ip_queue_xmit()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L501] and not within the routing lookup handling functions.

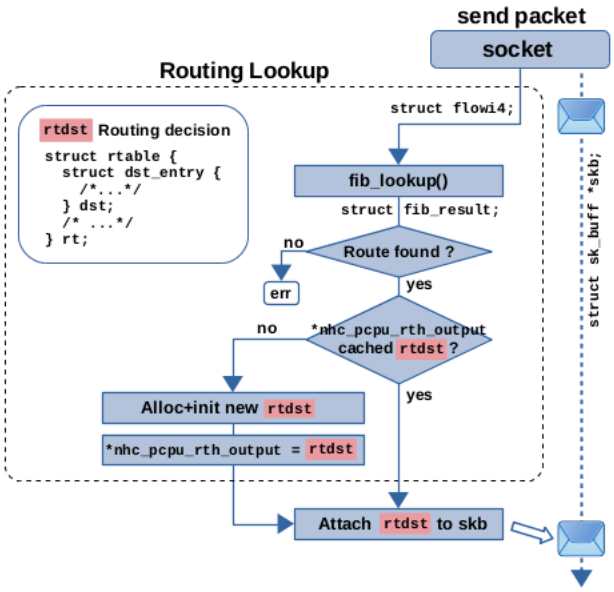

Figure 7: Routing table lookup and *nexthop* caching on the local output path.

FIB Nexthop Exception Caching

This mechanism is actually very similar to the *nexthop* caching described above. It actually co-exists with that mechanism in modern kernels; however, it is only used in case of *nexthop exceptions*. What are those? Well, routing entries can be changed not only by the user, but also by mechanisms like ICMPv4 Redirect messages or Path MTU discovery.

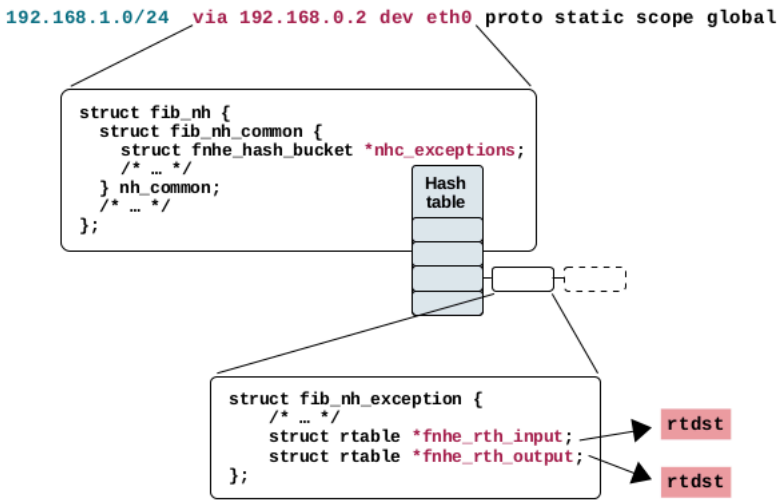

Figure 8: *Nexthop exceptions* are stored in hash table member of a *nexthop* instance. They possess member variables for caching *routing decisions* on the receive path and the local output path.

Changes like that do not overwrite an existing routing entry. Instead, as illustrated in Figure 8, `struct fib_nh_common` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L80] among other things contains a member named `*nhc_exceptions` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L100] which in those cases holds a hash lookup table, where *exceptions* to that routing entry can be stored. Those are represented by instances of `struct fib_nh_exception` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L58]. Members `*fnhe_rth_input` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L66] and `*fnhe_rth_output` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L67] of that structure are used to cache *routing decisions* in the same way as the `struct fib_nh_common` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L80] members used for *nexthop* caching.

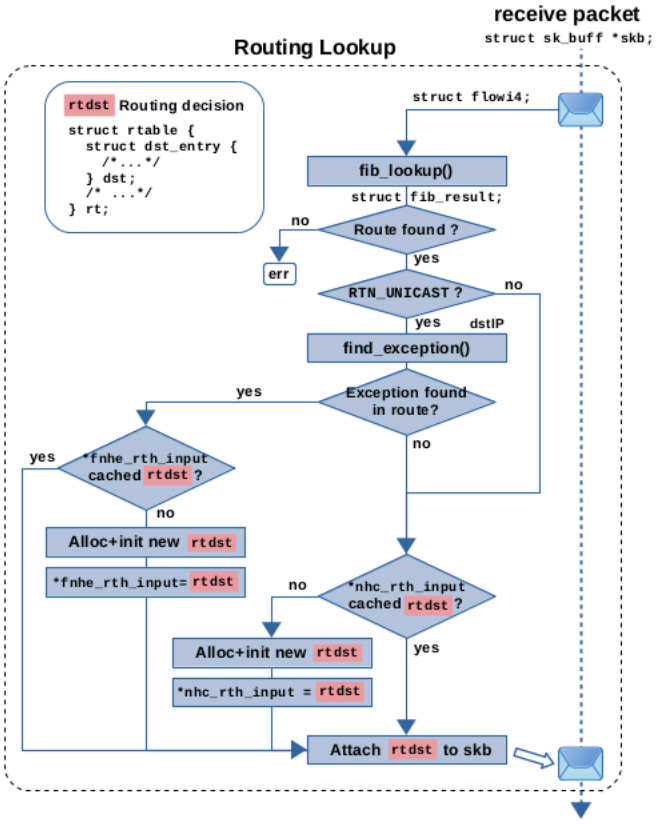

Figure 9: Routing table lookup + *FIB nexthop exception* caching (fnhe) + *nexthop* caching on receive path.

Figure 9 illustrates how that works on the receive path. Please compare it to Figure 6. Figure 9 shows the very same *nexthop* caching, but adds the *nexthop exception* caching mechanism as additional detail. The latter mechanism is actually only performed for packets on the receive path, which are to be forwarded and not destined for local reception (check for `RTN_UNICAST`). If the result of `fib_lookup()` determines that a packet is to be forwarded, then function `find_exception()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1849] is called to check whether the matching routing entry contains an *exception* which matches to the packet's destination IP address. If that is not the case, then everything works as described in the previous section. If a matching *exception* is found, member `*fnhe_rth_input` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1852] is checked for a cached *routing decision*, which, if it exists, is used and attached to the packet. Otherwise, a new *routing decision* object is allocated and initiated, cached in `*fnhe_rth_input` and attached to the packet. Thus, the caching mechanism works analogue to the *nexthop* caching.

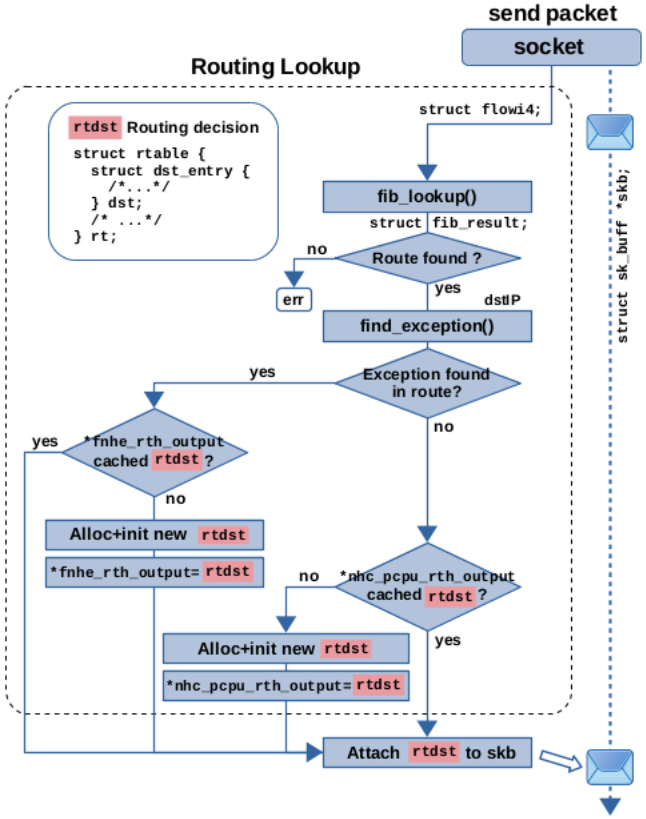

Figure 10: Routing table lookup + *FIB nexthop exception* caching (fnhe) + *nexthop* caching on local output path.

Figure 10 illustrates how that works on the local output path. Please compare it to Figure 7. Figure 10 shows the very same *nexthop* caching, but adds the *nexthop exception* caching mechanism as additional detail. Function `find_exception()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1849] is called to check whether the matching routing entry returned by `fib_lookup()` contains a *nexthop exception* which matches to the packet's destination IP address. If that is not the case, then everything works as described in the previous section. If a matching *exception* is found, member `*fnhe_rth_output` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/ip_fib.h#L67] is checked for a cached *routing decision*, which is used, if it exists. Otherwise, a new *routing decision* object is allocated, initiated, and cached in `*fnhe_rth_output`. The actual attachment of the *routing decision* to the packet happens at a later point in a function several layers up the call stack.

Socked Caching (TX)

This caching feature is being used on the local output path; thus, only for sending locally generated packets, not for received or forwarded packets. When a socket on the system (e.g. TCP or UDP socket, client or server) is sending data on the the network, then it is sufficient to only do a routing lookup for the very first packet it sends. After all, the destination IP address is the same for all following packets it sends and so is the routing decision. Thus, the *routing decision* object, once obtained from the initial lookup, is being cached within the socket. This feature has been around for a long time. It already had been present in kernel versions prior to 3.6 and thereby co-existed with the old routing cache. It is still present in modern kernels and here co-exists with the described *nexthop* caching and *nexthop exception* caching. It is actually built on top of those caching features, as shown in Figure 11. In other words, for each packet generated by a local socket, which is to be sent out on the network, first the socket cache it checked. If it already contains a *routing decision* object, then this one is attached to the packet and all is done. Otherwise, the complete routing lookup is performed. In kernels prior to 3.6 this means the routing cache lookup, plus (if needed) policy-based routing and the routing table lookup. In modern kernels this means policy-based routing, routing table lookup and afterwards *nexthop* caching or *nexthop exception* caching. Because of this feature, I emphasized in the sections above, that the *routing decision* object is not immediately being attached to the packet on the local output path. This logic is integrated with socket caching and in the end, an object is attached to the packet which either comes from the socket cache, from a routing table lookup, from the *nexthop* cache or from the *nexthop exception* cache.

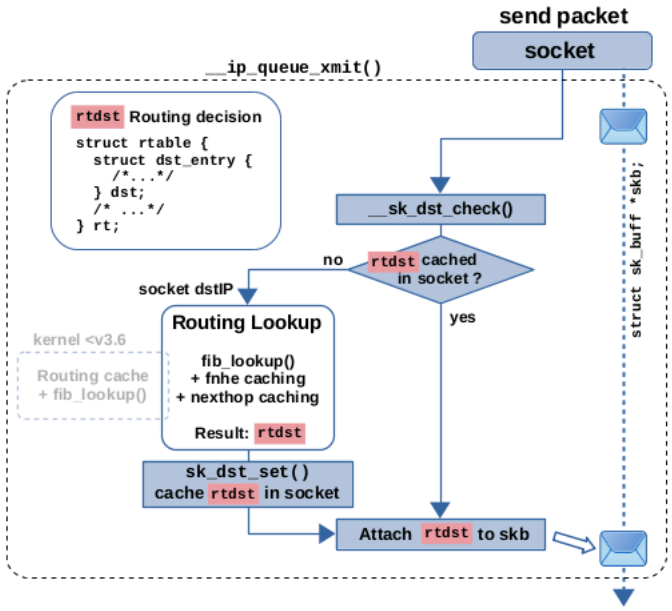

Figure 11: Routing table lookup and socket-based caching on the local output path.

For TCP, you'll find this feature implemented in function ip_queue_xmit() [https://elixir.bootlin.com/linux/v3.5/source/net/ipv4/ip_output.c#L335] in kernels prior to v3.6 and in function __ip_queue_xmit() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L455] in kernel v5.14. The code stayed very much the same between these two versions. Function __sk_dst_check() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L477], which in turn calls __sk_dst_get() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/sock.h#L1954], is used to check for the *routing decision* object in the socket cache. The cache itself is implemented as member sk_dst_cache [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/sock.h#L432] of the the socket structure.

```
struct sock {
   /* ... */
   struct dst_entry __rcu *sk_dst_cache;
};
```

If this cache currently does not hold a *routing decision* object, then ip_route_output_ports() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L490] is called, which executes the full-blown routing lookup as described in the sections above. The resulting *routing decision* object is then stored in the socket cache in function sk_setup_caps() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L499], which in turn calls function sk_dst_set() [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/sock.c#L2134]. Finally, cache or not, the *routing decision* is attached to the network packet by function skb_dst_set_noref() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_output.c#L501].

Socket Caching (RX)

There is yet another Socket-based caching feature. However, this one is only present in modern kernels. It is also referred to as *IP early demultiplexing* and is targeted at network packets on the receive path, which will end up to be received by a local TCP/UDP socket on the system; see Figure 12.

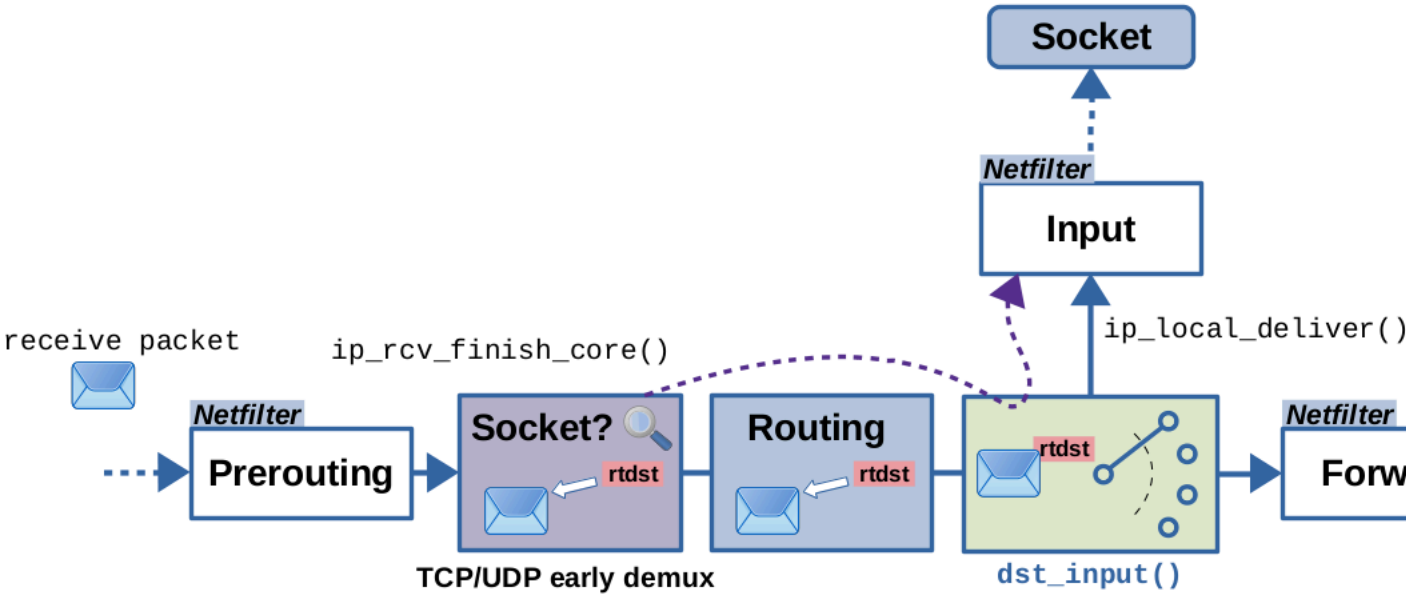

Figure 12: *IP early demultiplexing* for TCP/UDP sockets before Routing Lookup on the receive path (click to enlarge).

Without that feature, the normal routing lookup, including *nexthop* caching, would determine if a packet is destined for local reception. This feature adds an additional hash table lookup directly before the routing lookup on the receive path in function ip_rcv_finish_core() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L331]. This hash table contains the TCP/UDP sockets which currently exist on the system and already have an established connection. A hash lookup is done based on source and destination IP address and TCP/UDP source and destination port of the network packet, the input interface and the network namespace. If a match is found, a pointer to an instance of struct sock [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/sock.h#L351], which represents the matching socket, is returned. In the last section you saw, that this structure has a member variable

sk_dst_cache [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/sock.h#L432], which is used to cache a routing decision object on the output path. For this *early demux* feature, the structure provides yet another member to cache a routing decision, named sk_rx_dst [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/sock.h#L431].

```
struct sock {
   /* ... */
   struct dst_entry *sk_rx_dst;
};
```

If this member already holds a routing decision, then, after some validity checks, it is attached to the network packet. Thereby, the normal routing lookup can be skipped and the packet continues its way on the local input path via dst_input() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L456] doing an indirect call to what the attached routing decision object holds in its (*input)() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L35] function pointer, which means, ip_local_deliver() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L240] is being called[5]. In contrast to the old *routing cache*, this feature is not susceptible to DoS attacks in the same way, because it is not enough to just send an IP packet to the system which contains a new source and destination IP address pair, to create another entry in the hash table. New entries are only added in case an existing socket on the system reaches *established* state and this is done within OSI layer 4 handling, e.g. for TCP in function tcp_rcv_established() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/tcp_input.c#L5742]. Sysctls are implemented on network namespace level to switch this feature on/off either globally for IPv4 within the network namespace or just for TCP and/or UDP (default: on).

```
$ sudo sysctl -a -r early_demux
net.ipv4.ip_early_demux = 1
net.ipv4.tcp_early_demux = 1
net.ipv4.udp_early_demux = 1
```

Hint caching

It is debatable, whether this is actually a *caching* feature. However, it is yet another runtime optimization based on *routing decision* objects, which makes it possible to skip the actual full-blown routing lookup on the receive path under certain circumstances. You will only find it in modern kernels. As I mentioned in the previous article, IPv4 packets on the receive path can either be handled one by one by function ip_rcv() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L531] or based on packet lists by function ip_list_rcv() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L613]. This optimization feature here is only used in the latter case. To explain it, let me first explain how list-based packet handling works when it comes to the routing lookup on the receive path. Figure 13 illustrates this.

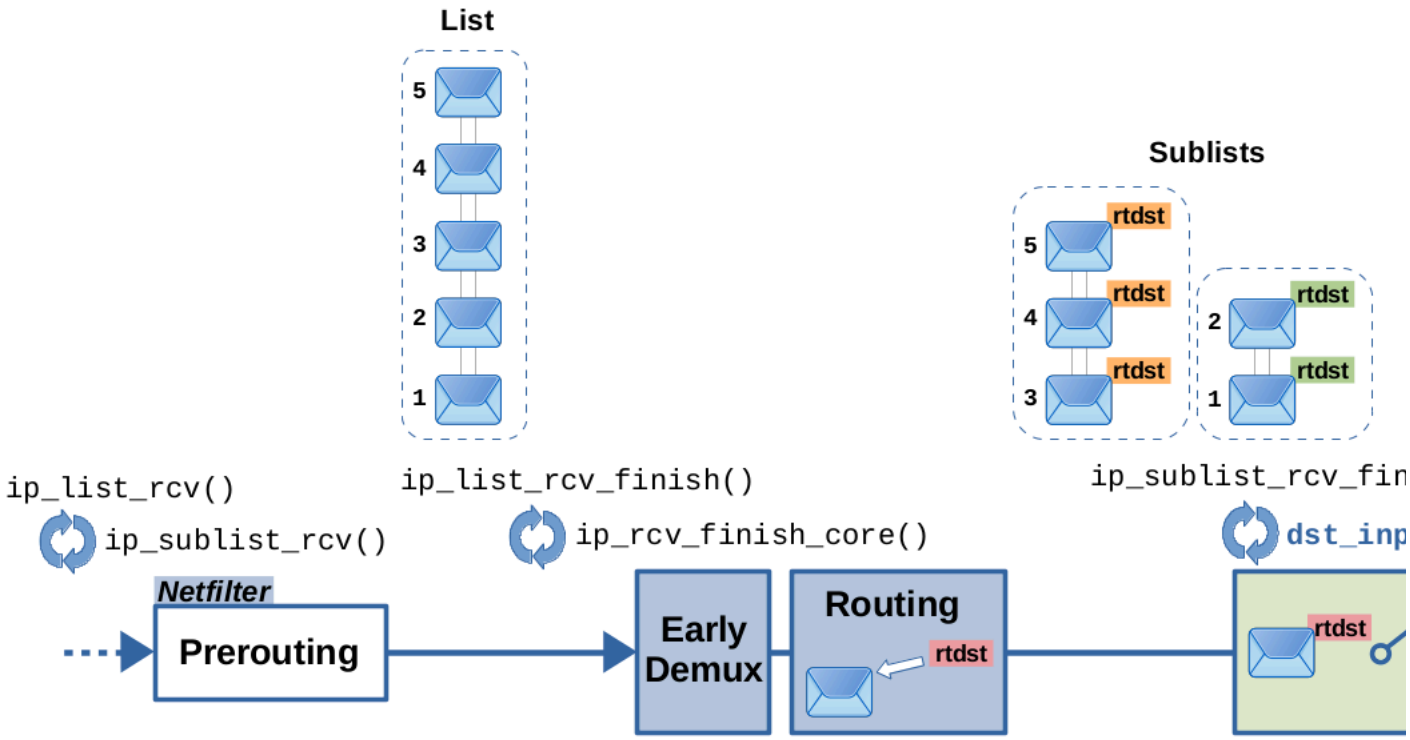

Figure 13: List-based packet handling on the receive path (click to enlarge).

After traversal of the Netfilter *Prerouting* hook, function ip_list_rcv_finish() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L564] is called and is given a list of packets. This function loops through the packets in the list and calls ip_rcv_finish_core() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L315] for each packet. The latter function contains the *early demux* feature and the full-blown routing lookup, which includes the *nexthop exception* and *nexthop* caching features explained above. While looping through the packets, ip_list_rcv_finish() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L564] groups them into sublists, based on the attached *routing decision*; meaning, successive packets which got the same *routing decision* attached, end up in the same sublist. In general it is quite likely that packets which are received in a burst directly one after another, turn out to have the same source and destination and thereby the same resulting *routing decision* attached. Thus, this grouping makes sense. Those sublists are then one by one fed into function ip_sublist_rcv_finish() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L545], which loops through the packets of the given sublist and calls dst_input() [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L456] for each packet. Ok, so far so good. Now let's get to this optimization feature; see Figure 14.

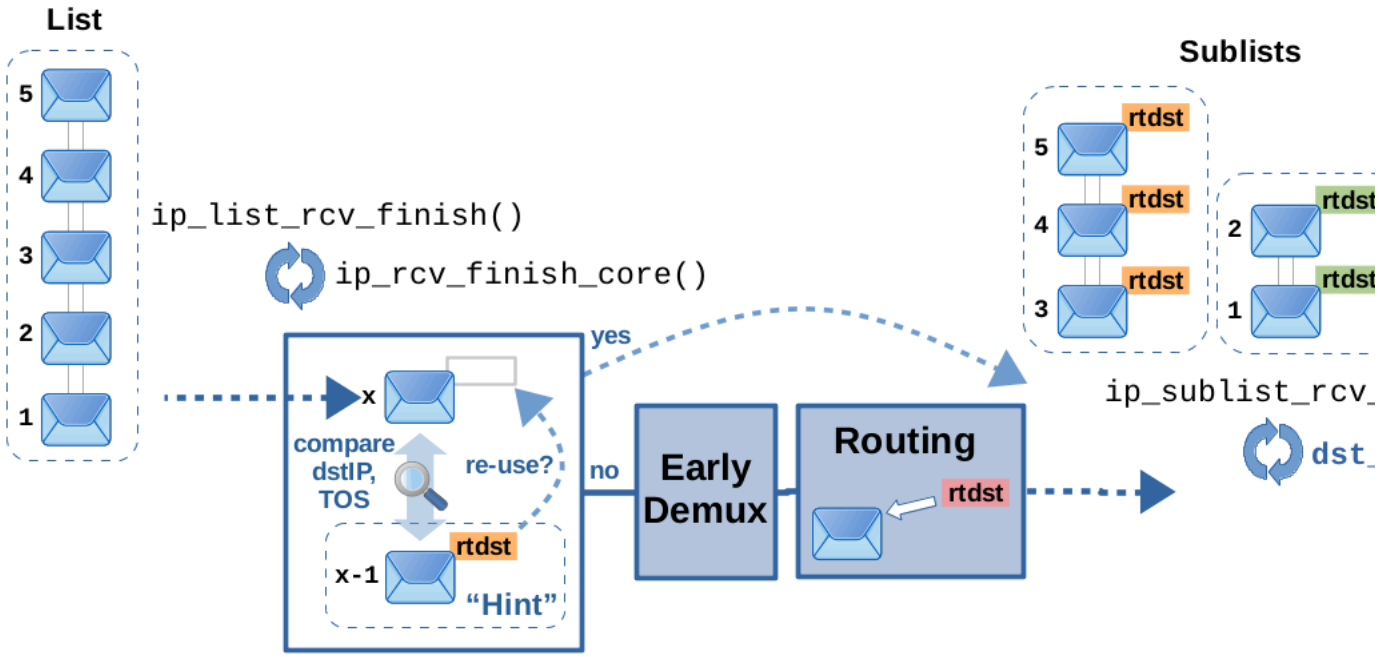

Figure 14: Using previous packet as *hint* and check, whether it's attached *routing decision* can be re-used (click to enlarge).

While `ip_list_rcv_finish()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L564] is looping through the packets of its given list, it remembers the previously handled packet as *hint* for the next packet. As I said, it is quite likely that successive packets within the same list will get the same *routing decision* attached. So, when `ip_rcv_finish_core()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L324] is called for the next packet, it gets the previous packet as *hint* and compares destination IP address and TOS value of both packets. If they are identical, the *routing decision* attached to the previous packet can simply be re-used for the next packet. Thereby, the *early demux* and the actual routing lookup can be skipped this this case, which results in a runtime speed-up. However, there are limitations to this feature. A check in `ip_list_rcv_finish()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L588], done by function `ip_extract_route_hint()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/ip_input.c#L555], prevents the *hint* from being used for *broadcast* packets and in case non-default policy-based routing rules are present. Remember, while this is not the case for the default rules, "real" policy-based routing rules can actually select a different routing table for lookup, based on parameters like the packet's source IP address, the `skb->mark` and so on. Thereby, you cannot simply re-use the *routing decision* of the previous packet in this case, just based on it having the same destination IP address and TOS value.

Flowtables

Flowtables is a software (and under certain circumstances also hardware) fastpath mechanism implemented by the *Netfilter* developers, which speeds up forwarded network packets by making them skip a big part of the normal slowpath handling. It is set up by Nftables rules, is based on the *connection tracking* feature and caches *routing decisions* in a hash table. Lookup into that table and thereby fastpath/slowpath demultiplexing for a network packet happens early on reception in the *Netfilter Ingress hook*. I describe *Flowtables* in detail in a separate article series, starting with Flowtables - Part 1: A Netfilter/Nftables Fastpath.

Cache Invalidation

What actually happens in case something changes, like a new entry being added to a routing table or an up/down state change of a network interface? Those events, and many others, do render currently cached routing decisions obsolete. Thus, there must be a means to get rid of them. This is called *cache invalidation* and seems it is implemented in the same way already for a long time, dating back to kernels earlier than v3.6. In other words, no matter if we are talking about the old *routing cache* or any of the newer caching mechanisms which I described in the sections above, they all use the same way to invalidate cached *routing decision* objects which are outdated. So, how does this work? In case of an event which invalidates the cache, you essentially would need to clean up ALL decisions currently cached in the system and you would need to do it immediately and it even would require to block the cache from being used until cleanup is done. This would suddenly consume a lot of CPU power and RAM. The latter, because you cannot immediately delete decisions which are currently still in use (still attached to skbs) and need to put those on a garbage list. Seems it was actually done this way a long time ago. However, developers came up with a more elegant way, which dates back to this commit [https://lists.openwall.net/netdev/2008/01/30/42]. They added a so-called *generation identifier* to each *routing decision* instance and created a global counterpart, so that cache invalidation could be performed by merely changing this global *generation identifier*, without having to scan through all instances. This "global" *generation identifier* is implemented globally within each network namespace in form of the atomic integer `rt_genid` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/netns/ipv4.h#L225]:

```
struct net {
        /* ... */
        struct netns_ipv4 {
                /* ... */
                atomic_t rt_genid;
        } ipv4;
};
```

Events that invalidate the cache are required to call function `rt_genid_bump_ipv4()` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/net_namespace.h#L443], which atomically increments this integer. Further, every cache entry (= every *routing decision* object) requires its own *generation identifier* member `rt_genid` [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/route.h#L52] as a counterpart to the global one:

```
struct rtable {
        struct dst_entry {
                short obsolete;
                /* ... */
        } dst;
        int rt_genid;
        /* ... */
};
```

When a new *routing decision* object is allocated by `rt_dst_alloc()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1644] or `rt_dst_clone()` [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1673], its *generation identifier* is simply set to the current value of the global *generation identifier*. The integer `obsolete`

[https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L55] also plays a role here. Both functions set it to DST_OBSOLETE_FORCE_CHK [https://elixir.bootlin.com/linux/v5.14.7/source/include/net/dst.h#L58]. This value specifies that the *generation identifier* mechanism has to be used for that instance. Thus, when a *routing decision* is about to be used once more (e.g. when it is about to be attached to yet another skb), its *generation identifier* value needs to be compared to the global one, which is done by function rt_is_expired() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L399]. If both are unequal, the cache entry is considered expired. In case of e.g. the *nexthop* caching mechanism described above, that function is wrapped inside function rt_cache_valid() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1567], which first checks the value of member `obsolete`:

```
static bool rt_cache_valid(const struct rtable *rt)
{
        return  rt &&
                rt->dst.obsolete == DST_OBSOLETE_FORCE_CHK &&
                !rt_is_expired(rt);
}
```

You can see, that if `obsolete` contains another value, then `rt_is_expired()` would not be called, but this instance would then anyway be considered expired. There are many more functions which all end up calling `rt_is_expired()`. Figure 15 shows several ones; no guarantee for completeness. You'll probably recognize these calls originating from several of the caching and optimization features described above, like the socket-based caching, early-demux and flowtables.

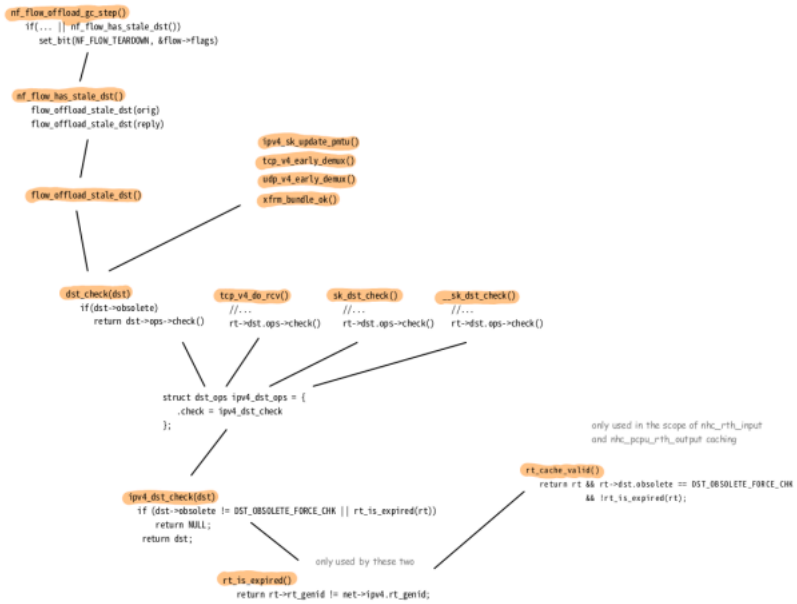

Figure 15: An overview of functions which call `rt_is_expired()` (click to enlarge).

There are many events which call `rt_genid_bump_ipv4()` and thereby invalidate all currently cached *routing decisions*. Figure 16 shows several ones; no guarantee for completeness. Examples are content changes in the routing table(s), MAC address changes, network device configuration and status (up/down) changes, writes to network stack related sysctls, IPsec (xfrm) policy changes, and so on.

Figure 16: An overview of the events which cause an increment of `rt_genid` (click to enlarge).

The Slab Cache

This is actually no *routing cache* mechanism. I just mention it here to prevent confusion. The *Slab cache* or *Slab allocation* is a common memory allocation mechanism within the Linux kernel. It is used in cases where objects of the same type and size need to be allocated and freed very frequently, like it is the case for e.g. *file descriptors*, *tasks* handled by the scheduler, *skb*s and so on. Simply spoken, this allocation mechanism provides pre-allocated *caches* for multiple instances of a certain data type and size. As a result, allocation of an instance of that type can be done very fast. Deallocation/freeing does usually not actually free the allocated memory. It just frees one slot within the cache which is preserved to be re-used by yet another allocation which is likely to happen any time soon. *Routing decisions*; thus, instances of `struct rtable` and its internal `struct dst_entry`, are being allocated and freed by this *Slab allocation* mechanism. This applies to ALL instances of those structs, no matter if they end up being used within one of the *routing cache* mechanisms which I described above or if they are simply attached to a single *skb*. You can observe this in function rt_dst_alloc() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L1633], which allocates and initializes a new *routing decision*. It calls dst_alloc() [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/dst.c#L79], which in turn calls kmem_cache_alloc() [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/slab.h#L429] to do the actual memory allocation. The latter function is part of the interface provided by the *Slab allocator*. Its counterpart for freeing is kmem_cache_free() [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/slab.h#L430], which is called by dst_destroy() [https://elixir.bootlin.com/linux/v5.14.7/source/net/core/dst.c#L103]. The *Slab cache* for `struct rtable` instances is being initialized in function ip_rt_init() [https://elixir.bootlin.com/linux/v5.14.7/source/net/ipv4/route.c#L3687] by calling kmem_cache_create() [https://elixir.bootlin.com/linux/v5.14.7/source/include/linux/slab.h#L147]. In case you like to do some further reading: The blog article The Slab Allocator in the Linux kernel (Andria Di Dio, 2020)

[https://hammertux.github.io/slab-allocator] describes the concept of *Slab allocation* and its implementation on Linux very nicely. One more detail on the way to prevent/add more confusion: There are 3 different backends to the programming interface of the *Slab allocation* mechanism: SLOB [https://en.wikipedia.org/wiki/SLOB], SLAB [https://en.wikipedia.org/wiki/Slab_allocation] and SLUB [https://en.wikipedia.org/wiki/SLUB_(software)]. By default, *Slub* is used in modern kernels. You can choose [https://elixir.bootlin.com/linux/v5.14.7/source/init/Kconfig#L1868] between those three backends when configuring the kernel at build time.

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. An exception to that is section The "old" Routing Cache, which describes Kernel version 3.5.

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.

published 2022-07-31, *last modified 2023-12-14*

---

1)
This function includes Policy-based routing and the actual lookup into the routing table(s). I described that in detail in the previous article.
2)
My previous article contains some links to further details on that algorithm.
3)
see previous article for more details and links to this topic
4)
I am not entirely sure if that is really always the case. Someone correct me here please, if this is not true…
5)
I described that in detail in the previous article.
