At Cloudflare we develop new products at a great pace. Their needs often challenge the architectural assumptions we made in the past. For example, years ago we decided to avoid using Linux's "conntrack" - stateful firewall facility. This brought great benefits - it simplified our iptables firewall setup, sped up the system a bit and made the inbound packet path easier to understand.

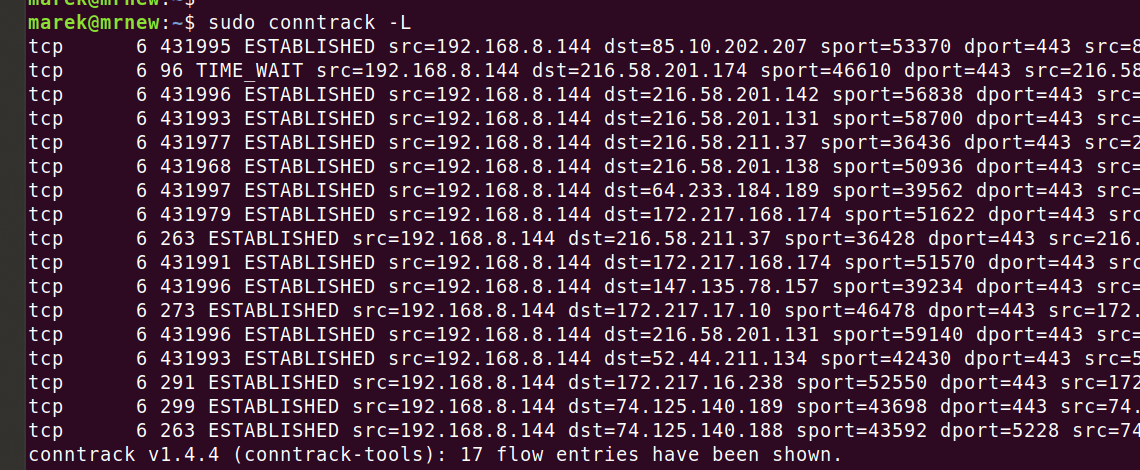
But eventually our needs changed. One of our new products had a reasonable need for it. But we weren't confident - can we just enable conntrack and move on? How does it actually work? I volunteered to help the team understand the dark corners of the "conntrack" subsystem.

**What is conntrack?**

"Conntrack" is a part of Linux network stack, specifically part of the firewall subsystem. To put that into perspective: early firewalls were entirely stateless. They could express only basic logic, like: allow SYN packets to port 80 and 443, and block everything else.

The stateless design gave some basic network security, but was quickly deemed insufficient. You see, there are certain things that can't be expressed in a stateless way. The canonical example is assessment of ACK packets - it's impossible to say if an ACK packet is legitimate or part of a port scanning attempt, without tracking the connection state.

To fill such gaps all the operating systems implemented connection tracking inside their firewalls. This tracking is usually implemented as a big table, with at least 6 columns: protocol (usually TCP or UDP), source IP, source port, destination IP, destination port and connection state. On Linux this subsystem is called "conntrack" and is often enabled by default. Here's how the table looks on my laptop inspected with "conntrack -L" command:



The obvious question is how large this state tracking table can be. This setting is under "/proc/sys/net/nf\_conntrack\_max":

$ cat /proc/sys/net/nf\_conntrack\_max

262144

This is a global setting, but the limit is per container. On my system each container, or "network namespace", can have up to 256K conntrack entries.

What exactly happens when the number of concurrent connections exceeds the conntrack limit?

**Testing conntrack is hard**

In past testing conntrack was hard - it required complex hardware or vm setup. Fortunately, these days we can use modern "user namespace" facilities which do permission magic, allowing an unprivileged user to feel like root. Using the tool "unshare" it's possible to create an isolated environment where we can precisely control the packets going through and experiment with iptables and conntrack without threatening the health of our host system. With appropriate parameters it's possible to create and manage a networking namespace, including access to namespaced iptables and conntrack, from an unprivileged user.

This script is the heart of our test:

# Enable tun interface

ip tuntap add name tun0 mode tun

ip link set tun0 up

ip addr add 192.0.2.1 peer 192.0.2.2 dev tun0

ip route add 0.0.0.0/0 via 192.0.2.2 dev tun0

# Refer to conntrack at least once to ensure it's enabled

iptables -t raw -A PREROUTING -j CT

# Create a counter in mangle table

iptables -t mangle -A PREROUTING

# Make sure reverse traffic doesn't affect conntrack state

iptables -t raw -A OUTPUT -p tcp --sport 80 -j DROP

tcpdump -ni any -B 16384 -ttt &

...

./venv/bin/python3 send\_syn.py

conntrack -L

# Show iptables counters

iptables -nvx -t raw -L PREROUTING

iptables -nvx -t mangle -L PREROUTING

This bash script is shortened for readability. See the [full version here](https://github.com/cloudflare/cloudflare-blog/blob/master/2020-04-conntrack-syn/test-1.bash). The accompanying "send\_syn.py" is just sending 10 SYN packets over "tun0" interface. [Here is the source](https://github.com/cloudflare/cloudflare-blog/blob/master/2020-04-conntrack-syn/send_syn.py) but allow me to paste it here - showing off "scapy" is always fun:

tun = TunTapInterface("tun0", mode\_tun=True)

tun.open()

for i in range(10000,10000+10):

ip=IP(src="198.18.0.2", dst="192.0.2.1")

tcp=TCP(sport=i, dport=80, flags="S")

send(ip/tcp, verbose=False, inter=0.01, socket=tun)

The bash script above contains a couple of gems. Let's walk through them.

First, please note that we can't just inject packets into the loopback interface using [SOCK\_RAW sockets](http://man7.org/linux/man-pages/man7/raw.7.html). The Linux networking stack is a complex beast. The semantics of sending packets over a SOCK\_RAW are different then delivering a packet over a real interface. We'll discuss this later, but for now, to avoid triggering unexpected behaviour, we will deliver packets over a tun/tap device which better emulates a real interface.

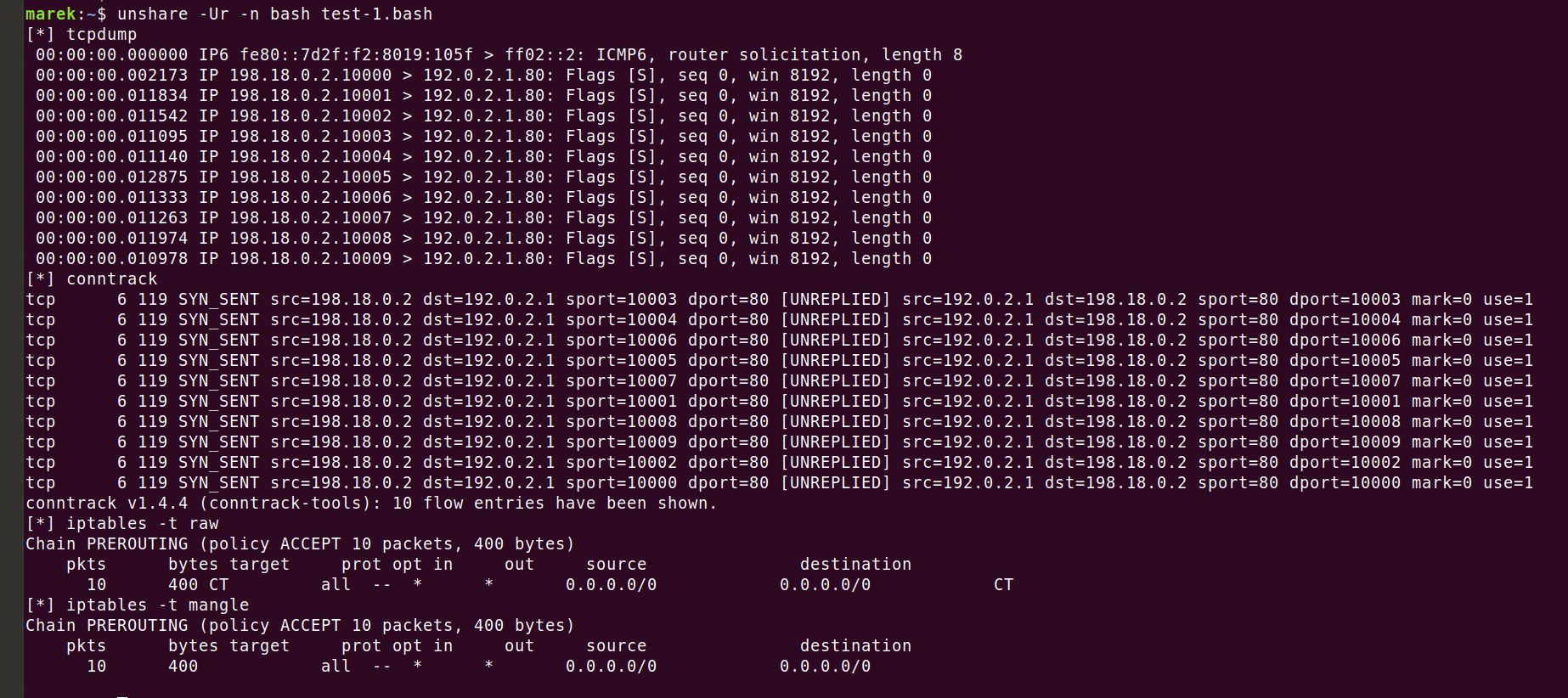
Then we need to make sure the conntrack is active in the network namespace we wish to use for testing. Traditionally, just loading the kernel module would have done that, but in the brave new world of containers and network namespaces, a method had to be found to allow conntrack to be active in some and inactive in other containers. Hence this is tied to usage - rules referencing conntrack must exist in the namespace's iptables for conntrack to be active inside the container.

As a side note, [containers triggering host to load kernel modules](https://lwn.net/Articles/740455/) is an [interesting subject](https://github.com/weaveworks/go-odp/blob/6b0aa22550d9325eb8f43418185859e13dc0de1d/odp/dpif.go#L67-L90).

After the "-t raw -A PREROUTING" rule, which we added "-t mangle -A PREROUTING" rule, but notice - it doesn't have any action! This syntax is allowed by iptables and it is pretty useful to get iptables to report rule counters. We'll need these counters soon. A careful reader might suggest looking at "policy" counters in iptables to achieve our goal. Sadly, "policy" counters (increased for each packet entering a chain), work only if there is at least one rule inside it.

The rest of the steps are self-explanatory. We set up "tcpdump" in the background, send 10 SYN packets to 127.0.0.1:80 using the "scapy" Python library. Then we print the conntrack table and iptables counters.

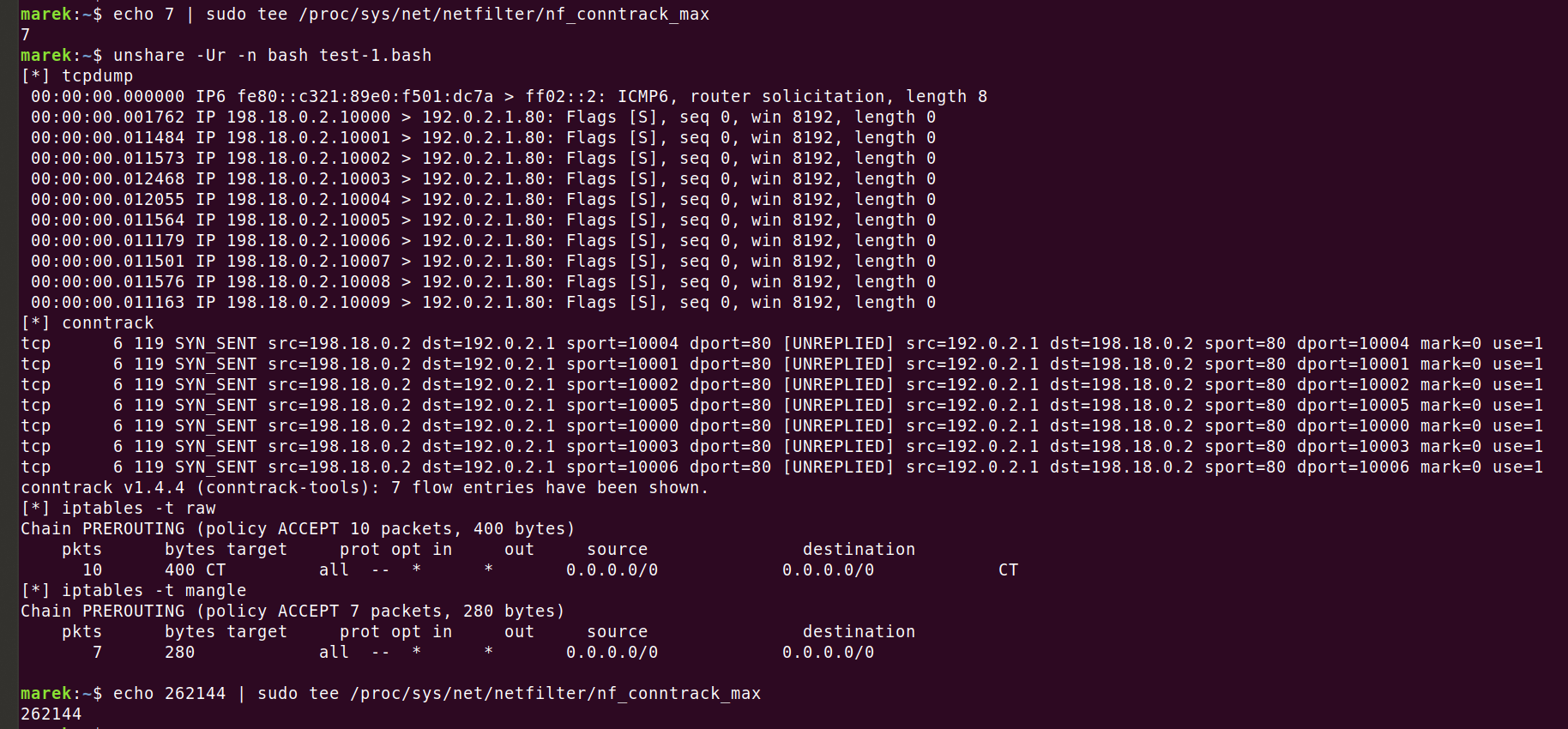
Let's run this script in action. Remember to run it under networking namespace as fake root with "unshare -Ur -n":



This is all nice. First we see a "tcpdump" listing showing 10 SYN packets. Then we see the conntrack table state, showing 10 created flows. Finally, we see iptables counters in two rules we created, each showing 10 packets processed.

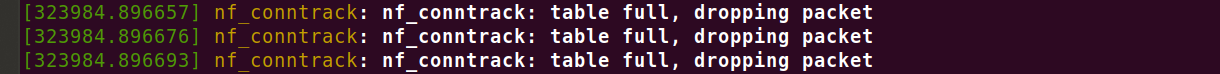
**Can conntrack table fill up?**

Given that the conntrack table is size constrained, what exactly happens when it fills up? Let's check it out. First, we need to drop the conntrack size. As mentioned it's controlled by a global toggle - it's necessary to tune it on the host side. Let's reduce the table size to 7 entries, and repeat our test:

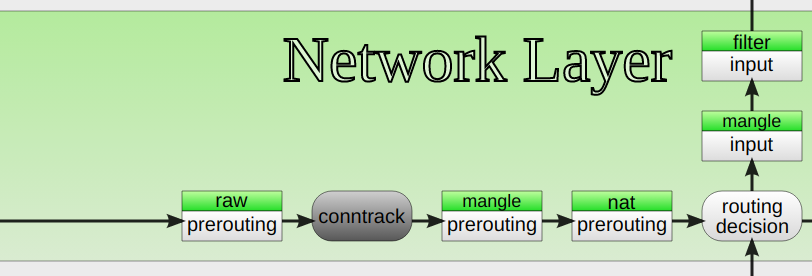


This is getting interesting. We still see the 10 inbound SYN packets. We still see that the "-t raw PREROUTING" table received 10 packets, but this is where similarities end. The "-t mangle PREROUTING" table saw only 7 packets. Where did the three missing SYN packets go?

It turns out they went where all the dead packets go. They were hard dropped. Conntrack on overfill does exactly that. It even complains in the "dmesg":



This is confirmed by our iptables counters. Let's review the [famous iptables](https://upload.wikimedia.org/wikipedia/commons/3/37/Netfilter-packet-flow.svg) diagram:

[image](https://commons.wikimedia.org/wiki/File:Netfilter-packet-flow.svg) by [Jan Engelhardt](https://commons.wikimedia.org/wiki/User_talk:Jengelh) CC BY-SA 3.0

As we can see, the "-t raw PREROUTING" happens before conntrack, while "-t mangle PREROUTING" is just after it. This is why we see 10 and 7 packets reported by our iptables counters.

Let me emphasize the gravity of our discovery. We showed three completely valid SYN packets being implicitly dropped by "conntrack". There is no explicit "-j DROP" iptables rule. There is no configuration to be toggled. Just the fact of using "conntrack" means that, when it's full, packets creating new flows will be dropped. No questions asked.

This is the dark side of using conntrack. If you use it, you absolutely must make sure it doesn't get filled.

We could end our investigation here, but there are a couple of interesting caveats.

**Strict vs loose**

Conntrack supports a "strict" and "loose" mode, as configured by "nf\_conntrack\_tcp\_loose" toggle.

$ cat /proc/sys/net/netfilter/nf\_conntrack\_tcp\_loose

1

By default, it's set to "loose" which means that stray ACK packets for unseen TCP flows will create new flow entries in the table. We can generalize: "conntrack" will implicitly drop all the packets that create new flow, whether that's SYN or just stray ACK.

What happens when we clear the "nf\_conntrack\_tcp\_loose=0" setting? This is a subject for another blog post, but suffice to say - it's a mess. First, this setting is not settable in the network namespace scope - although it should be. To test it you need to be in the root network namespace. Then, due to twisted logic the ACK will be dropped on a full conntrack table, even though in this case it doesn't create a flow. If the table is not full, the ACK packet will pass through it, having "-ctstate INVALID" from "mangle" table forward.

**When doesn't a conntrack entry get created?**

There are important situations when conntrack entry is not created. For example, we could replace these line in our script:

# Make sure reverse traffic doesn't affect conntrack state

iptables -t raw -A OUTPUT -p tcp --sport 80 -j DROP

With those:

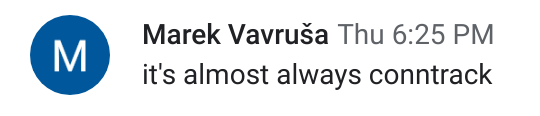
# Make sure inbound SYN packets don't go to networking stack

iptables -A INPUT -j DROP

Naively we could think dropping SYN packets past the conntrack layer would not interfere with the created flows. This is not correct. In spite of these SYN packets having been seen by conntrack, no flow state is created for them. Packets hitting "-j DROP" will not create new conntrack flows. Pretty magical, isn't it?

**Full Conntrack causes with EPERM**

Recently we hit a case when a "sendto()" syscall on UDP socket from one of our applications was erroring with EPERM. This is pretty weird, and not documented in the man page. My colleague had no doubts:



I'll save you the gruesome details, but indeed, the full conntrack table will do that to your new UDP flows - you will get EPERM. Beware. Funnily enough, it's possible to get EPERM if an outbound packet is dropped on OUTPUT firewall in other ways. For example:

marek:~$ sudo iptables -I OUTPUT -p udp --dport 53 --dst 192.0.2.8 -j DROP

marek:~$ strace -e trace=write nc -vu 192.0.2.8 53

write(3, "X", 1) = -1 EPERM (Operation not permitted)

+++ exited with 1 +++

If you ever receive EPERM from "sendto()", you might want to treat it as a transient error, if you suspect a filled conntrack problem, or permanent error if you blame iptables configuration.

This is also why we can't send our SYN packets directly using SOCK\_RAW sockets in our test. Let's see what happens on conntrack overfill with standard "hping3" tool:

$ hping3 -S -i u10000 -c 10 --spoof 192.18.0.2 192.0.2.1 -p 80 -I lo

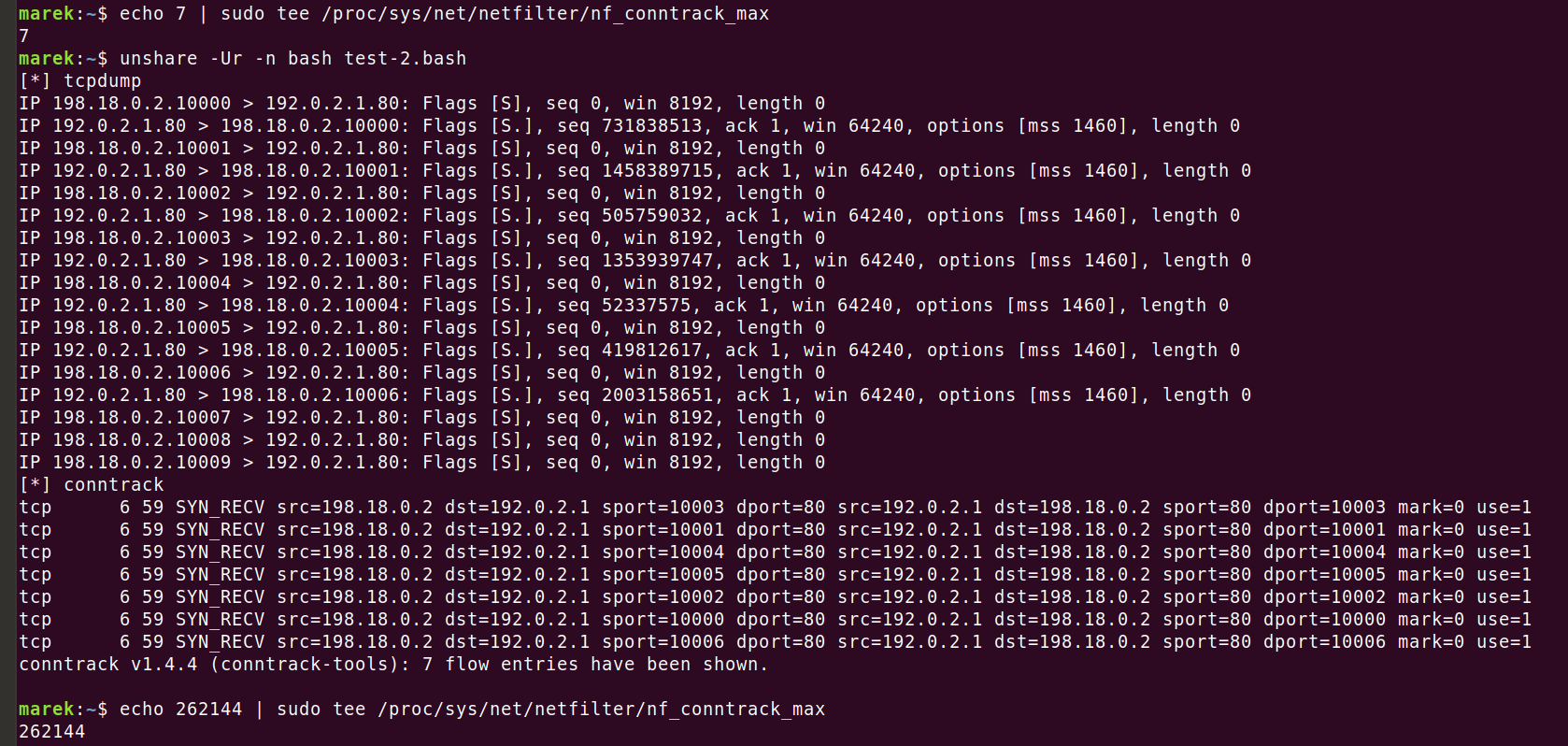
HPING 192.0.2.1 (lo 192.0.2.1): S set, 40 headers + 0 data bytes

[send\_ip] sendto: Operation not permitted

"send()" even on a SOCK\_RAW socket fails with EPERM when conntrack table is full.

**Full conntrack can happen on a SYN flood**

There is one more caveat. During a SYN flood, the conntrack entries will totally be created for the spoofed flows. Take a look at second test case we prepared, this time correctly listening on port 80, and sending SYN+ACK:



We can see 7 SYN+ACK's flying out of the port 80 listening socket. The final three SYN's go nowhere as they are dropped by conntrack.

This has important implications. If you use conntrack on publicly accessible ports, during SYN flood [mitigation technologies like SYN Cookies](https://blog.cloudflare.com/syn-packet-handling-in-the-wild/) won't help. You are still at risk of running out of conntrack space and therefore affecting legitimate connections.

For this reason, as a general rule consider avoiding conntrack on inbound connections (-j NOTRACK). Alternatively having some reasonable rate limits on iptables layer, doing "-j DROP". This will work well and won't create new flows, as we discussed above. The best method though, would be to trigger SYN Cookies from a layer before conntrack, like XDP. But this is a subject for another time.

**Summary**

Over the years Linux conntrack has gone through many changes and has improved a lot. While performance used to be a major concern, these days it's considered to be very fast. Dark corners remain. Correctly applying conntrack is tricky.

In this blog post we showed how it's possible to test parts of conntrack with "unshare" and a series of scripts. We showed the behaviour when the conntrack table gets filled - packets might implicitly be dropped. Finally, we mentioned the curious case of SYN floods where incorrectly applied conntrack may cause harm.

Stay tuned for more horror stories as we dig deeper and deeper into the Linux networking stack guts.

Connection tracking (“conntrack”) is a core feature of the Linux kernel’s networking stack.  It allows the kernel to keep track of all logical network connections or flows, and thereby identify all of the packets which make up each flow so they can be handled consistently together.

Conntrack is an important kernel feature that underpins some key mainline use cases:

* NAT relies on the connection tracking information so it can translate all of the packets in a flow in the same way. For example, when a pod accesses a Kubernetes service, kube-proxy’s load balancing uses NAT to redirect the connection to a particular backend pod. It is conntrack that records that for a particular connection, packets to the service IP should all be sent to the same backend pod, and that packets returning from backend pod should be un-NATed back to the source pod.
* Stateful firewalls, such as Calico, rely on the connection tracking information to precisely whitelist “response” traffic. This allows you to write a network policy that says “allow my pod to connect to any remote IP” without needing to write policy to explicitly allow the response traffic.  (Without this you would have to add the much less secure rule “allow packets to my pod from any IP”.)

In addition, conntrack normally improves performance (reduced CPU and reduced packet latencies) since only the first packet in a flow needs to go through the full network stack processing to work out what to do with it.  See the “[Comparing kube-proxy modes](https://www.tigera.io/blog/comparing-kube-proxy-modes-iptables-or-ipvs/)” blog for one example of this in action.

However, conntrack has its limits…

**So, where does it break down?**

The conntrack table has a configurable maximum size and, if it fills up, connections will typically start getting rejected or dropped.  For most workloads, there’s plenty of headroom in the table and this will never be an issue.  However, there are a few scenarios where the conntrack table needs a bit more thought:

* The most obvious case is if your server handles an extremely high number of simultaneously active connections. For example, if your conntrack table is configured to be 128k entries but you have >128k simultaneous connections, you’ll definitely hit issues!
* The slightly less obvious case is if your server handles an extremely high number of connections per second. Even if the connections are short-lived, connections continue to be tracked by Linux for a short timeout period (120s by default). For example, if your conntrack table is configured to be 128k entries, and you are trying to handle 1,100 connections per second, that’s going to exceed the conntrack table size even if the connections are very short-lived (128k / 120s = 1092 connections/s).

There are some niche workload types fall into these categories. In addition, if you’re in a hostile environment then flooding your server with lots of half-open connections can be used as a denial-of-service attack.  In both cases, conntrack can become the limiting bottleneck in your system.  For some scenarios tuning conntrack may be sufficient to meet your needs by increasing the conntrack table size or reducing conntrack timeouts (but if you get this tuning wrong it can lead to a lot of pain).  For other scenarios, you need to bypass conntrack for the offending traffic.

**A real-world example**

To give a concrete example, one large SaaS provider we worked with had a set of memcached servers running on bare metal servers (not virtualized or containerized) each handling 50k+ short-lived connections per second. This is way more than a standard Linux config can cope with.

They had experimented with tuning conntrack configuration to increase table sizes and reduce timeouts, but the tuning was fragile, the increased RAM use was a significant penalty (think GBytes!), and the connections were so short-lived that conntrack was not giving its usual performance benefits (reduced CPU or packet latencies).

Instead, they turned to Calico.  Calico’s network policies allow you to bypass conntrack for specific traffic (using the doNotTrack flag).  This gave them the performance they needed, plus the additional security benefits that Calico brings.

**What are the trade-offs of bypassing conntrack?**

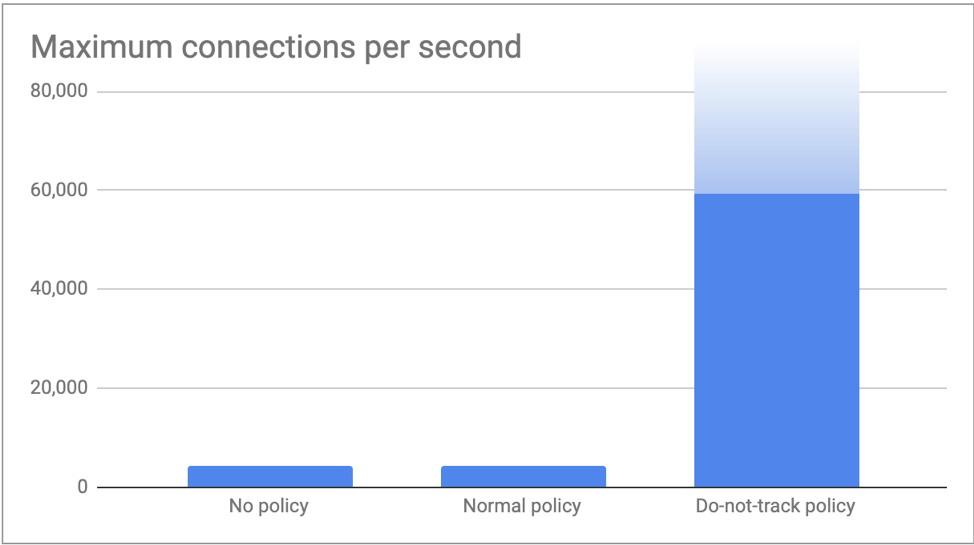
* Do-not-track network policy typically has to be symmetric. In the SaaS-provider’s case their workload was internal so using network policy they could very narrowly whitelist traffic to and from all workloads that were allowed to access the memcached service.
* Do-not-track policy is oblivious to the direction of the connection. So in the event that a memcached server was compromised, it could in theory attempt to connect out to any of the memcached clients so long as it used the right source port. However, assuming you correctly defined network policy for your memcached clients then these connection attempts will still be rejected at the client end.
* Do-not-track network policy is applied to every packet, whereas normal network policy is only applied to the first packet in a flow. This can increase the CPU cost per packet since every packet needs to be processed by network policy.  But with short-lived connections this extra processing is outweighed by the reduction in conntrack processing.  For example, in the SaaS provider’s case, the number of packets in each connection was very small so the additional overhead of applying policy to every packet was a reasonable trade-off.

**Putting it to the test**

We tested a single memcached server pod and a multitude of client pods running on remote nodes so we could drive very high connections per second. The memcached server pod host had 8 cores and a 512k entry conntrack table (standard setting for the size of the host).  We measured performance differences between: no network policy; Calico normal network policy; and Calico do-not-track network policy.

In the first test we limited the connections to 4,000 per second so we could focus on CPU differences. There was no measurable difference in performance no policy and normal policy, but do-not-track policy reduced CPU usage by around 20%.

In the second test, we pushed as many connections as our clients could muster and measured the maximum number of connections per second the memcached server was able to process.  As expected, no policy and normal policy both hit the conntrack table limit at just over 4,000 connections per second (512k / 120s = 4,369 connections/s).  With do-not-track policy in place our clients pushed 60,000 connections per second without hitting any issues.  We are confident we could have pushed beyond this by spinning up even more clients, but felt the numbers were already enough to illustrate the point of this blog!

[](https://dev-project-calico-2020.pantheonsite.io/wp-content/uploads/2019/04/Picture1-1.png)

**Conclusion**

Conntrack is an important kernel feature.  It’s good at what it does.  Many mainline use cases depend on it.  However, for some niche scenarios, the overhead of conntrack outweighs the normal benefits it brings.  In these scenarios, Calico network policy can be used to selectively bypass conntrack while still enforcing network security.  For all other traffic, conntrack continues to be your friend!

If you enjoyed this blog then you may also like:

* [Introducing the Calico eBPF dataplane](https://www.projectcalico.org/introducing-the-calico-ebpf-dataplane/) including performance benchmarking and new service routing options with source IP preservation and DSR (Direct Server Return)
* Free online training at [projectcalico.org/events](http://projectcalico.org/events) or subscribe to [Calico Essentials](https://www.tigera.io/tigera-products/calico-essentials/) for personalized training & workshops
* Learn about [Calico Enterprise](https://docs.projectcalico.org/calico-enterprise/)

**1 Introduction**

Connection tracking is the basis of many network services and applications. For example, [Kubernetes Service](https://kubernetes.io/docs/concepts/services-networking/service/), [ServiceMesh sidecar](https://istio.io/latest/docs/reference/config/networking/sidecar/), software layer 4 load balancer (L4LB) [LVS/IPVS](https://en.wikipedia.org/wiki/Linux_Virtual_Server), [Docker network](https://docs.docker.com/network/bridge/), [OpenvSwitch (OVS)](http://docs.openvswitch.org/en/latest/tutorials/ovs-conntrack/), OpenStack [security group](https://docs.openstack.org/nova/queens/admin/security-groups.html) (host firewall), etc, all rely on the functionalities of connection tracking.

**1.1 Concepts**

**Connection tracking (conntrack)**

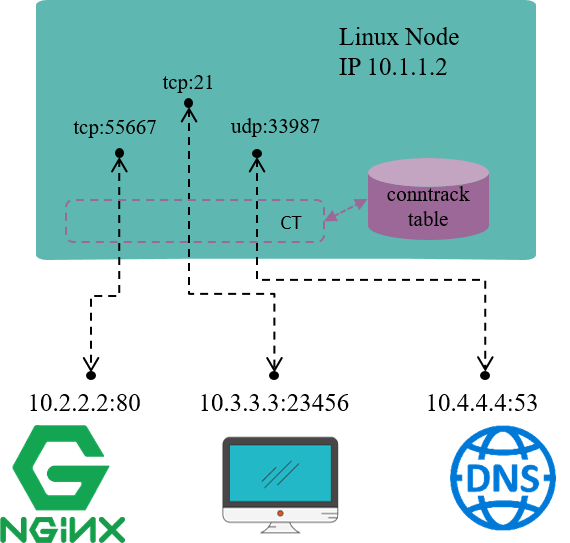


Fig 1.1 Connection tracking example on a Linux node

Connection tracking, as the name illustrates itself, **tracks (and maintains) connections’ states**.

For example, in Fig 1.1, the Linux node has an IP address 10.1.1.2, we could see 3 connections on this node:

1. 10.1.1.2:55667 <-> 10.2.2.2.:80: **locally originated** connection for accessing external HTTP/TCP service
2. 10.3.3.3:23456 <-> 10.3.3.2.:21: **externally originated** connection for accessing FTP/TCP service in this node
3. 10.1.1.2:33987 <-> 10.4.4.4.:53: **locally originated** connection for accessing external DNS/UDP service

Conntrack module’s responsibility is to **discover and record these connections and their statuses**, which include:

* Extract tuple from packets, distinguish flow and the related connection.
* Maintain a **“database”** (conntrack table) for all connections, store information such as connection’s created time, packets sent, bytes sent, etc.
* Garbage collecting (GC) stale connection info
* Serve for upper layer functionalities, e.g. as the foundation of NAT module

But note that, the term **“connection” in “connection tracking”** is different from the **“connection” concept that we mean in TCP/IP stack**. Put it simply,

* In TCP/IP stack, “connection” is a layer 4 (transport layer) concept.
  + TCP is a connection-oriented protocol, all packets need to be acknowledged (ACK), and there is retransmission mechanism.
  + UDP is a connectionless protocol, acknowledgement (ACK) is not required, no retransmission either.
* In connection tracking, a tuple uniquely defines a flow, and a flow represents a connection.
  + We will see later that UDP, or **even ICMP (layer 3 protocol) have connection entries**.
  + But **not all protocols will be connection tracked**.

When refering to the term “connection”, we mean the latter one in most cases, namely, the “connection” in “connection tracking” context.

**Network address translation (NAT)**

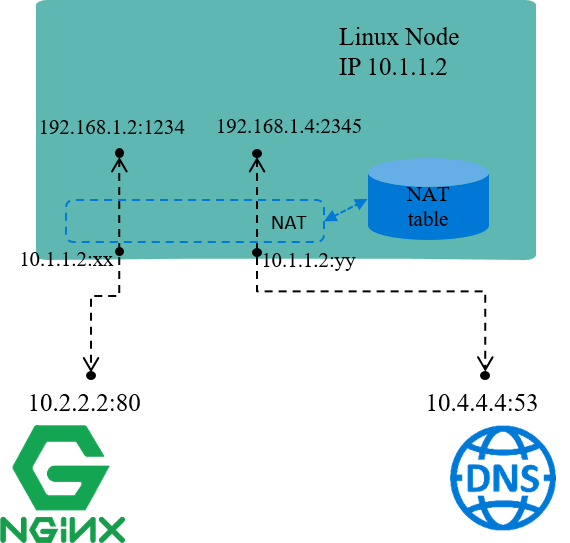


Fig 1.2 NAT for node local IP addresses

As the name illustrates, **NAT** translates (packets’) network addresses (IP + Port).

For example, in Fig 1.2, assume node IP 10.1.1.2 is reachable from other nodes, while IP addresses within network 192.168.1.0/24 is not reachable, this indicates that:

1. packets with source IP within network range 192.168.1.0/24 **could be sent out**, as egress routing only relies on destination IP.
2. but, the **reply packets** (with destination IP falls into 192.168.1.0/24) **could not come back**, as 192.168.1.0/24 is not routable within hosts.

So, one of a solution for this scenario is:

1. when packets with source IP belongs to 192.168.1.0/24 are going to be sent , the node replaces these source IP (and/or port) with its own IP 10.1.1.2 then send out.
2. when reply packets arrive, node does the reverse conversion, then forwards to the original sender.

This is the underlying working mechanism of NAT.

The default network mode of Docker, bridge network mode, uses NAT in the same way as above [4]. Within the node, each Docker container allocates a node local IP address. This address enables communications between containers within the node, but when containers communicate with services outside the node, the traffic will be NAT-ed.

NAT may replace the source ports as well. It’s easy to understand: each IP address can use the full port range (e.g. 1~65535). So assume we have two connections:

* 192.168.1.2:3333 <–> NAT <–> 10.2.2.2:80
* 192.168.1.3:3333 <–> NAT <–> 10.2.2.2:80

if NAT only replaces source IP addresses to node IP, the above two distinct connections after NAT will be:

* 10.1.1.2:3333 <–> 10.2.2.2:80
* 10.1.1.2:3333 <–> 10.2.2.2:80

which mixed into a same connection that could not be distinguished and the reverse translation will fail. So NAT also replaces source ports if collision happens.

NAT can be further categorized as:

* SNAT: do translation on source address
* DNAT: do translation on destination address
* Full NAT：do translation on both source and destination addresses

Our above examples falls into SNAT case.

**NAT relies on the results of connection tracking**, and, NAT the most important users of connection tracking.

**Layer 4 load balancing (L4 LB)**

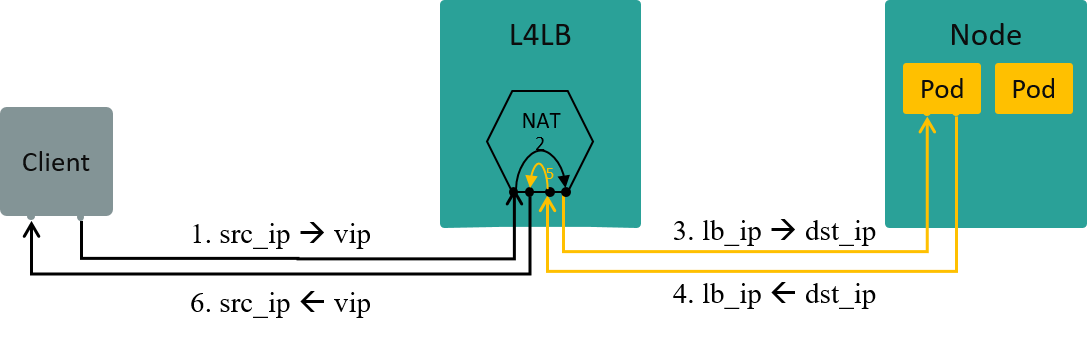


Fig 1.3 L4LB: traffic path in NAT mode [3]

Let’s enlarge our discussing scope slightly, to the topic of layer 4 load balancing **(L4LB) under NAT mode**.

L4 LB distributes traffic acorrding to packets’ L3+L4 info, e.g. src/dst ip, src/dst port, proto.

**VIP (Virtual IP)** is an mechanism/design to implement L4LB:

* Multiple backend nodes with distinct Real IPs registered to the same virtual IP (VIP)
* Traffic from clients will first arrive at VIP, then be load balanced to specific backend IPs

If L4LB uses NAT mode (between VIP and Real IPs), then L4LB will perform full NAT between client-server traffic, the data flow depicted as Fig 1.3.

**1.2 Thoery**

With the above concepts in mind, let’s reasons about the underlying theory of connction tracking.

To track the states of all connections on a node, we need to,

1. **Hook (or filter) every packet** passes through this node, and **analyze the packet**.
2. **Setup a “database”** for recoding the status of those connections (conntrack table).
3. **Update connection status timely** to database based on the extracted information from hooked packets.

For example,

1. When hooked a TCP SYNC packet, we could confirm that a new connection attempt is under the way, so we need to create a new conntrack entry to record this connection.
2. When got a packet that belongs to an existing connection, we need to update the connctrack entry statistics, e.g. bytes sent, packets sent, timeout value, e.g.
3. When no packets has matches a conntrack entry for more than 30 minutes, we need to consider deleting this entry from connection database.

Besides the above functional requirements, performance requirements also need to concern, as conntrack module needs filter and analyze every single packet. Performance considerations are fairly important, but it is beyond the scope of this post. We will refer to performance issues again when walking through the kernel conntrack implementation later.

Further, it’s better to have some management tools for faciliating the using of conntrack module.

**1.3 Design: Netfilter**

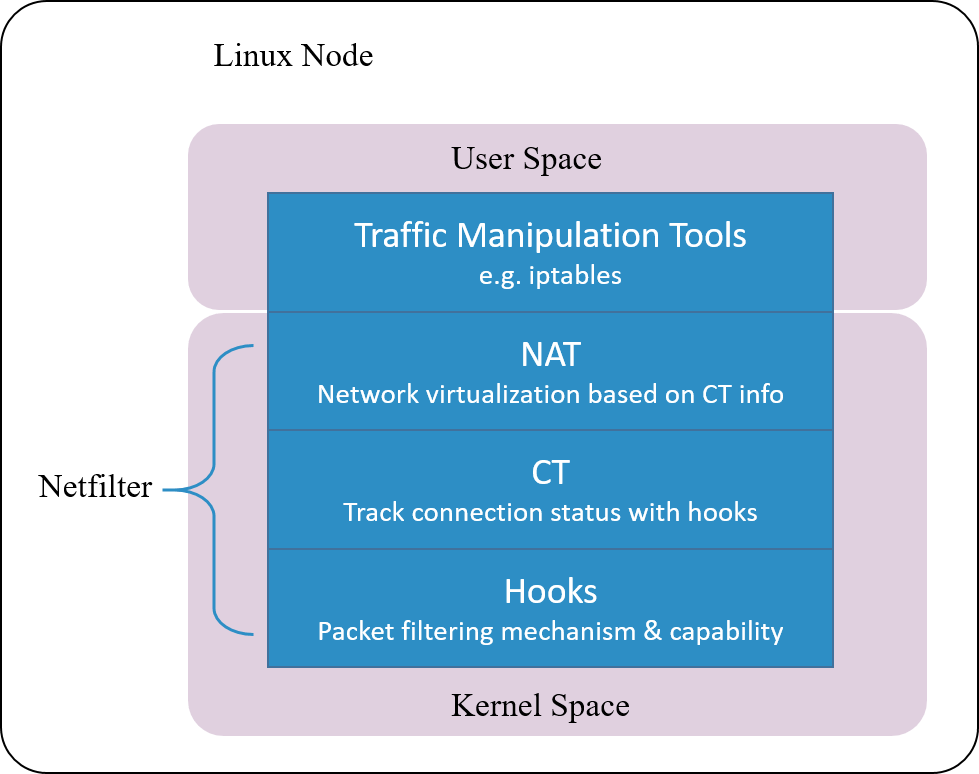


Fig 1.4 Netfilter architecture inside Linux kernel

**Linux’s connction tracking is implemented as a module in [Netfilter](https://en.wikipedia.org/wiki/Netfilter) framework.**

[Netfilter](https://en.wikipedia.org/wiki/Netfilter) is a packet manipulating and filtering framework inside Linux kernel. It provides several hooking positions inside kernel, so packet hooking, filtering and many other processings could be done.

Put it more clearly, hook is a mechanism that places several checking points in the travesal path of packets. When a packet arrives a hooking point, it first get checked, the checking result could be one of:

1. let it go: no modifications to the packet, push it back to the original travesal path and let it go
2. modify it: e.g. replace network address (NAT), then push back to the original travesal path and let it go
3. drop it: e.g. by security firewall rules configured at this checking (hooking) point

Note that conntrack module only extracts connection information and maintains its database, it does not modify or drop a packet. Modification and dropping are done by other modules, e.g. NAT.

Netfilter is one of the earliest networking frameworks inside Linux kernel, it initially got developed in 1998, and merged into 2.4.x kernel mainline in 2000.

After more than 20 years evolvement, it gets so complicated that could result to **degraded performance** in certain scenarios, we will talk a little more about this later.

**1.4 Design: further considerations**

From our discussion in section 1.2, we know that **connection tracking concept is independent from Netfilter**, the latter is only one of the implementations for connection tracking.

In other words, **as long as we have the hooking capability** - the ability to hook every single packet that goes through the system - **we could implement our own connection trakcing mechanism**.

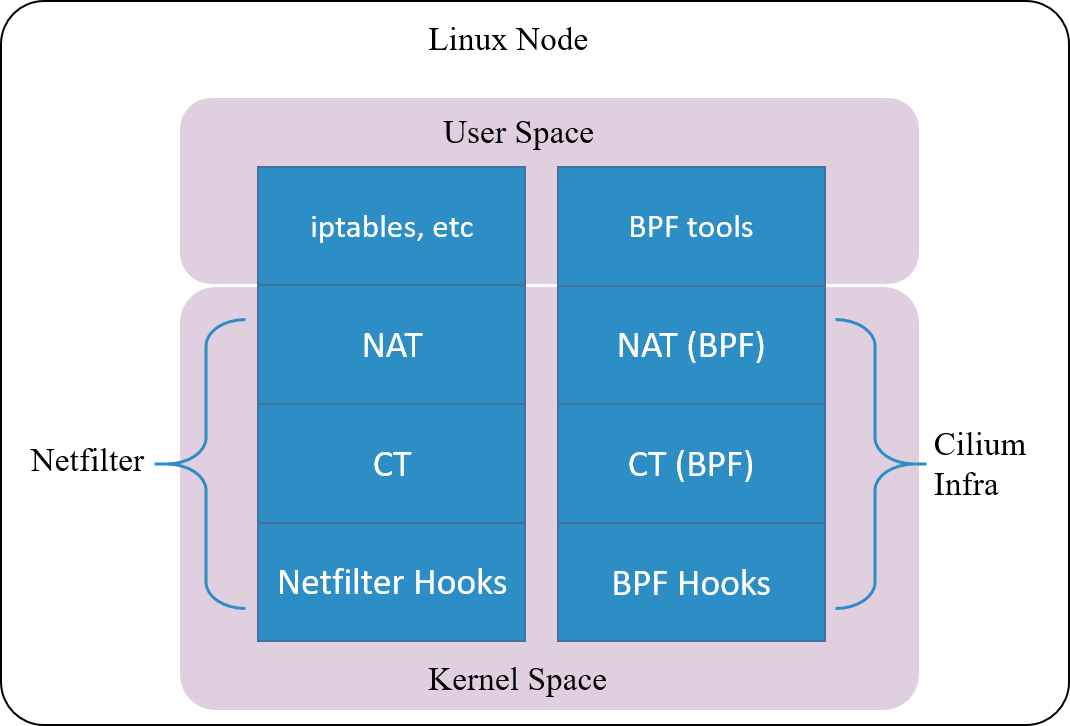


Fig 1.5. Cilium's conntrack and NAT architectrue

[Cilium](https://github.com/cilium/cilium), a cloud native networking solution for Kubernetes, implements such a conntrack and NAT mechanism. The underlyings of the impelentation:

1. Hook packets based on BPF hook points (BPF’s equivalent part of the Netfilter hooks)
2. Implement completely new conntrack and NAT modules based on BPF hooks (relies on kernel 4.19+ to be fully functionaly on itself)

So, you could even [remove the entire Netfilter module](https://github.com/cilium/cilium/issues/12879) , and Cilium will still work properly for Kubernetes functionalities such as ClusterIP, NodePort, ExternalIPs and LoadBalancer [2].

As this connction tracking mechanism is independent from Netfilter, its conntrack and NAT entries are not stored in system’s (namely, Netfilter’s) conntrack table and NAT table. So frequently used network tools conntrack/netstats/ss/lsof could not list them, you must use Cilium’s commands, e.g:

$ cilium bpf nat list

$ cilium bpf ct list global

Also, the configurations are also independent, you need to specify Cilium’s configuration parameters, such as command line argument --bpf-ct-tcp-max.

We say that conntrack module is independent from NAT module, but **for performance considerations**, their code may have certain couplings. For example, when performing GC for conntrack table, it will efficiently remove related entries in NAT table, rather than maintaining a separate GC loop for NAT table.

This ends our discussion on the theory of connection tracking, and in the following, we will dig into the kernel implementation.

**2 Implementation: Netfilter hooks**

Netfilter consists of several modules:

1. conntrack: kernel module
2. NAT: kernel module
3. iptables: userspace tools

**2.1 Netfilter framework**

**The 5 hooking points**

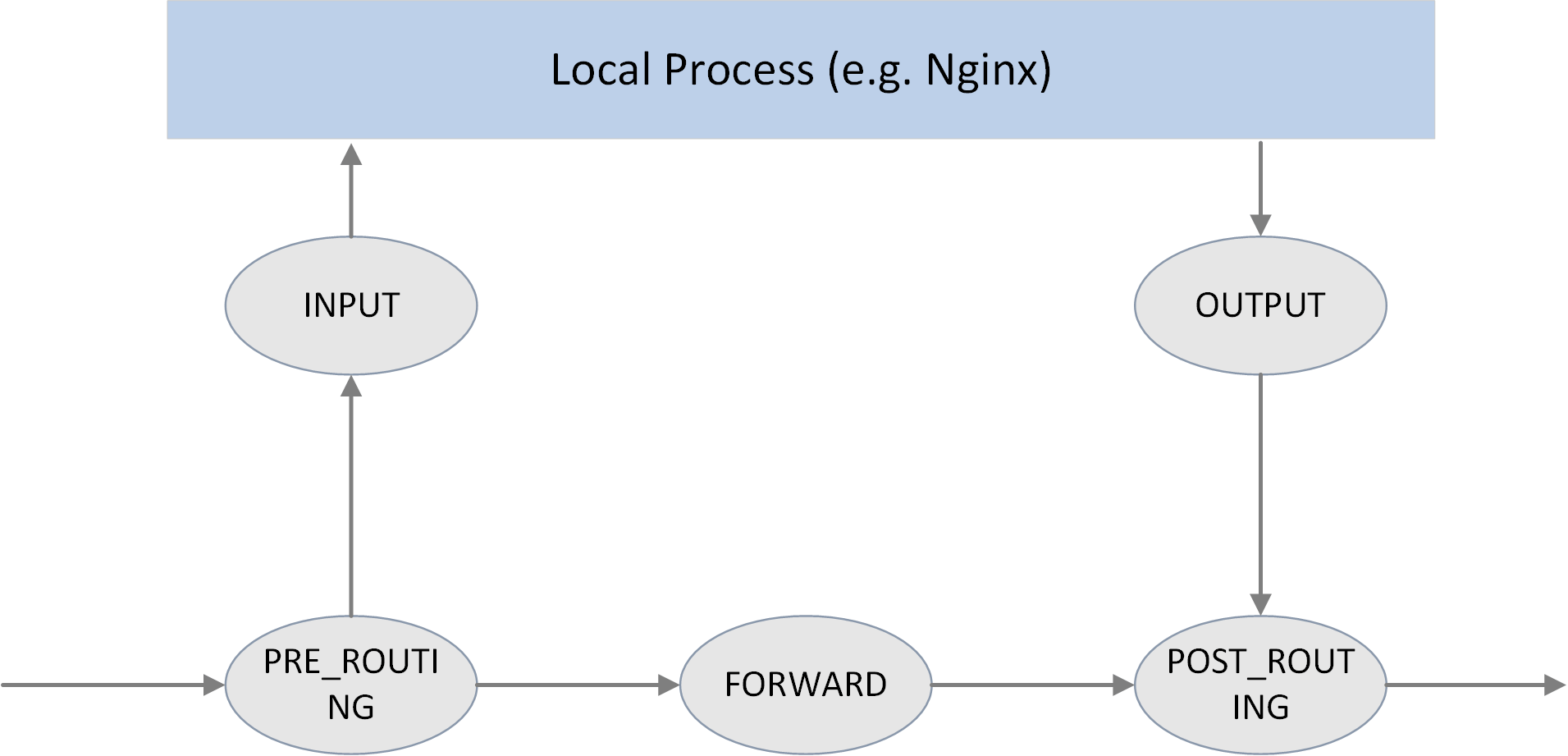


Fig 2.1 The 5 hook points in netfilter framework

As shown in the above picture, Netfilter provides 5 hooking points in the packet travesing path inside Linux kernel:

*// include/uapi/linux/netfilter\_ipv4.h*

**#define NF\_IP\_PRE\_ROUTING 0** */\* After promisc drops, checksum checks. \*/*

**#define NF\_IP\_LOCAL\_IN 1** */\* If the packet is destined for this box. \*/*

**#define NF\_IP\_FORWARD 2** */\* If the packet is destined for another interface. \*/*

**#define NF\_IP\_LOCAL\_OUT 3** */\* Packets coming from a local process. \*/*

**#define NF\_IP\_POST\_ROUTING 4** */\* Packets about to hit the wire. \*/*

**#define NF\_IP\_NUMHOOKS 5**

Users could register callback functions (handlers) at these points. When a packet arrives at the hook point, it will triger the related handlers being called.

There is another definition for these NF\_INET\_\* variables, in include/uapi/linux/netfilter.h. These two definitions are equivalent, from the comments in the code, NF\_IP\_\* are probably used for backward compatibility.

**enum** nf\_inet\_hooks {

NF\_INET\_PRE\_ROUTING,

NF\_INET\_LOCAL\_IN,

NF\_INET\_FORWARD,

NF\_INET\_LOCAL\_OUT,

NF\_INET\_POST\_ROUTING,

NF\_INET\_NUMHOOKS

};

**Hook handler return values**

hook handlers return a verdict result after checking against a given packet, and this verdict will guide next processings on this packet. Possible verdict results include:

*// include/uapi/linux/netfilter.h*

**#define NF\_DROP 0 // the packet has been dropped this packet in handler**

**#define NF\_ACCEPT 1 // the packet is no dropped, continue following processing**

**#define NF\_STOLEN 2 // the packet has been consumed by the handler, no further processing is needed**

**#define NF\_QUEUE 3 // the packet should be pushed into queue**

**#define NF\_REPEAT 4 // current handler should be called again against the packet**

**Hook handler priorities**

Multiple handlers could be registered to the same hooking point.

When register a handler, a priority parameter must be provided. So that when a packet arrives at this point, the system could call these handlers in order with their priorities.

**2.2 Filtering rules management**

iptables is the userspace tool for configuring Netfilter hooking capabilities. To faciliate management, it splits rules into several tables by functionalities:

* raw
* filter
* nat
* mangle

This is not the focus of this post, more information on this, refer to [A Deep Dive into iptables and Netfilter Architecture](https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-architecture).

**3 Implementation: Netfilter conntrack**

conntrack module traces the **connection status of trackable protocols** [1]. That is, connection tracking targets at **specific protocols**, not all protocols. We will see later what protocols it supports.

**3.1 Data structures and functions**

Key data structures:

* struct nf\_conntrack\_tuple {}: **defines a tuple**.
  + struct nf\_conntrack\_man {}: the manipulatable part of a tuple
    - struct nf\_conntrack\_man\_proto {}: the protocol specific part in tuple’s manipulatable part
* struct nf\_conntrack\_l4proto {}: a collection of **methods** a trackable protocol needs to implement (and other fields).
* struct nf\_conntrack\_tuple\_hash {}: **defines a conntrack entry** (value) stored in hash table (conntrack table), hash key is a uint32 integer computed from tuple info.
* struct nf\_conn {}: **defines a flow**.

Key functions:

* hash\_conntrack\_raw(): calculates a 32bit hash key from tuple info.
* nf\_conntrack\_in()：**core of the conntrack module, the entrypoint of connection tracking**.
* resolve\_normal\_ct() -> init\_conntrack() -> l4proto->new(): creates a new conntrack entry.
* nf\_conntrack\_confirm(): confirms the new connection that previously created via nf\_conntrack\_in().

**3.2 struct nf\_conntrack\_tuple {}: Tuple**

Tuple is one of the most important concepts in connection tracking.

**A tuple uniquely defines a unidirectional flow**, this is clearly explained in kernel code comments:

//include/net/netfilter/nf\_conntrack\_tuple.h

A tuple is a structure containing the information to uniquely identify a connection. ie. if two packets have the same tuple, they are in the same connection; if not, they are not.

**Data structure definitions**

To facilite NAT module’s implementation, kernel splits the tuple structure into “manipulatable” and “non-manipulatable” parts.

In the following code, \_man is short for manipulatable, which is a bad naming/abbreviating from today’s point of view.

// include/net/netfilter/nf\_conntrack\_tuple.h

// ude/uapi/linux/netfilter.h

union nf\_inet\_addr {

\_\_u32 all[4];

\_\_be32 ip;

\_\_be32 ip6[4];

struct in\_addr in;

struct in6\_addr in6;

/\* manipulatable part of the tuple \*/ / };

struct nf\_conntrack\_man { /

union nf\_inet\_addr u3; -->--/

union nf\_conntrack\_man\_proto u; -->--\

\ // include/uapi/linux/netfilter/nf\_conntrack\_tuple\_common.h

u\_int16\_t l3num; // L3 proto \ // protocol specific part

}; union nf\_conntrack\_man\_proto {

\_\_be16 all;/\* Add other protocols here. \*/

struct { \_\_be16 port; } tcp;

struct { \_\_be16 port; } udp;

struct { \_\_be16 id; } icmp;

struct { \_\_be16 port; } dccp;

struct { \_\_be16 port; } sctp;

struct { \_\_be16 key; } gre;

};

struct nf\_conntrack\_tuple { /\* This contains the information to distinguish a connection. \*/

struct nf\_conntrack\_man src; // source address info，manipulatable part

struct {

union nf\_inet\_addr u3;

union {

\_\_be16 all; // Add other protocols here

struct { \_\_be16 port; } tcp;

struct { \_\_be16 port; } udp;

struct { u\_int8\_t type, code; } icmp;

struct { \_\_be16 port; } dccp;

struct { \_\_be16 port; } sctp;

struct { \_\_be16 key; } gre;

} u;

u\_int8\_t protonum; // The protocol

u\_int8\_t dir; // The direction (for tuplehash)

} dst; // destination address info

};

There are only 2 fields (src and dst) inside struct nf\_conntrack\_tuple {}, each stores source and destination address information.

But src and dst are themselves also structs, storing protocol specific data. Take IPv4 UDP as example, information of the 5-tuple stores in the following fields:

* dst.protonum: protocol type (IPPROTO\_UDP)
* src.u3.ip: source IP address
* dst.u3.ip: destination IP address
* src.u.udp.port: source UDP port number
* dst.u.udp.port: destination UDP port number

**Connection-trackable protocols**

From the above code, we could see that **only 6 protocols support connection tracking currently**: TCP, UDP, ICMP, DCCP, SCTP, GRE.

**Pay attention to the ICMP protocol**. People may think that connction tracking is done by hashing over L3+L4 headers of packets, while ICMP is a L3 protocol, so it could not be conntrack-ed. But actually it could be, from the above code, we see that the **type and code fields in ICMP header** are used for defining a tuple and performing subsequent hashing.

**3.3 struct nf\_conntrack\_l4proto {}: methods trackable protocols need to implement**

Protocols that support connection tracking need to implement the methods defined in struct nf\_conntrack\_l4proto {}, for example pkt\_to\_tuple(), which extracts tuple information from given packet’s L3/L4 header.

*// include/net/netfilter/nf\_conntrack\_l4proto.h*

**struct** nf\_conntrack\_l4proto {

u\_int16\_t l3proto; */\* L3 Protocol number. \*/*

u\_int8\_t l4proto; */\* L4 Protocol number. \*/*

*// extract tuple info from given packet (skb)*

bool (**\***pkt\_to\_tuple)(**struct** sk\_buff **\***skb, ... **struct** nf\_conntrack\_tuple **\***tuple);

*// returns verdict for packet*

**int** (**\***packet)(**struct** nf\_conn **\***ct, **const** **struct** sk\_buff **\***skb ...);

*// create a new conntrack, return TRUE if succeeds.*

*// if returns TRUE, packet() method will be called against this skb later*

bool (**\***new)(**struct** nf\_conn **\***ct, **const** **struct** sk\_buff **\***skb, **unsigned** **int** dataoff);

*// determin if this packet could be conntrack-ed.*

*// if could, packet() method will be called against this skb later*

**int** (**\***error)(**struct** net **\***net, **struct** nf\_conn **\***tmpl, **struct** sk\_buff **\***skb, ...);

...

};

**3.4 struct nf\_conntrack\_tuple\_hash {}: conntrack entry**

conntrack modules stores active connections in a hash table:

* key: 32bit value calculated from tuple info
* value: conntrack entry (struct nf\_conntrack\_tuple\_hash {})

Method hash\_conntrack\_raw() calculates a 32bit hash key from tuple info:

*// net/netfilter/nf\_conntrack\_core.c*

**static** u32 **hash\_conntrack\_raw**(**struct** nf\_conntrack\_tuple **\***tuple, **struct** net **\***net)

{

get\_random\_once(**&**nf\_conntrack\_hash\_rnd, **sizeof**(nf\_conntrack\_hash\_rnd));

*/\* The direction must be ignored, so we hash everything up to the*

*\* destination ports (which is a multiple of 4) and treat the last three bytes manually. \*/*

u32 seed **=** nf\_conntrack\_hash\_rnd **^** net\_hash\_mix(net);

**unsigned** **int** n **=** (**sizeof**(tuple**->**src) **+** **sizeof**(tuple**->**dst.u3)) **/** **sizeof**(u32);

**return** jhash2((u32 **\***)tuple, n, seed **^** ((tuple**->**dst.u.all **<<** 16) **|** tuple**->**dst.protonum));

}

Pay attention to the calculating logic, and how source and destination address fields are used for the final hash value.

Conntrack entry is defined as struct nf\_conntrack\_tuple\_hash {}:

*// include/net/netfilter/nf\_conntrack\_tuple.h*

*// Connections have two entries in the hash table: one for each way*

**struct** nf\_conntrack\_tuple\_hash {

**struct** hlist\_nulls\_node hnnode; *// point to the related connection `struct nf\_conn`,*

*// list for fixing hash collisions*

**struct** nf\_conntrack\_tuple tuple;

};

**3.5 struct nf\_conn {}: connection**

Each flow in Netfilter is called a connection, even for those connectionless protocols (e.g. UDP). A connection is defined as struct nf\_conn {}, with important fields as follows:

*// include/net/netfilter/nf\_conntrack.h*

*// include/linux/skbuff.h*

**------>** **struct** nf\_conntrack {

**|** atomic\_t use; *// refcount?*

**|** };

**struct** nf\_conn { **|**

**struct** nf\_conntrack ct\_general;

**struct** nf\_conntrack\_tuple\_hash tuplehash[IP\_CT\_DIR\_MAX]; *// conntrack entry, array for ingress/egress flows*

**unsigned** **long** status; *// connection status, see below for detailed status list*

u32 timeout; *// timer for connection status*

possible\_net\_t ct\_net;

**struct** hlist\_node nat\_bysource;

*// per conntrack: protocol private data*

**struct** nf\_conn **\***master; **union** nf\_conntrack\_proto {

*/\* insert conntrack proto private data here \*/*

u\_int32\_t mark; */\* mark skb \*/* **struct** nf\_ct\_dccp dccp;

u\_int32\_t secmark; **struct** ip\_ct\_sctp sctp;

**struct** ip\_ct\_tcp tcp;

**union** nf\_conntrack\_proto proto; **---------->----->** **struct** nf\_ct\_gre gre;

}; **unsigned** **int** tmpl\_padto;

};

All possible status of a connection, enum ip\_conntrack\_status：

*// include/uapi/linux/netfilter/nf\_conntrack\_common.h*

**enum** ip\_conntrack\_status {

IPS\_EXPECTED **=** (1 **<<** IPS\_EXPECTED\_BIT),

IPS\_SEEN\_REPLY **=** (1 **<<** IPS\_SEEN\_REPLY\_BIT),

IPS\_ASSURED **=** (1 **<<** IPS\_ASSURED\_BIT),

IPS\_CONFIRMED **=** (1 **<<** IPS\_CONFIRMED\_BIT),

IPS\_SRC\_NAT **=** (1 **<<** IPS\_SRC\_NAT\_BIT),

IPS\_DST\_NAT **=** (1 **<<** IPS\_DST\_NAT\_BIT),

IPS\_NAT\_MASK **=** (IPS\_DST\_NAT **|** IPS\_SRC\_NAT),

IPS\_SEQ\_ADJUST **=** (1 **<<** IPS\_SEQ\_ADJUST\_BIT),

IPS\_SRC\_NAT\_DONE **=** (1 **<<** IPS\_SRC\_NAT\_DONE\_BIT),

IPS\_DST\_NAT\_DONE **=** (1 **<<** IPS\_DST\_NAT\_DONE\_BIT),

IPS\_NAT\_DONE\_MASK **=** (IPS\_DST\_NAT\_DONE **|** IPS\_SRC\_NAT\_DONE),

IPS\_DYING **=** (1 **<<** IPS\_DYING\_BIT),

IPS\_FIXED\_TIMEOUT **=** (1 **<<** IPS\_FIXED\_TIMEOUT\_BIT),

IPS\_TEMPLATE **=** (1 **<<** IPS\_TEMPLATE\_BIT),

IPS\_UNTRACKED **=** (1 **<<** IPS\_UNTRACKED\_BIT),

IPS\_HELPER **=** (1 **<<** IPS\_HELPER\_BIT),

IPS\_OFFLOAD **=** (1 **<<** IPS\_OFFLOAD\_BIT),

IPS\_UNCHANGEABLE\_MASK **=** (IPS\_NAT\_DONE\_MASK **|** IPS\_NAT\_MASK **|**

IPS\_EXPECTED **|** IPS\_CONFIRMED **|** IPS\_DYING **|**

IPS\_SEQ\_ADJUST **|** IPS\_TEMPLATE **|** IPS\_OFFLOAD),

};

**3.6 nf\_conntrack\_in(): enter conntrack**

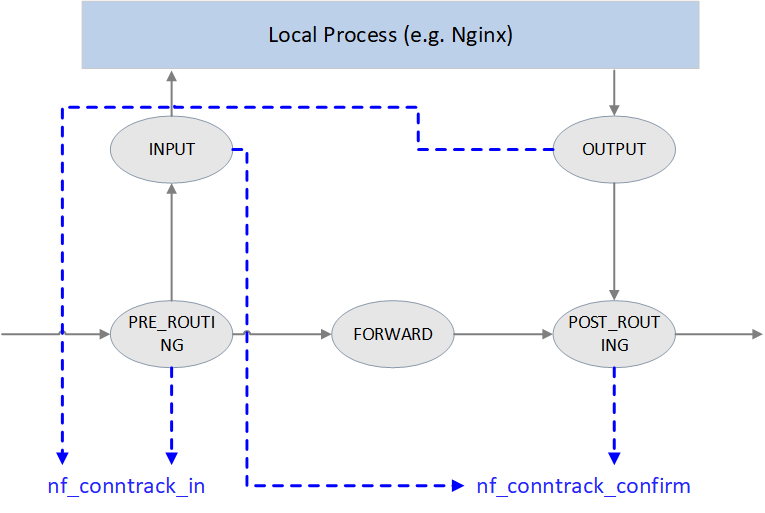


Fig 3.1 Conntrack points in Netfilter framework

As illustrated in Fig 3.1, Netfilter performs connection tracking at 4 hooking positions:

1. PRE\_ROUTING and LOCAL\_OUT

**Start connection tracking** by calling nf\_conntrack\_in(), normally this will create a new conntrack entry, then insert it into **unconfirmed list**.

Why starting at these two places? Because both of them are the **earliest points that the initial packet of a new connection arrives Netfilter framework**.

* + PRE\_ROUTING: earliest point that the first packet of a new **externally-initiated** connection arrives.
  + LOCAL\_OUT: earliest point that the first packet of a new **locally-initiated** connection arrives.

1. POST\_ROUTING 和 LOCAL\_IN

Call nf\_conntrack\_confirm(), move the newly created (in previous nf\_conntrack\_in()) connection in unconfirmed list to **confirmed list**.

Again, why these two hooking points? It is because if the first packet of a new connection is not dropped during internal processing, it should arrive these places and these are **the last place before it leaving Netfilter framework**:

* + For packet of **externally-initiated** connection, LOCAL\_IN is the last hooking point before it is sent to upper layer application (e.g. a Nginx process).
  + For packet of **locally-initiated** connection, POST\_ROUTING is the last hooking point before this packet is sent out on the wire.

We could see how these handlers get registered into the netfilter framwork:

*// net/netfilter/nf\_conntrack\_proto.c*

*/\* Connection tracking may drop packets, but never alters them, so make it the first hook. \*/*

**static** **const** **struct** nf\_hook\_ops ipv4\_conntrack\_ops[] **=** {

{

.hook **=** ipv4\_conntrack\_in, *// enter conntrack by calling nf\_conntrack\_in()*

.pf **=** NFPROTO\_IPV4,

.hooknum **=** NF\_INET\_PRE\_ROUTING, *// PRE\_ROUTING hook point*

.priority **=** NF\_IP\_PRI\_CONNTRACK,

},

{

.hook **=** ipv4\_conntrack\_local, *// enter conntrack by calling nf\_conntrack\_in()*

.pf **=** NFPROTO\_IPV4,

.hooknum **=** NF\_INET\_LOCAL\_OUT, *// LOCAL\_OUT hook point*

.priority **=** NF\_IP\_PRI\_CONNTRACK,

},

{

.hook **=** ipv4\_confirm, *// call nf\_conntrack\_confirm()*

.pf **=** NFPROTO\_IPV4,

.hooknum **=** NF\_INET\_POST\_ROUTING, *// POST\_ROUTING hook point*

.priority **=** NF\_IP\_PRI\_CONNTRACK\_CONFIRM,

},

{

.hook **=** ipv4\_confirm, *// call nf\_conntrack\_confirm()*

.pf **=** NFPROTO\_IPV4,

.hooknum **=** NF\_INET\_LOCAL\_IN, *// LOCAL\_IN hook point*

.priority **=** NF\_IP\_PRI\_CONNTRACK\_CONFIRM,

},

};

Method **nf\_conntrack\_in() is the core of connection tracking module**.

*// net/netfilter/nf\_conntrack\_core.c*

**unsigned** **int**

**nf\_conntrack\_in**(**struct** net **\***net, u\_int8\_t pf, **unsigned** **int** hooknum, **struct** sk\_buff **\***skb)

{

**struct** nf\_conn **\***tmpl **=** nf\_ct\_get(skb, **&**ctinfo); *// get related conntrack\_info and conntrack entry*

**if** (tmpl **||** ctinfo **==** IP\_CT\_UNTRACKED) { *// if conntrack exists, or if is un-trackable protocols*

**if** ((tmpl **&&** **!**nf\_ct\_is\_template(tmpl)) **||** ctinfo **==** IP\_CT\_UNTRACKED) {

NF\_CT\_STAT\_INC\_ATOMIC(net, ignore); *// no need to conntrack, inc ignore stats*

**return** NF\_ACCEPT; *// return NF\_ACCEPT, continue normal processing*

}

skb**->**\_nfct **=** 0; *// need to conntrack, reset refcount of this packet,*

} *// prepare for further processing*

**struct** nf\_conntrack\_l4proto

**\***l4proto **=** \_\_nf\_ct\_l4proto\_find(); *// extract protocol-specific L4 info*

**if** (l4proto**->**error(tmpl, skb, pf) **<=** 0) { *// skb validation*

NF\_CT\_STAT\_INC\_ATOMIC(net, error);

NF\_CT\_STAT\_INC\_ATOMIC(net, invalid);

**goto** out;

}

**repeat:**

resolve\_normal\_ct(net, tmpl, skb, ... l4proto); *// start conntrack, extract tuple:*

*// create new conntrack or update existing one*

l4proto**->**packet(ct, skb, ctinfo); *// protocol-specific processing, e.g. update UDP timeout*

**if** (ctinfo **==** IP\_CT\_ESTABLISHED\_REPLY **&&** **!**test\_and\_set\_bit(IPS\_SEEN\_REPLY\_BIT, **&**ct**->**status))

nf\_conntrack\_event\_cache(IPCT\_REPLY, ct);

**out:**

**if** (tmpl)

nf\_ct\_put(tmpl); *// un-reference tmpl*

}

Rough processing steps:

1. Get conntrack info of this skb.
2. Determine if conntrack is needed for this skb. If not needed, update ignore counter, return NF\_ACCEPT; if needed, init (reset) this skb’s refcount.
3. Extract L4 information from skb, init protocol-specific struct nf\_conntrack\_l4proto {} variable, which contains the protocol’s callback methods for performing connection tracking.
4. Call protocol-specific error() method for data validation, e.g. checksum.
5. **Start conntrack** by calling resolve\_normal\_ct() method, it will create new tuple, new conntrack entry, or update status of existing conntrack entry.
6. Call packet() method for some protocol-specific processing. E.g. for UDP, if IPS\_SEEN\_REPLY is set in status, it will update timeout value. timeout varies according to different protocols, the smaller it is, the more it will be stronger on anti-DDos attacks (DDos attacks a system by exausting all the system’s connections).

**3.7 init\_conntrack(): create new conntrack entry**

If connection does not exist yet (the first packet of a flow), resolve\_normal\_ct() method will call init\_conntrack(), and the latter will further call protocol-specific method new() to create a new conntrack entry.

*// include/net/netfilter/nf\_conntrack\_core.c*

*// Allocate a new conntrack*

**static** noinline **struct** nf\_conntrack\_tuple\_hash **\***

**init\_conntrack**(**struct** net **\***net, **struct** nf\_conn **\***tmpl,

**const** **struct** nf\_conntrack\_tuple **\***tuple,

**const** **struct** nf\_conntrack\_l4proto **\***l4proto,

**struct** sk\_buff **\***skb, **unsigned** **int** dataoff, u32 hash)

{

**struct** nf\_conn **\***ct;

ct **=** \_\_nf\_conntrack\_alloc(net, zone, tuple, **&**repl\_tuple, GFP\_ATOMIC, hash);

l4proto**->**new(ct, skb, dataoff); *// protocol-specific method for creating a conntrack entry*

local\_bh\_disable(); *// disable softirq*

**if** (net**->**ct.expect\_count) {

exp **=** nf\_ct\_find\_expectation(net, zone, tuple);

**if** (exp) {

\_\_set\_bit(IPS\_EXPECTED\_BIT, **&**ct**->**status);

*/\* exp->master safe, refcnt bumped in nf\_ct\_find\_expectation \*/*

ct**->**master **=** exp**->**master;

ct**->**mark **=** exp**->**master**->**mark;

ct**->**secmark **=** exp**->**master**->**secmark;

NF\_CT\_STAT\_INC(net, expect\_new);

}

}

*/\* Now it is inserted into the unconfirmed list, bump refcount \*/*

nf\_conntrack\_get(**&**ct**->**ct\_general);

nf\_ct\_add\_to\_unconfirmed\_list(ct);

local\_bh\_enable(); *// re-enable softirq*

**if** (exp) {

**if** (exp**->**expectfn)

exp**->**expectfn(ct, exp);

nf\_ct\_expect\_put(exp);

}

**return** **&**ct**->**tuplehash[IP\_CT\_DIR\_ORIGINAL];

}

Implementations of protocol-specific new() method, see net/netfilter/nf\_conntrack\_proto\_\*.c.

If current packet will influence the status of subsequent packets, init\_conntrack() will set the master field in struct nf\_conn. Connection oriented protocols (e.g. TCP) uses this feature.

**3.8 nf\_conntrack\_confirm(): confirm a new connection**

The new conntrack entry that nf\_conntrack\_in() creates will be inserted into an **unconfirmed connection list**.

If this packet is not dropped during intermediate processing, then when it arrives POST\_ROUTING hook, it will be further processed by nf\_conntrack\_confirm() method. We have analyzed why it is further processed here instead of other hooking points.

After nf\_conntrack\_confirm() is done, status of the connection will turn to IPS\_CONFIRMED, and the conntrack will be move from unconfirmed list to **confirmed list**.

Why bother to split the conntrack creating process into two stages (new and confirm)?

It is because after the initial packet passes nf\_conntrack\_in(), but before it arrives nf\_conntrack\_confirm(), it is possible that the packet get dropped by the kernel somewhere in the middle. This may result in large half-connected conntrack entries, and it would be a big concern in terms of both performance and security. Spliting into two steps will significantly accelerate the GC process.

*// include/net/netfilter/nf\_conntrack\_core.h*

*/\* Confirm a connection: returns NF\_DROP if packet must be dropped. \*/*

**static** **inline** **int** **nf\_conntrack\_confirm**(**struct** sk\_buff **\***skb)

{

**struct** nf\_conn **\***ct **=** (**struct** nf\_conn **\***)skb\_nfct(skb);

**int** ret **=** NF\_ACCEPT;

**if** (ct) {

**if** (**!**nf\_ct\_is\_confirmed(ct))

ret **=** \_\_nf\_conntrack\_confirm(skb);

**if** (likely(ret **==** NF\_ACCEPT))

nf\_ct\_deliver\_cached\_events(ct);

}

**return** ret;

}

confirm logic, error handling code omitted:

*// net/netfilter/nf\_conntrack\_core.c*

*/\* Confirm a connection given skb; places it in hash table \*/*

**int**

**\_\_nf\_conntrack\_confirm**(**struct** sk\_buff **\***skb)

{

**struct** nf\_conn **\***ct;

ct **=** nf\_ct\_get(skb, **&**ctinfo);

local\_bh\_disable(); *// disable softirq*

hash **=** **\***(**unsigned** **long** **\***)**&**ct**->**tuplehash[IP\_CT\_DIR\_REPLY].hnnode.pprev;

reply\_hash **=** hash\_conntrack(net, **&**ct**->**tuplehash[IP\_CT\_DIR\_REPLY].tuple);

ct**->**timeout **+=** nfct\_time\_stamp; *// update timer, will be GC-ed after timeout*

atomic\_inc(**&**ct**->**ct\_general.use); *// update conntrack entry refcount?*

ct**->**status **|=** IPS\_CONFIRMED; *// set status as `confirmed`*

\_\_nf\_conntrack\_hash\_insert(ct, hash, reply\_hash); *// insert into conntrack table*

local\_bh\_enable(); *// re-enable softirq*

nf\_conntrack\_event\_cache(master\_ct(ct) **?** IPCT\_RELATED **:** IPCT\_NEW, ct);

**return** NF\_ACCEPT;

}

One thing needs to be noted here: we could see softirq (soft interrupts) is disabled/re-enabled at many places; besides, there are lots of lock/unlock operations (omitted in the above code). This may be the main reasons of degraded performance in certain scenarios (e.g. massive concurrent short-time connections)?

**4 Implementation: Netfilter NAT**

NAT is a function module built upon conntrack module, it relies on connection tracking’s results to work properly.

Again, not all protocols supports NAT.

**4.1 Data structures and functions**

**Data structures:**

Protocols that support NAT needs to implement the methods defined in:

* struct nf\_nat\_l3proto {}
* struct nf\_nat\_l4proto {}

**Functions:**

* nf\_nat\_inet\_fn(): core of NAT module, will be called at all hooking points except NF\_INET\_FORWARD.

**4.2 NAT module init**

*// net/netfilter/nf\_nat\_core.c*

**static** **struct** nf\_nat\_hook nat\_hook **=** {

.parse\_nat\_setup **=** nfnetlink\_parse\_nat\_setup,

.decode\_session **=** \_\_nf\_nat\_decode\_session,

.manip\_pkt **=** nf\_nat\_manip\_pkt,

};

**static** **int** \_\_init **nf\_nat\_init**(**void**)

{

nf\_nat\_bysource **=** nf\_ct\_alloc\_hashtable(**&**nf\_nat\_htable\_size, 0);

nf\_ct\_helper\_expectfn\_register(**&**follow\_master\_nat);

RCU\_INIT\_POINTER(nf\_nat\_hook, **&**nat\_hook);

}

MODULE\_LICENSE("GPL");

module\_init(nf\_nat\_init);

**4.3 struct nf\_nat\_l3proto {}: protocol specific methods**

*// include/net/netfilter/nf\_nat\_l3proto.h*

**struct** nf\_nat\_l3proto {

u8 l3proto; *// e.g. AF\_INET*

u32 (**\***secure\_port )(**const** **struct** nf\_conntrack\_tuple **\***t, \_\_be16);

bool (**\***manip\_pkt )(**struct** sk\_buff **\***skb, ...);

**void** (**\***csum\_update )(**struct** sk\_buff **\***skb, ...);

**void** (**\***csum\_recalc )(**struct** sk\_buff **\***skb, u8 proto, ...);

**void** (**\***decode\_session )(**struct** sk\_buff **\***skb, ...);

**int** (**\***nlattr\_to\_range)(**struct** nlattr **\***tb[], **struct** nf\_nat\_range2 **\***range);

};

**4.4 struct nf\_nat\_l4proto {}: protocol specific methods**

manip is the abbraviation of manipulate in the code:

*// include/net/netfilter/nf\_nat\_l4proto.h*

**struct** nf\_nat\_l4proto {

u8 l4proto; *// L4 proto id, e.g. IPPROTO\_UDP, IPPROTO\_TCP*

*// Modify L3/L4 header according to the given tuple and NAT type (SNAT/DNAT)*

bool (**\***manip\_pkt)(**struct** sk\_buff **\***skb, **\***l3proto, **\***tuple, maniptype);

*// Create a unique tuple*

*// e.g. for UDP, will generate a 16bit dst\_port with src\_ip, dst\_ip, src\_port and a rand*

**void** (**\***unique\_tuple)(**\***l3proto, tuple, **struct** nf\_nat\_range2 **\***range, maniptype, **struct** nf\_conn **\***ct);

*// If the address range is exhausted the NAT modules will begin to drop packets.*

**int** (**\***nlattr\_to\_range)(**struct** nlattr **\***tb[], **struct** nf\_nat\_range2 **\***range);

};

Implementations of these methods, see net/netfilter/nf\_nat\_proto\_\*.c. For example, the TCP’s implementation:

*// net/netfilter/nf\_nat\_proto\_tcp.c*

**const** **struct** nf\_nat\_l4proto nf\_nat\_l4proto\_tcp **=** {

.l4proto **=** IPPROTO\_TCP,

.manip\_pkt **=** tcp\_manip\_pkt,

.in\_range **=** nf\_nat\_l4proto\_in\_range,

.unique\_tuple **=** tcp\_unique\_tuple,

.nlattr\_to\_range **=** nf\_nat\_l4proto\_nlattr\_to\_range,

};

**4.5 nf\_nat\_inet\_fn(): enter NAT**

nf\_nat\_inet\_fn() will be called in following hooking points:

* NF\_INET\_PRE\_ROUTING
* NF\_INET\_POST\_ROUTING
* NF\_INET\_LOCAL\_OUT
* NF\_INET\_LOCAL\_IN

namely, all Netfilter hooking points except NF\_INET\_FORWARD.

Priorities at these hooking points: **Conntrack > NAT > Packet Filtering**.

**conntrack has a higher priority than NAT, since NAT relies on the results of connection tracking**.

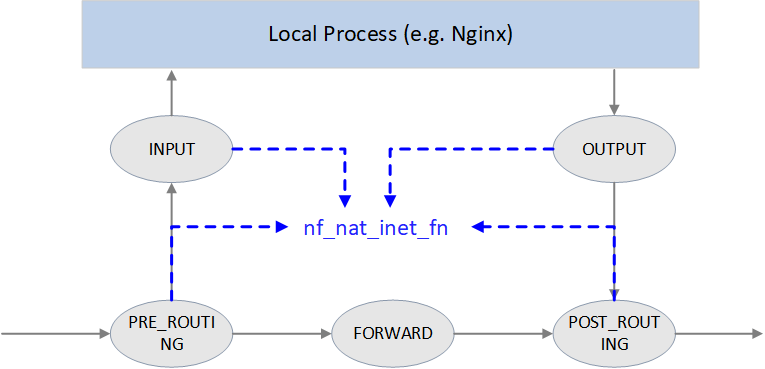


Fig. NAT

**unsigned** **int**

**nf\_nat\_inet\_fn**(**void** **\***priv, **struct** sk\_buff **\***skb, **const** **struct** nf\_hook\_state **\***state)

{

ct **=** nf\_ct\_get(skb, **&**ctinfo);

**if** (**!**ct) *// exit NAT if conntrack not exist. This is why we say NAT relies on conntrack's results*

**return** NF\_ACCEPT;

nat **=** nfct\_nat(ct);

**switch** (ctinfo) {

**case** IP\_CT\_RELATED:

**case** IP\_CT\_RELATED\_REPLY: */\* Only ICMPs can be IP\_CT\_IS\_REPLY. Fallthrough \*/*

**case** IP\_CT\_NEW: */\* Seen it before? This can happen for loopback, retrans, or local packets. \*/*

**if** (**!**nf\_nat\_initialized(ct, maniptype)) {

**struct** nf\_hook\_entries **\***e **=** rcu\_dereference(lpriv**->**entries); *// obtain all NAT rules*

**if** (**!**e)

**goto** null\_bind;

**for** (i **=** 0; i **<** e**->**num\_hook\_entries; i**++**) { *// execute NAT rules in order*

**if** (e**->**hooks[i].hook(e**->**hooks[i].priv, skb, state) **!=** NF\_ACCEPT )

**return** ret; *// return if any rule returns non ACCEPT verdict*

**if** (nf\_nat\_initialized(ct, maniptype))

**goto** do\_nat;

}

**null\_bind:**

nf\_nat\_alloc\_null\_binding(ct, state**->**hook);

} **else** { *// Already setup manip*

**if** (nf\_nat\_oif\_changed(state**->**hook, ctinfo, nat, state**->**out))

**goto** oif\_changed;

}

**break**;

**default:** */\* ESTABLISHED \*/*

**if** (nf\_nat\_oif\_changed(state**->**hook, ctinfo, nat, state**->**out))

**goto** oif\_changed;

}

**do\_nat:**

**return** nf\_nat\_packet(ct, ctinfo, state**->**hook, skb);

**oif\_changed:**

nf\_ct\_kill\_acct(ct, ctinfo, skb);

**return** NF\_DROP;

}

It first queries conntrack info for this packet, if conntrack info not exists, it means this connection could not be tracked, then we could never perform NAT for it. So just exit NAT in this case.

If conntrack info exists, and the connection is in IP\_CT\_RELATED or IP\_CT\_RELATED\_REPLY or IP\_CT\_NEW states, then get all NAT rules.

If found, execute nf\_nat\_packet() method, it will further call protocol-specific manip\_pkt method to modify the packet. If failed, the packet will be dropped.

**Masquerade**

NAT module often configured in this fashion: change IP1 to IP2 if matching XXX。

There is also another fashion for **SNAT**, called masquerade: change IP1 to dev1's IP if matching XXX.

Masquerade differentiates itself from SNAT in that when device’s IP address changes, the rules still valid. It could be seen as dynamic SNAT (dynamically adapting to the source IP changes in SNAT rules).

The drawback of masquerade is that it has degraded performance compared with SNAT, and this is easy to understand.

**4.6 nf\_nat\_packet(): executing NAT**

*// net/netfilter/nf\_nat\_core.c*

*/\* Do packet manipulations according to nf\_nat\_setup\_info. \*/*

**unsigned** **int** **nf\_nat\_packet**(**struct** nf\_conn **\***ct, **enum** ip\_conntrack\_info ctinfo,

**unsigned** **int** hooknum, **struct** sk\_buff **\***skb)

{

**enum** nf\_nat\_manip\_type mtype **=** HOOK2MANIP(hooknum);

**enum** ip\_conntrack\_dir dir **=** CTINFO2DIR(ctinfo);

**unsigned** **int** verdict **=** NF\_ACCEPT;

statusbit **=** (mtype **==** NF\_NAT\_MANIP\_SRC**?** IPS\_SRC\_NAT **:** IPS\_DST\_NAT)

**if** (dir **==** IP\_CT\_DIR\_REPLY) *// Invert if this is reply dir*

statusbit **^=** IPS\_NAT\_MASK;

**if** (ct**->**status **&** statusbit) *// Non-atomic: these bits don't change. \*/*

verdict **=** nf\_nat\_manip\_pkt(skb, ct, mtype, dir);

**return** verdict;

}

**static** **unsigned** **int** **nf\_nat\_manip\_pkt**(**struct** sk\_buff **\***skb, **struct** nf\_conn **\***ct,

**enum** nf\_nat\_manip\_type mtype, **enum** ip\_conntrack\_dir dir)

{

**struct** nf\_conntrack\_tuple target;

*/\* We are aiming to look like inverse of other direction. \*/*

nf\_ct\_invert\_tuplepr(**&**target, **&**ct**->**tuplehash[**!**dir].tuple);

l3proto **=** \_\_nf\_nat\_l3proto\_find(target.src.l3num);

l4proto **=** \_\_nf\_nat\_l4proto\_find(target.src.l3num, target.dst.protonum);

**if** (**!**l3proto**->**manip\_pkt(skb, 0, l4proto, **&**target, mtype)) *// 协议相关处理*

**return** NF\_DROP;

**return** NF\_ACCEPT;

}

**5. Summary**

Connection tracking (conntrack) is a fairly fundamental and important network module, but it goes into normal developer or system maintainer’s eyes only when they are stucked in some specific network troubles.

For example, in highly concurrent connection conditions, L4LB node will receive large amounts of short-lived requestions/connections, which may breakout the conntrack table. Phenomenons in this case:

* Clients connect to L4LB failed, the failures may be random, but may also be bulky.
* Client retries may succeed, but may also failed again and again.
* Capturing traffic at L4LB nodes, could see that L4LB nodes received SYNC (take TCP as example) packets, but no ACK is replied, in other words, the packets get siliently dropped.

The reasons here maybe that conntrack table size is not big enough, or GC interval is too large, or even there are [bugs in conntrack GC](https://github.com/cilium/cilium/pull/12729).

**References**

1. [Netfilter connection tracking and NAT implementation](https://wiki.aalto.fi/download/attachments/69901948/netfilter-paper.pdf). Proc. Seminar on Network Protocols in Operating Systems, Dept. Commun. and Networking, Aalto Univ. 2013.
2. [Cilium: Kubernetes without kube-proxy](https://docs.cilium.io/en/v1.7/gettingstarted/kubeproxy-free/)
3. [L4LB for Kubernetes: Theory and Practice with Cilium+BGP+ECMP](http://arthurchiao.art/blog/k8s-l4lb/)
4. [Docker bridge network mode](https://docs.docker.com/network/bridge/)
5. [Wikipedia: Netfilter](https://en.wikipedia.org/wiki/Netfilter)