Per-IP rate limiting with iptables

WILL SEWELL AND JIM FISHER · 19 SEPTEMBER 2017

Every second, Pusher's main pub-sub system handles 9,000 new WebSocket connections. No sweat. But earlier this year, when the system started receiving spikes of 20,000 new connections every second *per server*, the sweat began to bead on our foreheads. What, or who, were these new connections? Were they malicious or mistaken? Most importantly, how could we keep the system going for everyone else while we dealt with this mysterious new force? This is the story of how we quelled the biggest threat to our service uptime for several years. The hero you will meet today is `iptables`, Linux's powerful (but dangerous) tool for interacting with the networking stack. Come with us and you too will learn how to wield `iptables`, and her secret weapons `conntrack` and `hashlimit`, to implement per-IP rate limiting!

Suddenly, one quiet afternoon in March …

At Pusher, one of our key health metrics is "new WebSocket connections per second". Each new connection is, say, a web page doing `pusher.connect()`. For our main cluster, this ticks along at 50 new connections per server every second. So we were concerned when, over the course of one day in March, random servers started experiencing spikes of 1,500 new connections per second.

What is it? A DDOS? It looked like the DDOS technique called a “SYN flood”, in which lots of fake TCP connections are opened by the attacker.

It turned out that this was not a malicious DDOS. Actually, faulty WebSocket clients were deployed somewhere in the wild which were stuck in connection retry loops, opening tens of thousands of connections per second to a single Pusher app! Not all of these connections show up in the above graphs because they were being dropped before our server process was even reporting them.

Noisy neighbors in the multi-tenant apartment

The flagship Pusher product is a multi-tenant system: a single cluster of Pusher servers runs apps for thousands of customers. Multi-tenancy has efficiency benefits, but comes with a significant drawback: there is now potential for a single customer to use more than their “fair share” of the system, and reduce quality of service for our other customers. This is the “noisy neighbor” problem.

Normally, noisy neighbors are silenced by Pusher’s limits system. For each tenant, Pusher limits both the number of connections and the number of messages. When a server process receives a WebSocket connection, the process first checks whether the account has reached its limits, and if so then closes the connection.

But in this case, our limits system was not enough. Our servers *were* rejecting the new connections from faulty clients, but just the act of handling and closing these new connections was incurring significant overhead! We found the servers couldn’t keep up with the connections when they were propagated to our userspace server process.

The only option left to us was to block the connections in the kernel. The solution we went for was to use the Linux firewalling system, Netfilter, which in turn can be manipulated by the `iptables` tool.

Fix #1: blocking an IP with `iptables`

Netfilter allows kernel modules to define callback functions that get executed when

packets are sent or received by the kernel networking stack. These functions will commonly perform operations like address or port translation, and importantly for us, they can drop packets entirely. Here's the task for this section: drop all packets from a specific blacklisted IP. We'll guide you through implementing this.

`iptables` is a userland program and command line tool for manipulating Netfilter callback functions. Conceptually `iptables` is based around the concepts of *rules* and *chains*. A rule is a small piece of logic for matching packets. A chain is a series of rules that each packet is checked against, in order. Packets eventually end up in one of a predefined set of *targets*, which determine what is done with the packet. The key targets are to `ACCEPT` the packet or `DROP` the packet[1]. Each rule defines where to "jump" if a packet matches; this can be to another chain or a target. If the rule does not match, then the packet is checked against the next rule in the chain. Each chain has a *policy*, which is the target packets jump to if they reach the end of the chain without matching any rules.

Let's see how you can add a rule that drops all packets from the IP address `123.123.123.123`. The rule can be appended with the following command:

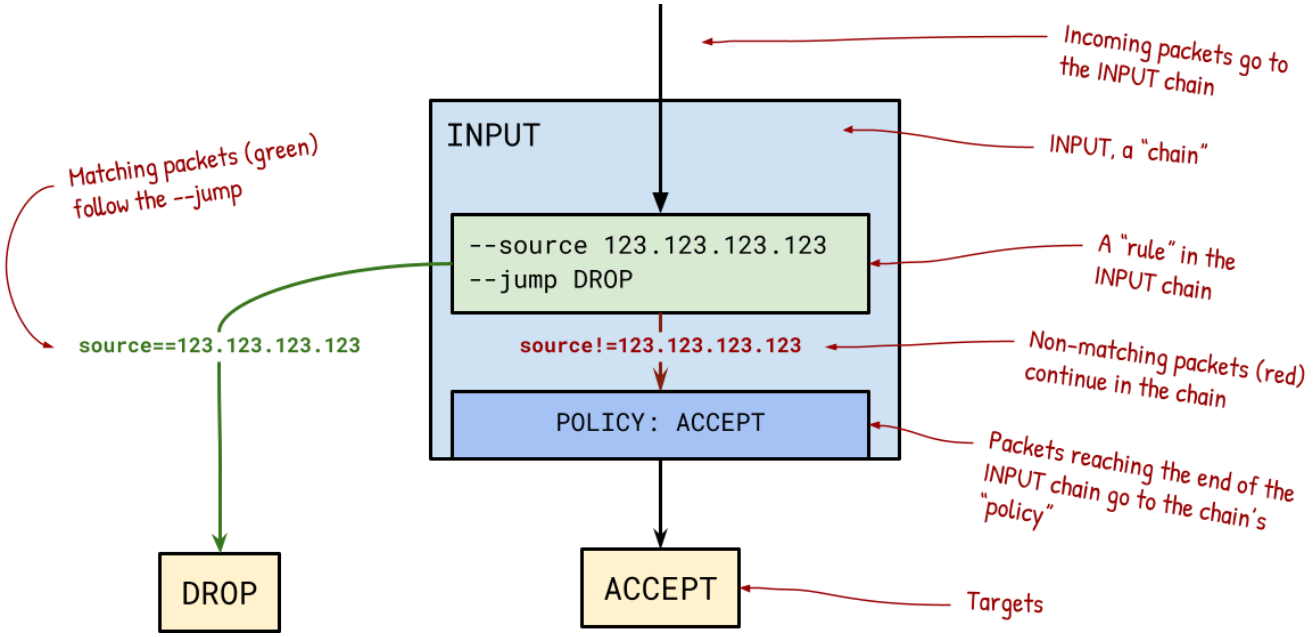
```
$ sudo iptables --append INPUT --source 123.123.123.123 --jump DROP
```

A word of warning: `iptables` can be dangerous. (The `sudo` should hint at this!) To follow along at home, you should use a disposable virtual machine. The classic `iptables` horror story is `ssh`ing to the machine, adding a broken rule, then suddenly finding your `ssh` connection is broken. Congratulations, your broken `iptables` rule blocked your SSH packets! If you're lucky, you can take a taxi to the data centre to fix the machine. Lesson: always test your `iptables` commands thoroughly on a local VM!

Warnings over with, let's analyse the above command. In English, it translates to: *append* a rule to the `INPUT` chain (the chain all packets destined for this server arrive on) and if a packet matches source IP `123.123.123.123`, then drop it.

Let's visualise what this chain looks like now. Follow the diagram below: if it receives a packet from IP `1.2.3.4`, what target does it reach? Answer: It reaches the ACCEPT target. It first hits the only rule in the INPUT chain, which it does not match. It then falls through to the default policy of the INPUT chain, which is the ACCEPT

target.



In the shell, you don't get a pretty diagram like that. Instead, you run `iptables --list`. Here's the textual equivalent of the above diagram:

```
$ sudo iptables --list
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  123.123.123.123      anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

With the above rule, we are able to block blacklisted IP addresses from ever communicating with our service. This is great in a bind, but it's pretty brutal! First, we only needed to block *new* connections (identified by the TCP SYN flag), and leave current connections intact. Second, we only needed to *rate limit* these connections, not block them entirely.

It turns out such rate limiting is not possible with out-of-the-box `iptables`. All `iptables` rules are stateless, but rate limiting requires state (for counters). For more power and flexibility, we need `iptables` modules. These modules are required to define more complex, stateful rules like a connection rate limiter. First up is a module called `conntrack`. `conntrack` to the rescue!

called `conntrack`. `conntrack` to the rescue!

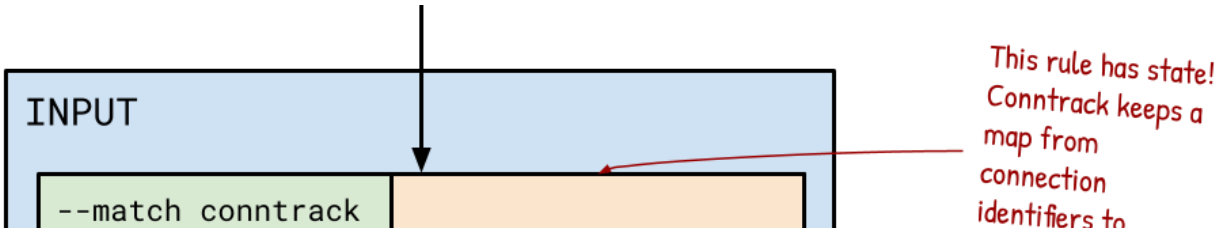
Modules warmup: the `conntrack` and `log` modules

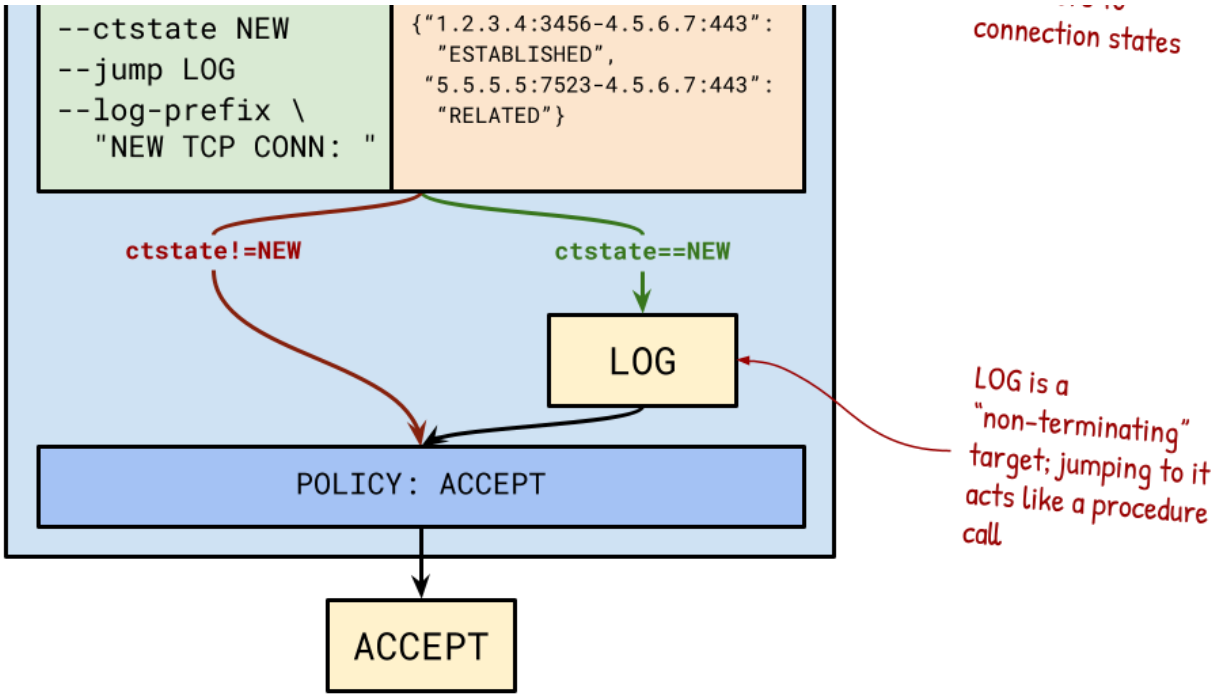
As an introduction to `iptables` modules, let's create a rule that will log all new TCP connections. To do this, we'll use two modules: `conntrack` to find new TCP connections, and `log` to log them.

Normal `iptables` rules work independently on each IP packet. But to rate limit *connections*, we need to track "connections" as well as packets. For this, there is the Netfilter connection tracking system. This maintains a set of all connections and updates the connection states as new packets arrive. A single connection is identified by the layer 3 and 4 connection information, e.g. the IP address and port respectively. To access this connection information, we use `conntrack`, which is an `iptables` module. With `conntrack`, you can create rules which access the connection of the current packet. For example, `--match conntrack --ctstate NEW` will only match packets where the connection is in the `NEW` state. Here, `-m` defines a module to use in the rule, and subsequent parameters apply to that module.

Logging is provided by a module called `log`. The `log` module defines a new target: `LOG`. If a rule jumps to this target, a string is logged to the syslog with the `kern` facility (in `/var/log/kern.log` on Debian/Ubuntu by default). `LOG` is a "non-terminating target", which means that the next rule is checked in the chain immediately before we jumped to `LOG`, whether or not the log rule matches. Let's add a new rule to log each dropped packet:

```
$ sudo iptables --flush  # start again
$ sudo iptables --append INPUT --protocol tcp --match conntrack --
ctstate NEW --jump LOG --log-prefix "NEW TCP CONN: "
```





Check your understanding: if the above table receives a packet from `10.0.0.51`, will it get logged? Answer: Yes! The packet hits the `conntrack` rule, which checks whether it's a new connection. There is no mention of `10.0.0.51` in `conntrack`'s connection states, so it must be a new connection. Therefore, the packet matches the rule, and jumps to LOG. Take a look in the system log:

```
$ tail -f /var/log/kern.log
Aug 30 15:47:32 ubuntu-xenial kernel: [10766.412639] NEW TCP CONN:
IN=enp0s8 OUT= MAC=08:00:27:dd:80:b3:08:00:27:e2:1b:be:08:00
SRC=10.0.0.51 DST=10.0.0.50 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=20232
DF PROTO=TCP SPT=33528 DPT=1234 WINDOW=29200 RES=0x00 SYN URGP=0
Aug 30 15:48:39 ubuntu-xenial kernel: [10833.521111] NEW TCP CONN:
IN=enp0s8 OUT= MAC=08:00:27:dd:80:b3:08:00:27:e2:1b:be:08:00
SRC=10.0.0.51 DST=10.0.0.50 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=42506
DF PROTO=TCP SPT=42156 DPT=1233 WINDOW=29200 RES=0x00 SYN URGP=0
```

Warning: this might look like it's working correctly, but it might not be! If you leave it for a while, you might eventually see lines like this:

```
nf_conntrack: table full, dropping packet.
```

We never told `conntrack` to drop packets; we only told it to log new connections! It turns out `conntrack` has a maximum number of tracked connections, and once

`conntrack` reaches this limit, all new connections are dropped! The details are pretty ugly,[2] but in short you must set the maximum high enough to avoid dropped connections, and you must set a "number of buckets" high enough to avoid poor performance:

```
$ conn_count=$(sysctl --values net.netfilter.nf_conntrack_count)
$ sysctl --write net.netfilter.nf_conntrack_max=${conn_count}  # Set
this much higher than your conn count!
$ sysctl --write
net.netfilter.nf_conntrack_buckets=$((${conn_count}/4))  # Technical
reasons, see footnote
```

After all this, you have a system which distinguishes `NEW` connections from `ESTABLISHED` ones (and ones in other states). With this, we could jump to `DROP` instead of `LOG` to block all new connections. This would be an improvement over our previous attempt, which brutally blocked all connections. But it's still not what we want. We don't want to block *all* new connections, only ones which exceed a certain rate. For this, we need another module: the `limit` module.

Fix #2: Rate limiting with the `limit` module

The `limit` module enables rate limiting against all packets which hit a rule. First we'll create a new chain, `RATE-LIMIT`. We'll send packets to the `RATE-LIMIT` chain if they are in the `NEW` connection state. Then, in the `RATE-LIMIT` chain, we will add the rate limiting rule.

```
$ sudo iptables --flush  # start again
$ sudo iptables --new-chain RATE-LIMIT
$ sudo iptables --append INPUT --match conntrack --ctstate NEW --jump
RATE-LIMIT
```

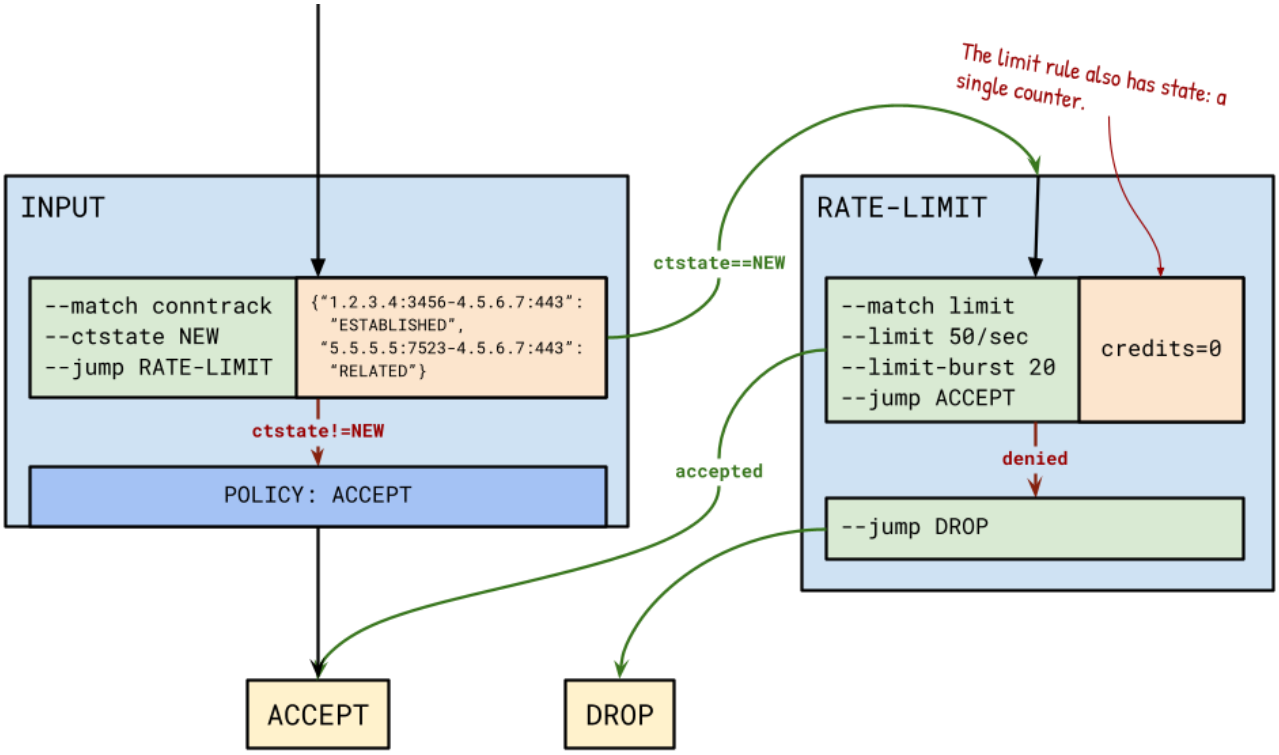
Then in the `RATE-LIMIT` chain, create a rule which matches no more than 50 packets per second. These are the connections we'll accept per second, so jump to `ACCEPT` (I'll explain `--limit-burst` later)

ACCEPT. (I'll explain `--limit-burst` later.)

```
$ sudo iptables --append RATE-LIMIT --match limit --limit 50/sec --
limit-burst 20 --jump ACCEPT
```

The `limit` rule rate-limits packets by *not* matching them, so they fall through to the next rule. These packets we'll drop:

```
$ sudo iptables --append RATE-LIMIT --jump DROP
```

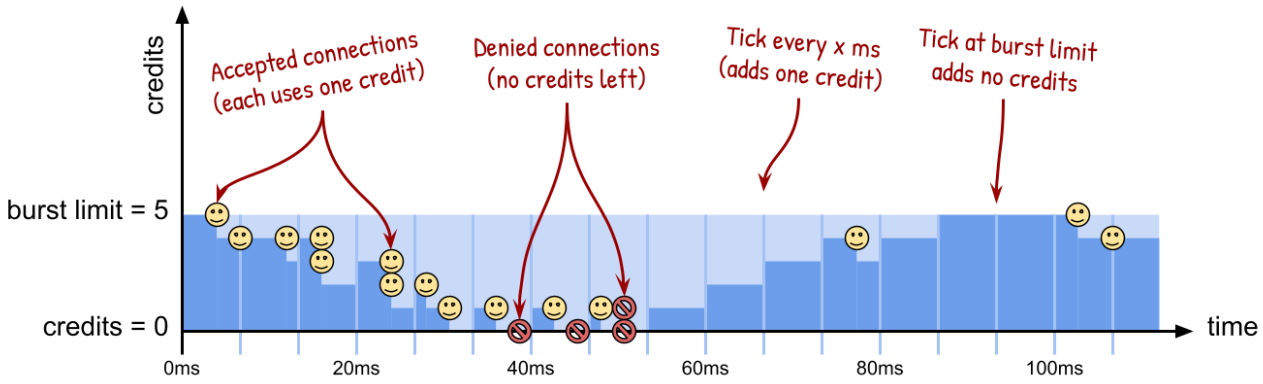


The rate limit (above, `50/sec`) is enforced as follows. Each `limit` rule has a bank account which stores "credits". Credits are tokens to be spent on matching packets. The limit rule only earns credits one way: through its salary, which is one credit per "tick". The above rule earns one credit every 20ms, thus restricting its spending to at most 50 matches per second. When a new packet comes in, if the rule has at least one credit, the credit is spent and the packet is matched. Otherwise, the packet falls through due to insufficient funds. Now, follow the diagram above: will a packet from `5.5.5.5` be accepted or dropped? Answer: The packet will reach the DROP target. The packet will first be identified by `conntrack` as a new connection, because its source is not in the connection states. It therefore jumps to the RATE-LIMIT chain. Here, the `limit` module checks its credits, but there are 0! Therefore, `limit` does

not match the packet, so it falls through to the next rule which jumps to the DROP target.

This “credit” scheme allows “bursty” traffic. If 100 different users just happen to connect at the same time, we want to allow them all. The credit scheme’s tolerance of random fluctuation is desirable, but it has an undesirable side-effect. Note that, during the night when users are asleep, the rule could earn a huge amount of credits, which can then be spent to allow users to overload the system in the morning when they all open connections at once. We want to allow “bursty” traffic, but only up to a limit.

This is exactly what `--limit-burst` fixes. The burst limit is a cap on the number of credits that the rule can have in its account. The above rule is only allowed 20 credits. If the rule already has 20 credits when a tick happens, it doesn’t earn any more credits. This “use it or lose it” logic prevents enormous build-ups of credit, and thus prevents overloading the system. (The burst limit also happens to be the rule’s initial number of credits, so it doesn’t have to wait to build up credits.) See an example:



Another quiz! If a `limit` rule is configured as `--limit 50/sec --limit-burst 20`, and then receives 1 packet every millisecond over a period of 1 second, how many packets will be matched? Answer: 70. The first 20 packets will be accepted in the first 20ms, depleting the credit to 0. The credit will recharge by 1 every 20ms (1/50 seconds), allowing a single packet through each time it is recharged. This gives a total of 50.

Without the `limit` module, we were blocking *all* new connections. This was useless, and the `limit` module is a big improvement: we can now prevent our server from getting destroyed by huge numbers of new connections. But there is a fundamental limitation with `limit` in our multi-tenant system: it applies this rate limit globally. If a

limitation with `limit` in our multi-tenant system. It applies this rate limit globally. If a single client exceeds the limit, connections from clients who are using their fair share will be dropped.

One solution would be to match on a blacklist of source IP addresses. But this would require us to manually add new IPs to the tables (or implement our own system for doing this). Ideally we want to rate limit every source IP address separately. This is exactly what the `hashlimit` module is for.

Fix #3: Rate limiting per IP address with `hashlimit`

`hashlimit` generalises the `limit` module. Whereas `limit` applies a single limit globally to all connections, `hashlimit` applies limits to *groups* of connections (a group can be a single source address). This works similarly to the `limit` module, but now each connection group gets its own credit account.

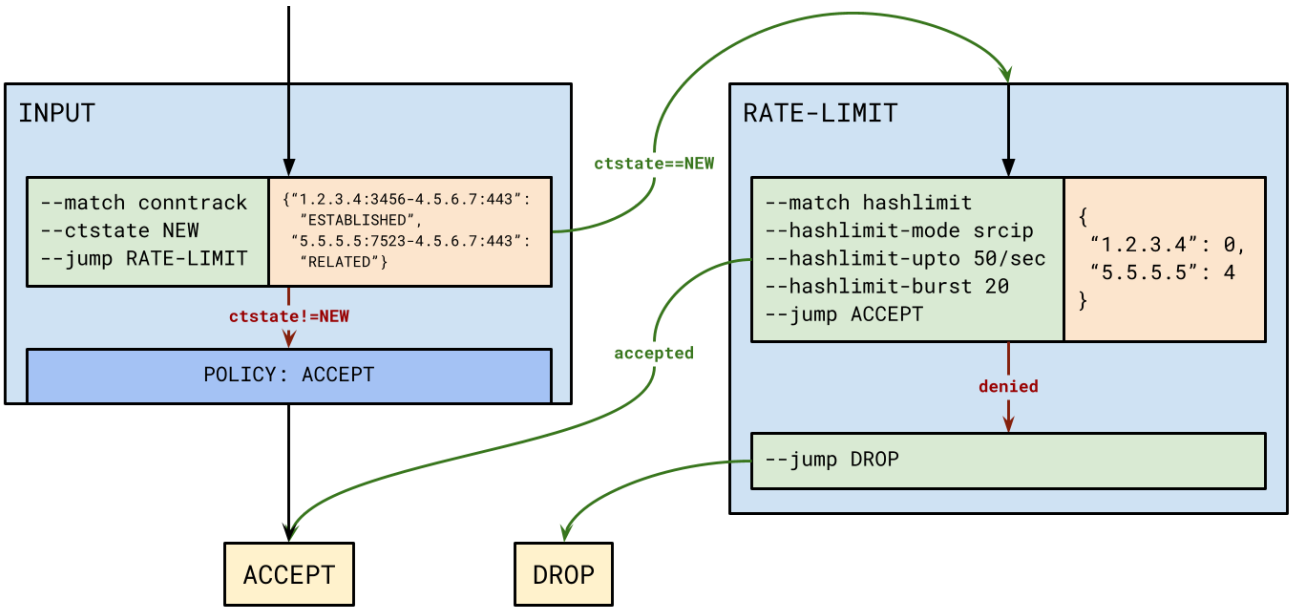
All `limit` rules can be defined with `hashlimit` by putting all connections in one big global group. This is what `hashlimit` does by default. So we can define the previous `limit` rule with `hashlimit` like this:

```
$ sudo iptables --flush  # start again
$ sudo iptables --new-chain RATE-LIMIT
$ sudo iptables --append RATE-LIMIT \
    --match hashlimit \
    --hashlimit-upto 50/sec \
    --hashlimit-burst 20 \
    --hashlimit-name conn_rate_limit \
    --jump ACCEPT
$ sudo iptables --append RATE-LIMIT --jump DROP
```

To instead limit per source IP, we need to tell `hashlimit` to group by source IP address. We do this with the `--hashlimit-mode` parameter, which defines how to group the packets. With `--hashlimit-mode srcip`, we create a group per source IP:

```
$ sudo iptables --append RATE-LIMIT \
```

```
        --match hashlimit \
        --hashlimit-mode srcip \
        --hashlimit-upto 50/sec \
        --hashlimit-burst 20 \
        --hashlimit-name conn_rate_limit \
        --jump ACCEPT
    $ sudo iptables --append RATE-LIMIT --jump DROP
```

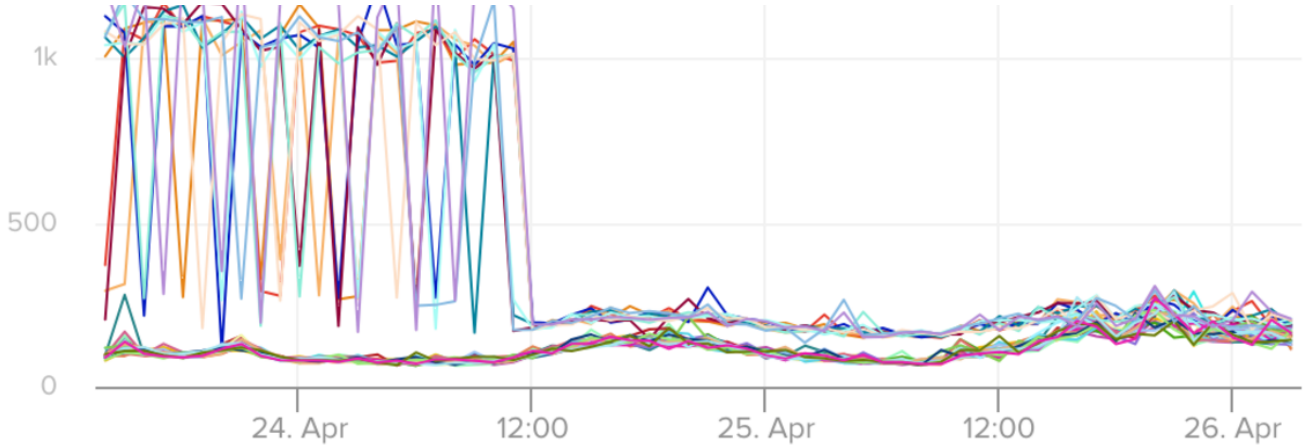


Follow the diagram. If a packet comes in on the connection from `1.2.3.4:3456`, what happens to the packet? Click to reveal the answer ...

Like `conntrack`, `hashlimit` tables have a maximum number of entries, and you should not let the table fill up. You can view the current hash table entries in `/proc/net/ipt_hashlimit/conn_rate_limit`. You should set `--hashlimit-htable-max` higher than the number of lines. You should also set `--hashlimit-htable-size` to max/4.[3]

Success!

We finally had the tools to rate limit new connections: `conntrack` and `hashlimit`. Here's the satisfying moment when the rules were deployed:



This was a great success: our servers breathed a massive sigh of relief, and the noisy neighbor was effectively silenced. What is more, there was no noticeable performance impact of these `iptables` rules, relative to everything else we are running. This is amazing considering these rules are executed against every single packet hitting the system!

Keeping an eye on your dropped connections

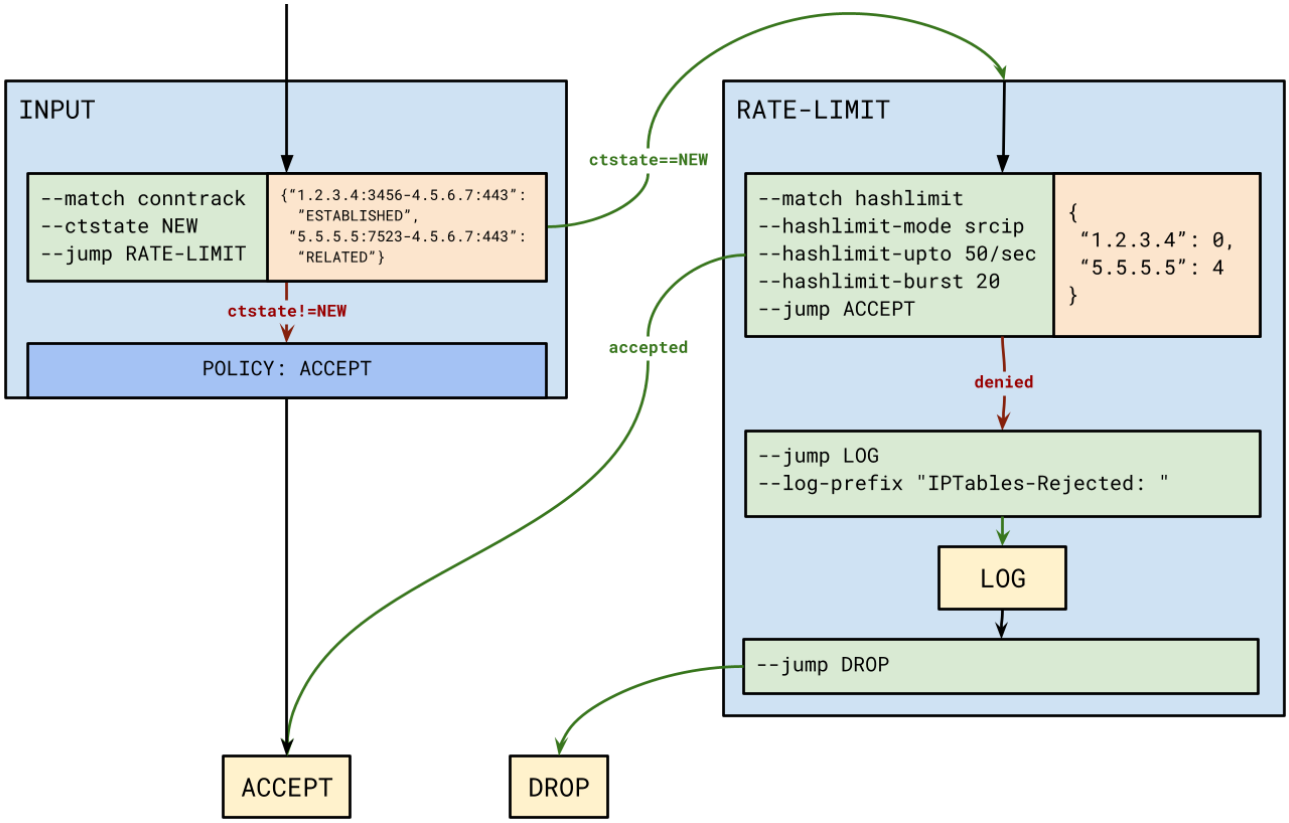
Dropping packets is risky because if an error is made in the rule, legitimate connections will be silently dropped. To keep an eye on your rate-limited connections, you have a couple of options. One is `iptables --list --verbose`, which shows the number of packets that have matched a rule; the count is under the `pkts` column. For more information, you can use the `log` module from earlier. Let's add a new rule to log each dropped packet:

```
$ sudo iptables --append RATE-LIMIT \
    --match hashlimit \
    --hashlimit-mode srcip \
    --hashlimit-upto 50/sec \
    --hashlimit-burst 20 \
    --hashlimit-name conn_rate_limit \
    --jump ACCEPT
$ sudo iptables --append RATE-LIMIT --jump LOG --log-prefix
"IPTables-Rejected: "
$ sudo iptables --append RATE-LIMIT --jump REJECT
```

You may find it excessive to log every single packet that is dropped; most likely it will be just as useful if we log a sample. How can we do this? Answer: the `limit` module again! Let's see how we can do this:

```
$ sudo iptables --append RATE-LIMIT --match limit --limit 1/sec --
jump LOG --log-prefix "IPTables-Rejected: "
```

This means only one dropped packet per second will be logged. I think this is a neat demonstration of how these simple and general modules can be composed in rules; we have used the `limit` module to achieve two things that are superficially very different: rate limit *and* logging!



Persisting `iptables`

`iptables` chains and rules are stored in-memory. If you reboot the machine they are lost. For that reason, `iptables` has a tool for saving and loading rule definitions. To save the current set of rules to a file, run

```
$ sudo iptables-save | tee rules.txt
```

```
*filter
:INPUT ACCEPT [66:3398]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [45:3516]
:RATE-LIMIT - [0:0]
-A RATE-LIMIT --match hashlimit --hashlimit-upto 50/sec --hashlimit-
burst 20 --hashlimit-mode srcip --hashlimit-name conn_rate_limit -j
ACCEPT
-A RATE-LIMIT --match limit --limit 1/sec -j LOG --log-prefix
"IPTables-Rejected: "
-A RATE-LIMIT -j REJECT --reject-with icmp-port-unreachable
COMMIT
```

To restore the rules run:[4]

```
$ sudo iptables-restore < rules.txt
```

We have this run on startup as an upstart task.

Conclusion

Hopefully we have demonstrated the power and flexibility of `iptables`: with just a few commands, we implemented a very efficient per-IP rate limiter, and the use-cases go way beyond just rate limiting. `iptables` has a reputation for being unfriendly. Yes, it's easy to kill your server with one wrong command, but it's also possible to save your server from attackers with just one good command! Yes, `iptables` is fiddly, low-level, and leaks many implementation details, but its efficiency goes well beyond what you can achieve in your userland process. And yes, `iptables` is poorly documented, but hopefully this blog post helps remedy that!

Footnotes

1. You can also `REJECT` packets by sending an ICMP error response, which could inform the client of what it is doing wrong. In these examples we will stick to `DROP` for simplicity. ↩

2. The `conntrack` module tracks its connection state `structs` in a global hash table. Most hash table implementations ensure good performance by dynamically resizing their number of buckets, but `conntrack` module does not implement this resizing! Instead, the number of hash table buckets is set statically with `nf_conntrack_buckets`. This leaves `conntrack` open to performance degradation if the number of entries in the table grows larger than the number of buckets. To protect against this performance degradation, `conntrack` has this artificial maximum number of entries, `nf_conntrack_max`. The ratio between the two settings, `nf_conntrack_buckets` and `nf_conntrack_max`, is important for performance. The `conntrack` documentation recommends `nf_conntrack_max = nf_conntrack_buckets*4`, but leaves this problem to the user.

   This imposes quite a burden on the user of `conntrack`! She must avoid dropped packets by setting `nf_conntrack_max` higher than her server's maximum number of hash table entries. This means she must know that maximum number, but this depends on many messy things: concurrent connections, various configured TTLs, and hash table access patterns. More, she must avoid performance degradation by setting `nf_conntrack_buckets` in line with her chosen `nf_conntrack_max`. She must also ensure this config is persisted in her `sysctl` configuration file. ↩

3. Like `conntrack`, `hashlimit` uses hash tables which do not dynamically resize, and protects against bad performance with a configurable maximum number of entries. Also like `conntrack`, `hashlimit` lets you shoot yourself in the foot with a wrong number of buckets. ↩

4. `iptables-restore` is not entirely idempotent: it will reset the packet counts. ↩