A Brief Look at Tuning TCP Retransmission Behaviour

Grabbing Response Time by the Tail

Kevin Hogan
Rick Jones

Disclaimer: In no way, shape, or form should the results presented in this document be construed as defining an SLA, SLI, SLO, or any other TLA. The authors' sole intent is to offer helpful examples to facilitate a deeper understanding of the subject matter.

Introduction

While we all strive wherever possible to avoid it, packet loss is an unpleasant fact of life. It cannot be eliminated completely - otherwise there would be no need for TCP's retransmission functionality. :) The reasons for packet loss run the gamut of VMs/systems being unable to keep-up with incoming traffic, to temporary overload of a piece of network kit, to the occasional need to restart a piece of infrastructure for an update, to … cows! And any number of other reasons in between.

In this writeup we will take a brief look at the effects of short interruptions in the flow of packets and what effect "two" Linux TCP settings might have on what a TCP-using application would see. Those two settings are:

- the net.ipv4.tcp_thin_linear_timeouts sysctl
- the net.ipv4.tcp_rto_min_us sysctl / rto_min metric of a Linux route - aka "rto_min"

Here we focus on "request/response" applications where the requests and/or responses are small, rather than a bulk-transfer application. Such request/response applications will be particularly sensitive to packet loss because the only way TCP has to recover from such loss is via retransmission timeouts. And those can end-up being very much longer than the round-trip-time across the network.

Our methodology will be to use netperf TCP_RR (request/response) benchmarking and its report for the maximum response time it records between two VMs while we disrupt communication for various lengths of time. We use two VMs in the same Availability Zone, where the underlying network round-trip-time is already low. This tends to be where the effects of packet loss can be worst for a request/response application's response time. As clients and servers get further apart, and the network round-trip-time gets larger, the effect of a retransmission timeout is proportionately smaller.

Finally, we note that none of these settings apply to retransmissions of lost SYN[1]s or SYN/ACKs from a TCP connection. Retransmissions for these packets are treated differently because they are sent before the path RTT (round-trip-time) can be estimated - but the effect of a lost SYN can be even more damaging to a request latency that includes connection set-up time. Luckily, modern Linux kernels provide a way to tune the retransmit behavior of these initial packets as well, through eBPF code. In an appendix below, we discuss this issue in-depth and provide a working example of how this behavior may be tuned on a modern Linux distribution.

[1] A SYN(chronize) segment is a special packet TCP sends when establishing a TCP connection. It is part of the "three-way-handshake" of TCP connection establishment.

Summary

There ends-up being something of a game of “leapfrog” here between the options as to which is better when. It all comes down to just how long the interruption is expected to last. And which kernel you happen to be running. But in broad, handwaving terms, if the disruption period is for a second or less, using an rto_min of 50 milliseconds, and setting net.ipv4.tcp_thin_linear_timeout to 1 (one) is a good thing, shaving many score milliseconds from the length of time a given TCP connection will remain idle thanks to and after the disruption period.

If you happen to be running a 6.11 or later kernel, or a kernel with backported support for the net.ipv4.tcp_rto_min_us sysctl on both ends you can go as low as 5 milliseconds for an rto_min value because there is logic included which bases the Standalone ACK timer on the minimum retransmission timeout. (By default the Standalone ACK timer under Linux is 40 milliseconds.)

Tuning the net.ipv4.tcp_rto_min_us will also assist with something present in Google Cloud known as Protective ReRoute.

While the focus of this paper is disruptions of various durations, these suggested values are also of use when dealing with just the odd/random packet loss out there on the network.

Theory

A full description of TCP retransmission heuristics is beyond the scope of this writeup. The aspects of it which pertain to what we look at here are these:

- Packet loss for request/response workloads where the request and/or response is just one or a small number of TCP data segments in flight at one time will be recovered via retransmission timeout.
- There is a minimum Retransmission Timeout value ("rto_min") which depending on one's kernel can be configured on a per-route basis or also via the net.ipv4.tcp_rto_min_us sysctl, and which defaults to 200 milliseconds.[2]
- There is a sysctl setting (net.ipv4.tcp_thin_linear_timeouts) which can be configured to cause TCP to double the timeout for each successive retransmission of a given segment (value 0/zero - default) or retransmit that segment with the same timeout a few times first (value 1/one) when TCP determines a flow to be "thin" - basically not sending a lot of data at any given moment.

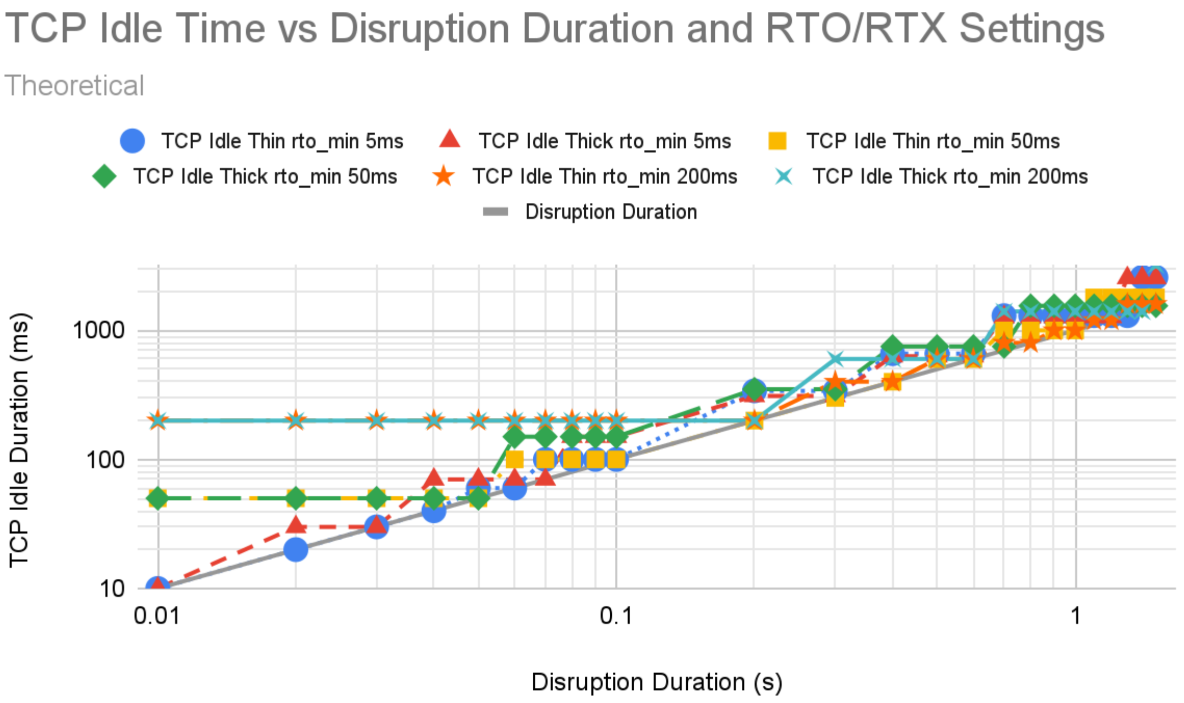
We will consider three values for rto_min. The 200 millisecond default, 50 milliseconds, and 5 milliseconds.

We will also consider both values for net.ipv4.tcp.thin_linear_timeouts - 0 (disabled) and 1 (enabled).

And with that, we can compute a theoretical length of time a given TCP connection would remain idle for a given duration of time when all traffic is disrupted. We assume that the "natural" round-trip time is much less than rto_min[3] and so a longer retransmission timeout is not calculated by TCP. When going below 50 milliseconds, one needs to be concerned with the Standalone ACK timer. Linux kernels with support for the net.ipv4.tcp_rto_min_us sysctl will also tune the Standalone ACK timer when that sysctl is set, making the Standalone ACK timer no longer a worry. Let's look at theory in an overview first:

[2] Kernels with support for net.ipv4.tcp_rto_min_us have that functionality used via the per-route "rto_min" metric as well. Take your pick.

[3] As we would expect it to be between two VMs in a given Availability Zone.

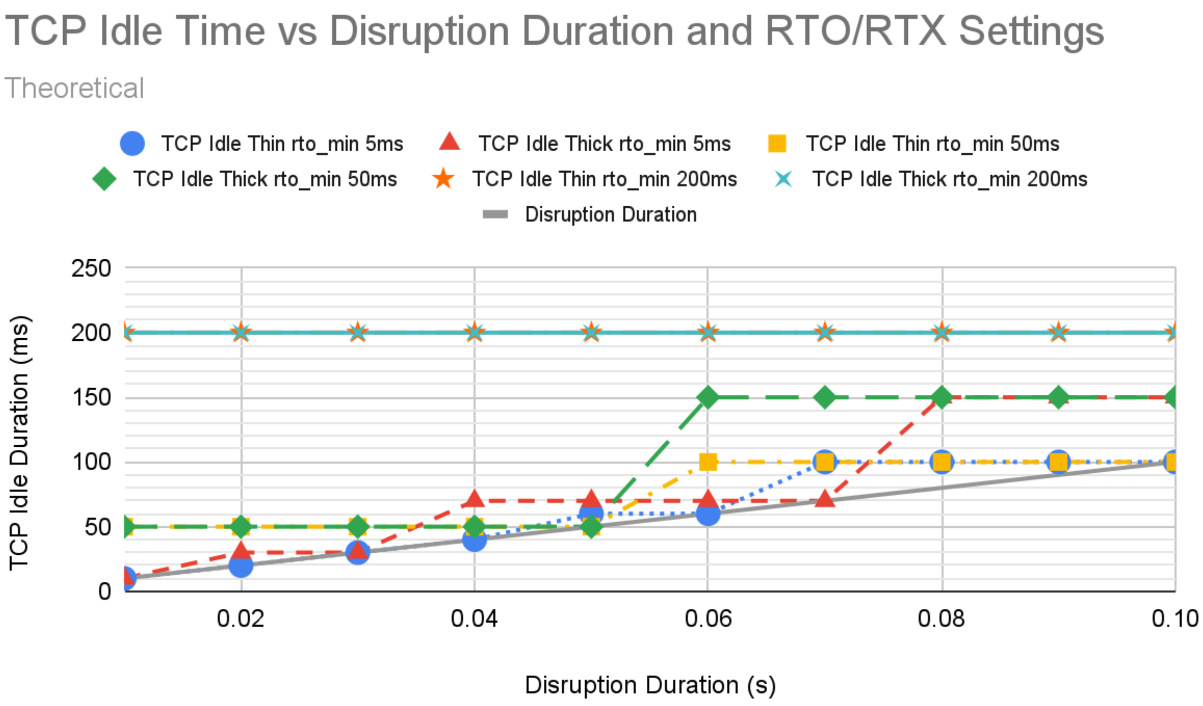


The logarithmic axes may be a bit unpleasant but are necessary at this level since we are spanning a few orders of magnitude along both axes. Basically what this shows is that for Disruption Durations less than the value of rto_min, the TCP connection will go idle for the length of rto_min.

What happens for Disruption Durations longer than rto_min depends on the value of net.ipv4.tcp_thin_retransmissions. When it is 0 (zero - called "Thick" in the chart) then we see the doubling each time until the TCP Idle Duration is longer than the Disruption Duration. When it is 1 (one - called "Thin" in the chart)) we first see repeats of the same rto_min timeout, and so generally sticking closer to the Disruption Duration. Then after a while, the doubling will start[4].

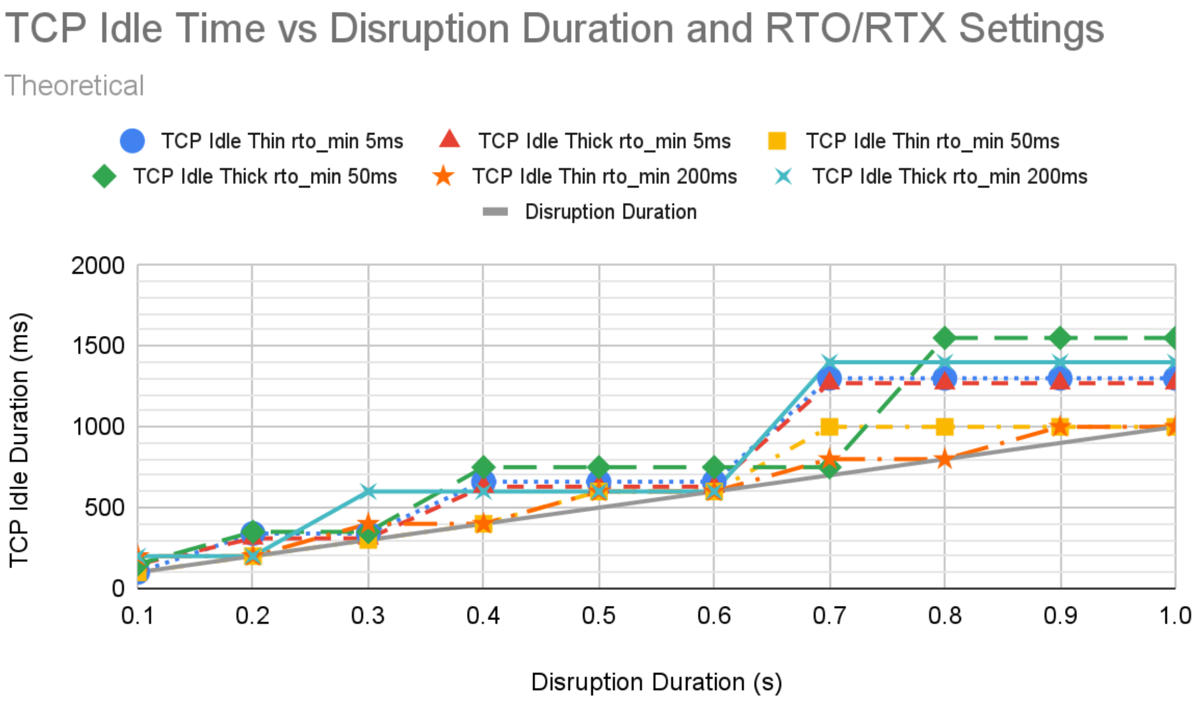
For those not terribly fond of logarithmic axes, we can look at the three different "regions" along the x-axis, using linear axes. First up, from 0.010 seconds to 0.1 seconds (ie from 10 to 100 milliseconds), in increments of 10 milliseconds:

[4] Remaining "linear" the entire time risks the dreaded "congestive collapse" which is why net.ipv4.tcp_thin_linear_timeouts=1 must start backing-off exponentially (doubling each time) after a few linear timeouts.



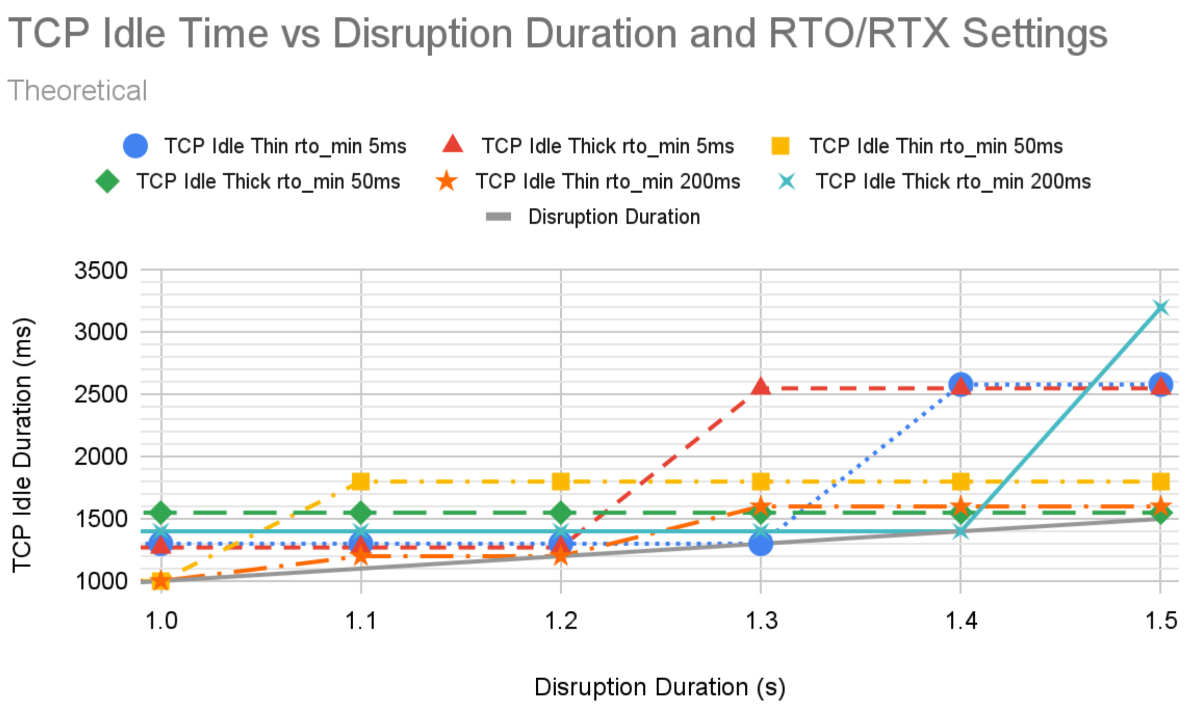
Broadly speaking, the lower the value of rto_min and the "thinner" the retransmission style, the closer to Disruption Duration TCP Idle Duration will be. We can see that the "thick rto_min" of 50 milliseconds has the connection idle for 50 milliseconds, then jumping to 150 milliseconds as the Disruption Duration grows beyond 50 milliseconds. The second Retransmission Timeout (RTO) becomes 100 milliseconds so the cumulative TCP Idle Duration becomes 150 milliseconds. With the "thin rto_min" of 50 milliseconds, that doubling does not happen and the TCP Idle Duration is just 100 milliseconds.   With an rto_min of 200 milliseconds, the TCP Idle Duration is the same through thick and thin… since the Disruption Duration is smaller than rto_min.

Next let's look at the interval from 0.1 to 1.0 seconds, in 100 millisecond intervals:



Without getting into too much detail, we can see how “thin” continues until it hits its limit before starting doubling. And we can see that sometimes, the TCP Idle Duration can be longer for a smaller value of rto_min when the timeouts are “thick.” The smaller rto_min has been able to double more often and can “leapfrog” the longer rto_min. But the longer rto_min will then itself leapfrog.

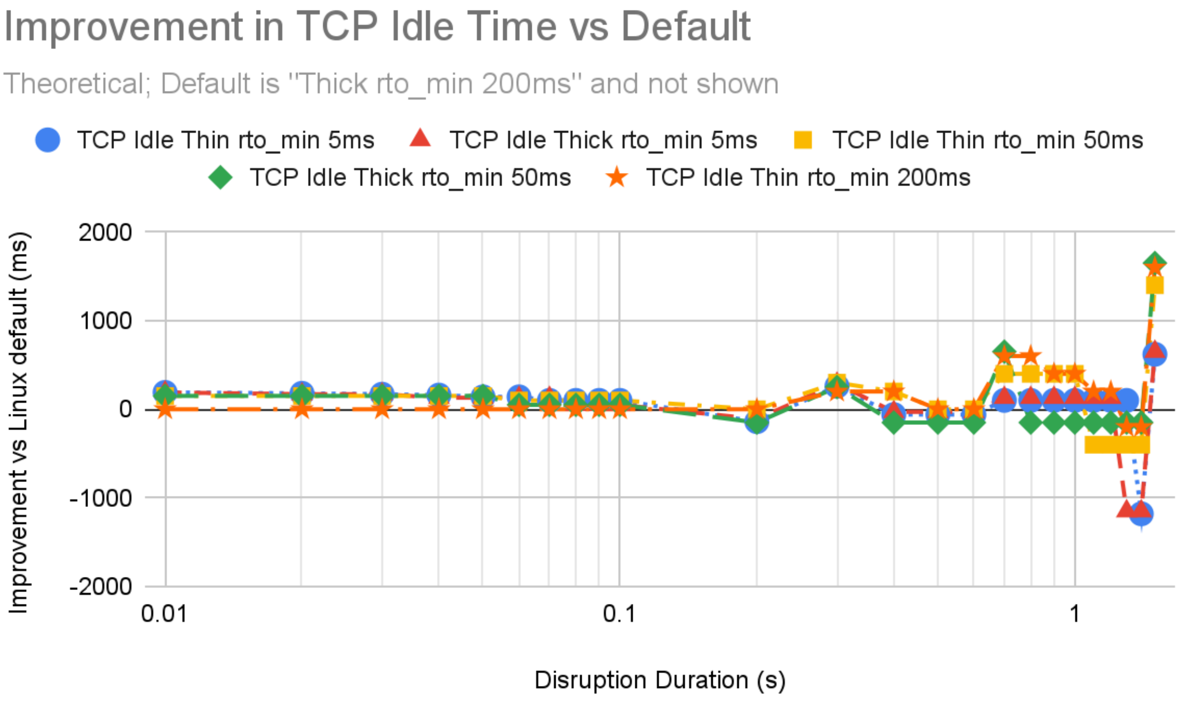
Let’s look at the last Disruption Duration region computed - from 1.0 to 1.5 seconds, in 100 millisecond intervals:



Variations on previous themes. A few hundred milliseconds here, a few hundred milliseconds there, pretty soon we're talking about real time ...

Improvement

Here we seek to distill all the above into a chart which suggests how much better (or worse) the TCP Idle Duration would be for a given Disruption Duration for each of the options. The default is "TCP Idle Thick (200ms)" which we will not include in the chart since it would always be an improvement of 0 (zero). In what follows, a positive value means improvement - a reduction in TCP Idle time relative to the default. Negative values mean degradation - an increase in TCP idle time relative to the default.

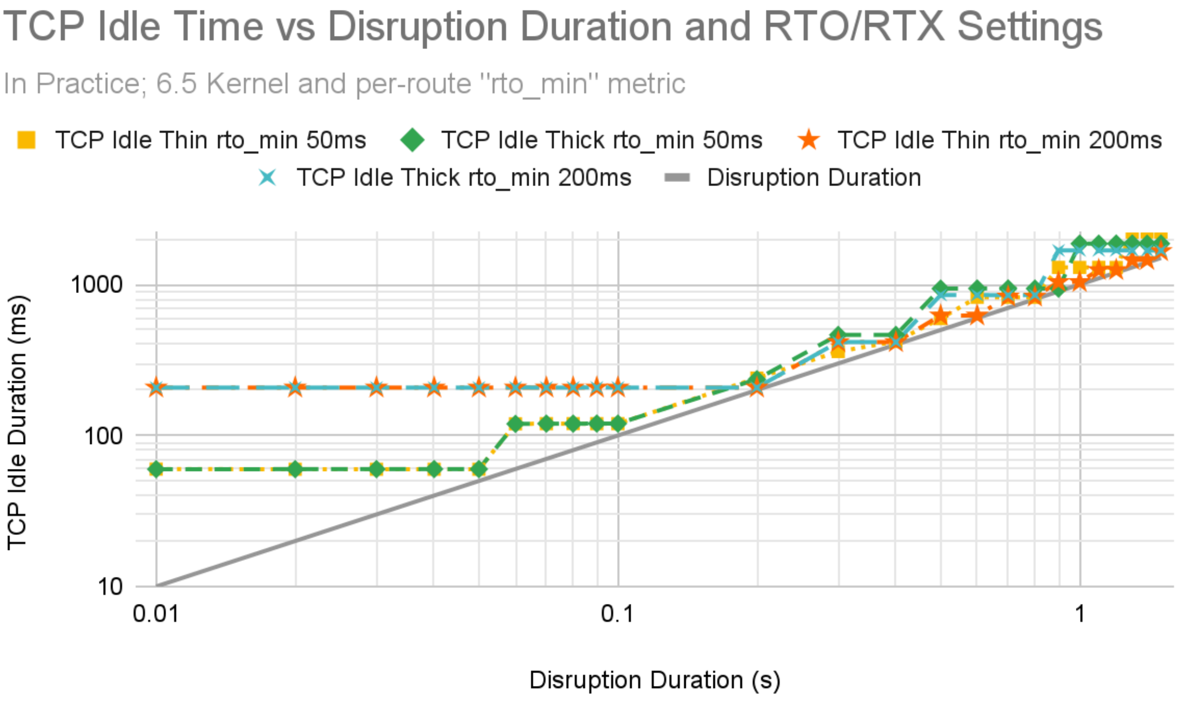


If there were a greater number of datapoints, better covering the places where stair-stepping happens, it would look more like a stair-step chart than what you see. Still, it shows that a given setting isn't always an improvement. It depends.

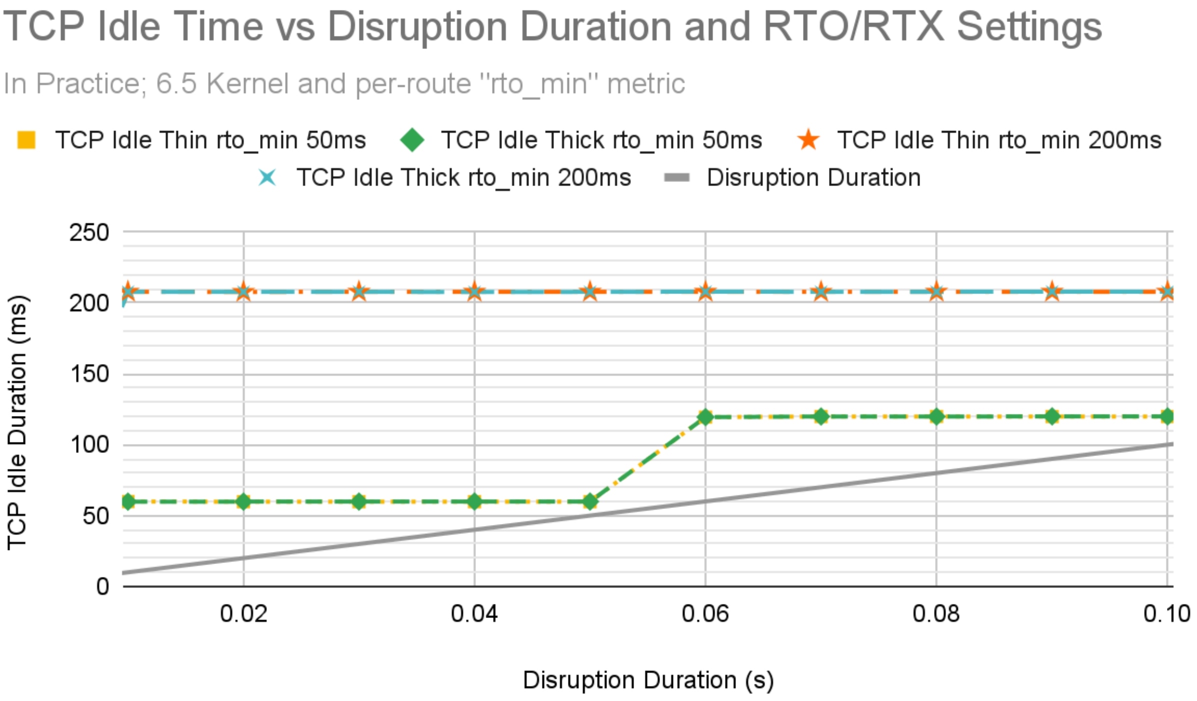
Rather than go region by region with Disruption Duration, let's just cut to the chase and see what happens in practice.

Practice

So, what does this all look like in practice? After all, we don't have spherical packets in a vacuum ... The VMs used for these charts were running a 6.5-based Linux kernel but the behaviour is likely similar with earlier kernels. The "route-based" rto_min methodology was used since the 6.5 kernel does not support the net.ipv4.tcp_rto_min_us sysctl. And we eschew the 5 ms rto_min value since we are on a kernel where the Standalone ACK timer does not adjust based on rto_min. Let's first look at the overall:

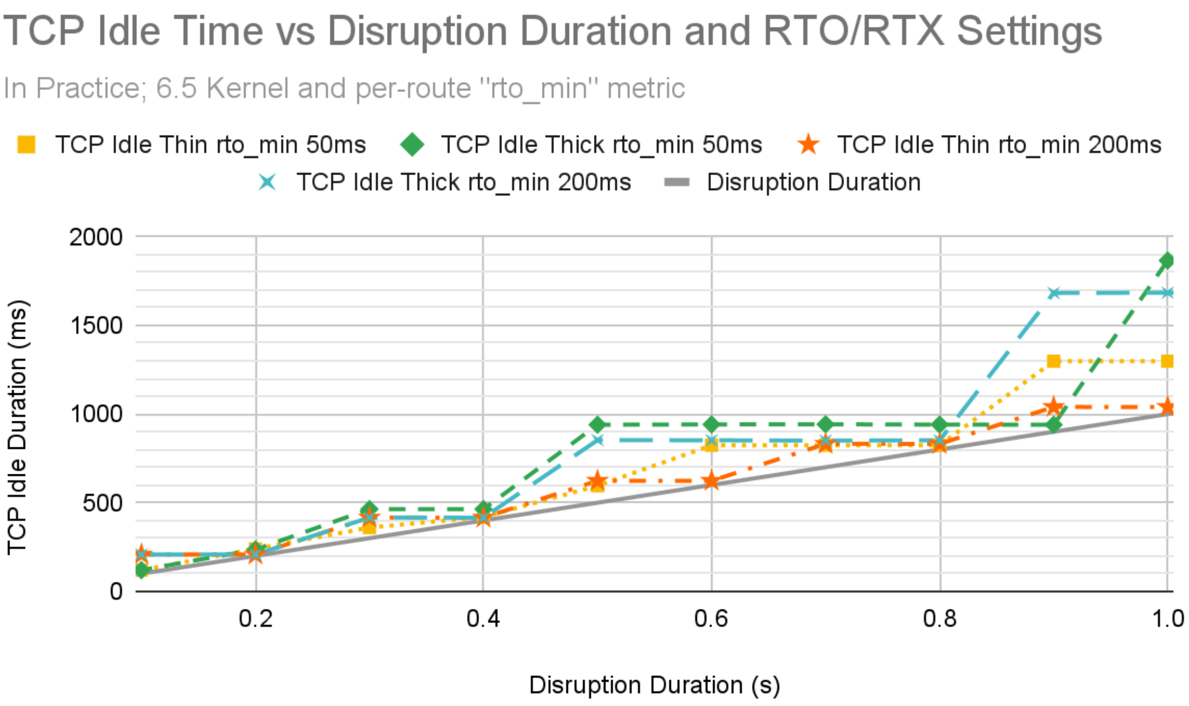


Rather like Theory, when the Disruption Duration is less than rto_min, the TCP connection remains idle for what appears to be rto_min. While it isn't perhaps immediately obvious, the y-axis on this chart does not go as high as it did on our theoretical chart. And we will see why in a moment.   A clue to that is visible in the next chart as we look at the first "region" of Disruption Duration.

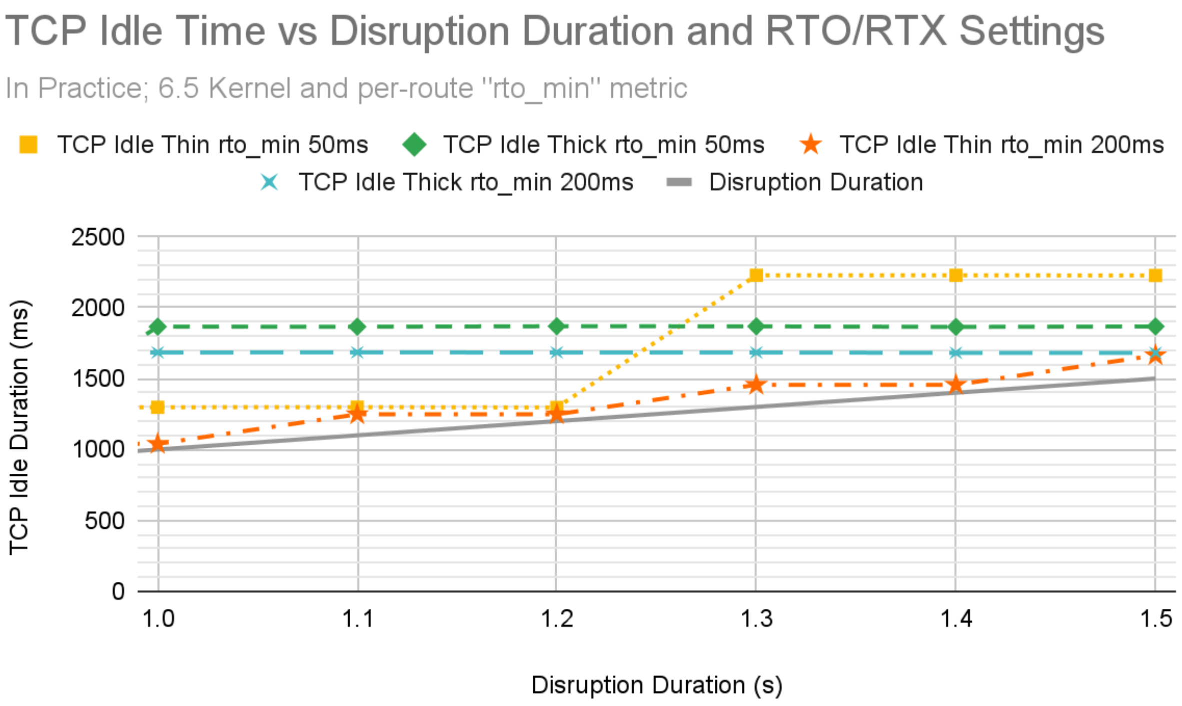


The sharp-eyed will notice that for the 50ms value of rto_min, our idle duration isn't actually 50 milliseconds, but closer to 60. While we configured the rto_min in units of milliseconds, those got converted to something else in the Linux kernel 6.5 networking stack, and the granularity of that conversion makes the timeout greater than 50 milliseconds. A similar effect can be seen with the 200ms value of rto_min.

And where we might have expected the TCP Idle Duration to be 100 milliseconds, it is actually more like but not quite 120 milliseconds. Since we are starting from a timeout which is actually a little larger than what we thought, we get a bit further than we thought. Also, for "thick" retransmissions with 50 ms rto_min we would have expected things to jump to more like 150/180 with a Disruption Duration of 0.6 seconds. However, the first retransmission under Linux here is actually a tail loss probe and so Linux does not start the doubling until one retransmission later. So, on to the next region:



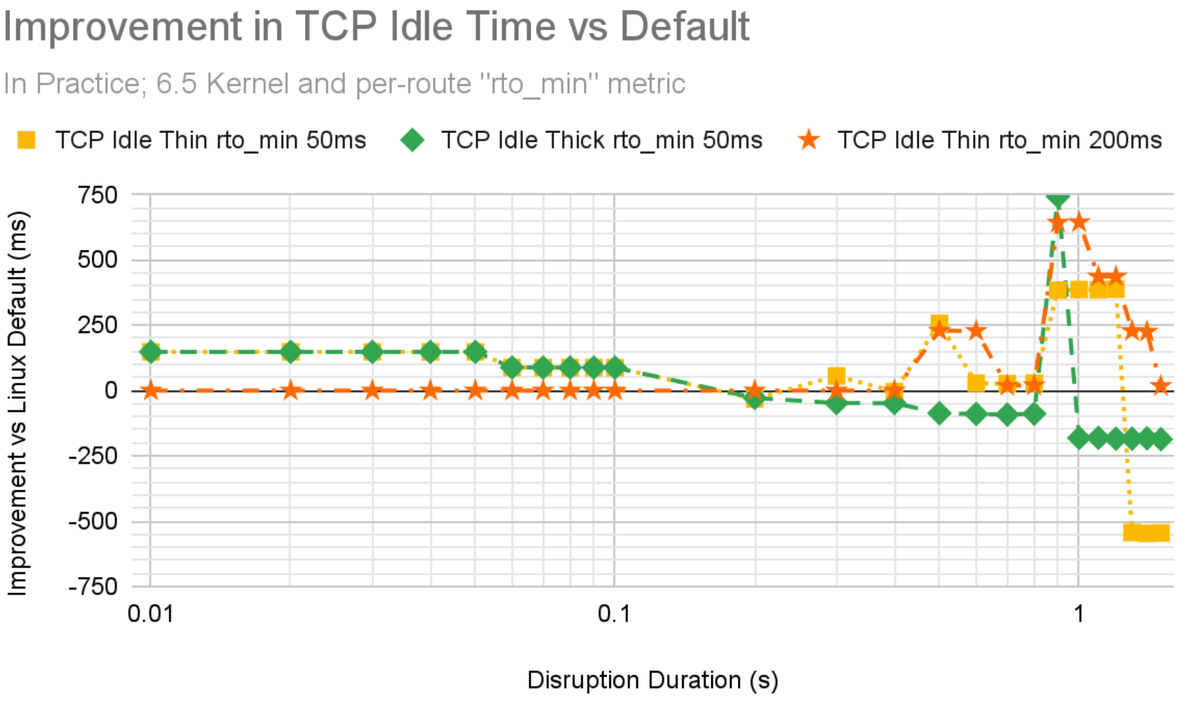
In the Theoretical chart for this region of Disruption Duration, TCP Idle Thick (200ms) was up to only 1400 milliseconds. But since the underlying starting point was more like 209 milliseconds, our doubling has taken us to not quite 1700 milliseconds. Similar things have happened with rto_min of 50 milliseconds. Which is why, when we look at the last region of Disruption Duration:



We no longer double again as we get to a backout duration of 1.5 seconds. And so why this chart's y-axis tops-out lower than the same chart from Theory. It isn't always in our favor though. In the theoretical chart for rto_min of 50 milliseconds and "thin" retransmissions we went only as high as 1800 milliseconds but here we end up between 2000 and 2300 milliseconds.

Improvement

In what follows, a positive value means improvement - a reduction in TCP Idle time relative to the default. Negative values mean degradation - an increase in TCP idle time relative to the default.



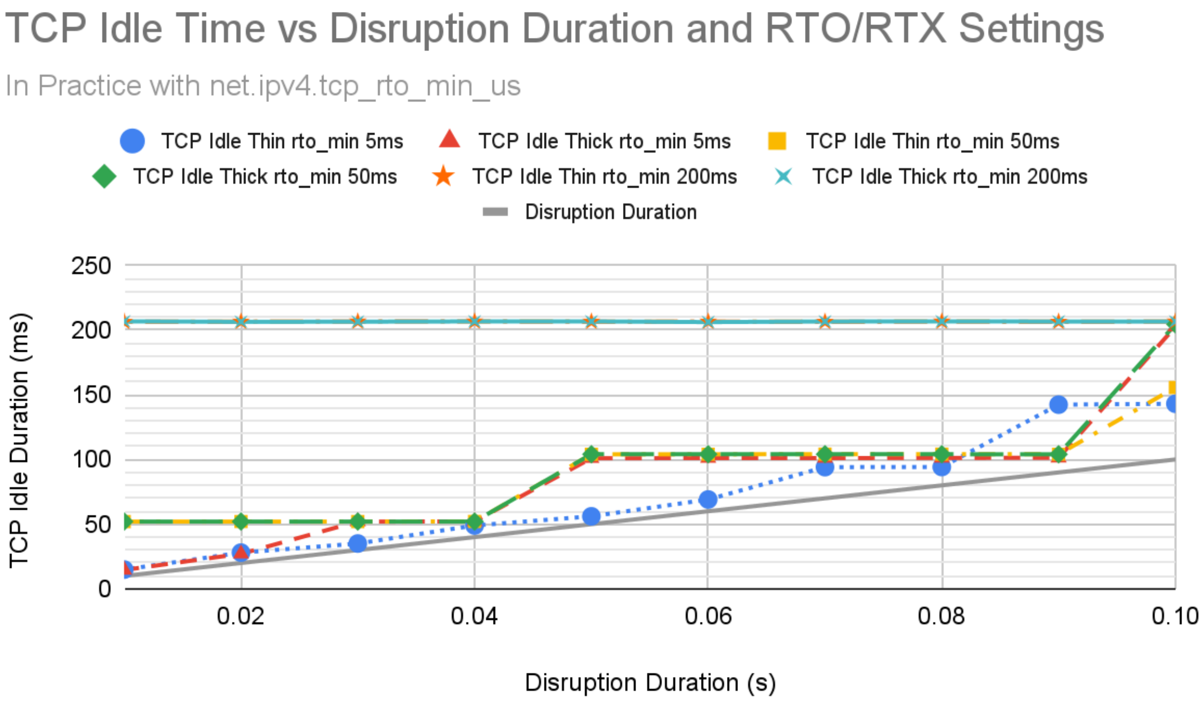
We don't bother showing the improvement with "thick" retransmissions and an "rto_min" of 200 milliseconds as it *is* the default.

Near Present

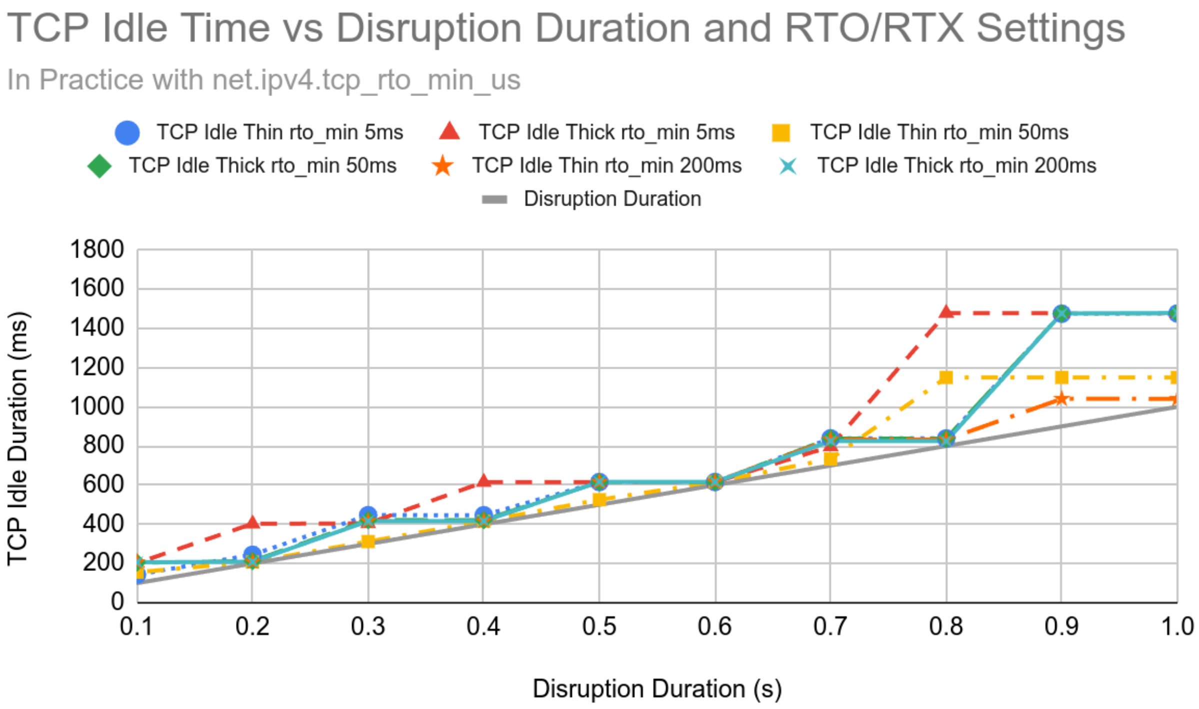
Up to this point we have been working with a 6.5-based kernel. The folks working on Linux networking are never standing still and upon passing an earlier version of this document to a few of them for their input, they mentioned that in a later kernel it is possible to go lower than we have to this point because a new sysctl has been added - net.ipv4.tcp_rto_min_us - and the value specified for that will influence the Standalone ACK timer.

This is available in the 6.11 version of the Linux kernel, which was being declared stable while some of these measurements were being made. Hence "near present." Support for the sysctl is also backported to "recent"[5] Ubuntu 6.8 and 5.15 kernels. For other distros you should consult the distro provider. So, another round of benchmarking was made, where a 5 ms rto_min was added, these being set via the newly-added net.ipv5.tcp_rto_min_us sysctl, which also tunes the value of the Standalone ACK timer. Rather than continue with the eye-strain that is the overall chart, let's go straight to the Disruption Duration region charts:

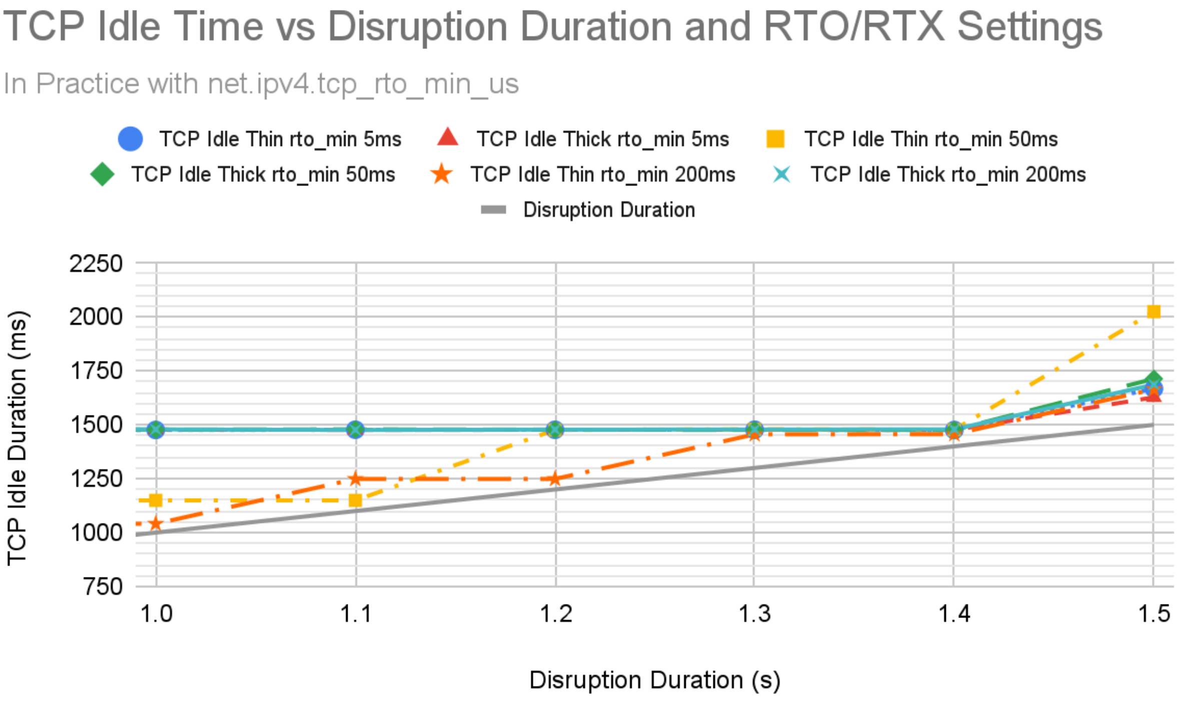
[5] Circa October 2024.



You can see that with an “rto_min” of 5 ms we eliminate still more TCP idleness for a given Disruption Duration compared to 200 ms rto_min (default) through most of this region. The one exception is circa a Disruption Duration of 0.07 seconds (70 milliseconds) where “thin” rto_min 5 ms improves less than rto_min 50 ms. By this point we’ve started doubling the retransmission timeout each time with rto_min 5 ms even with “thin” retransmissions. The sharp-eyed will also notice that now a “50 ms” rto_min is closer to 50 ms than 60 ms as it was in the 6.5 kernel, suggesting some improvements in the timer code. “200 ms” rto_min still seems to be in the 207 millisecond range. So, on to the next region:



Looks largely like variations on the theme, and the occasional bit of “leapfrog.” And then the final “region”

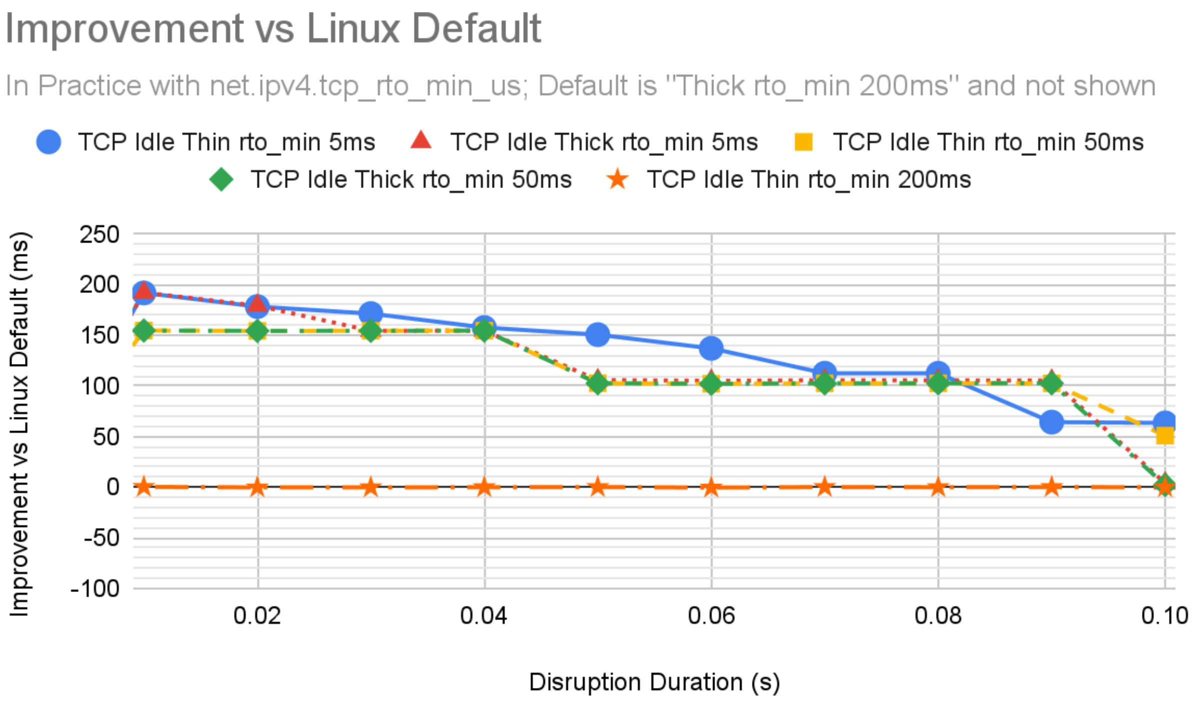


Lather. Rinse. Repeat.

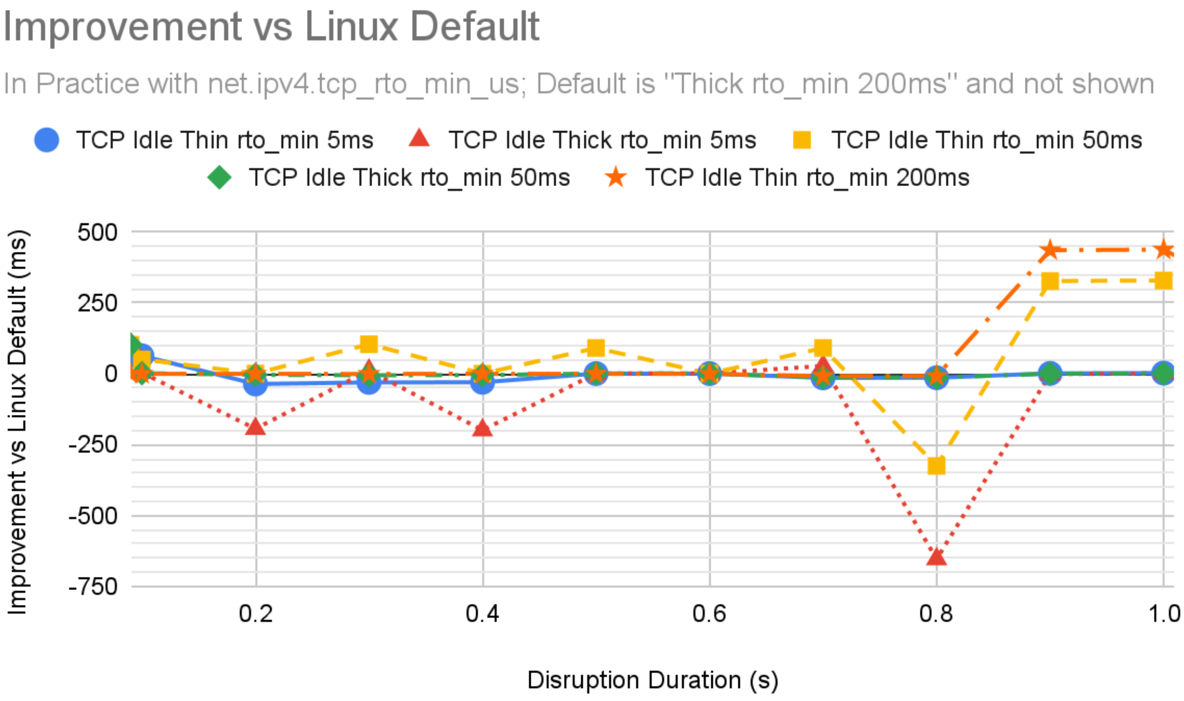
Improvement

In what follows, a positive value means improvement - a reduction in TCP Idle time relative to the default. Negative values mean degradation - an increase in TCP idle time relative to the default.

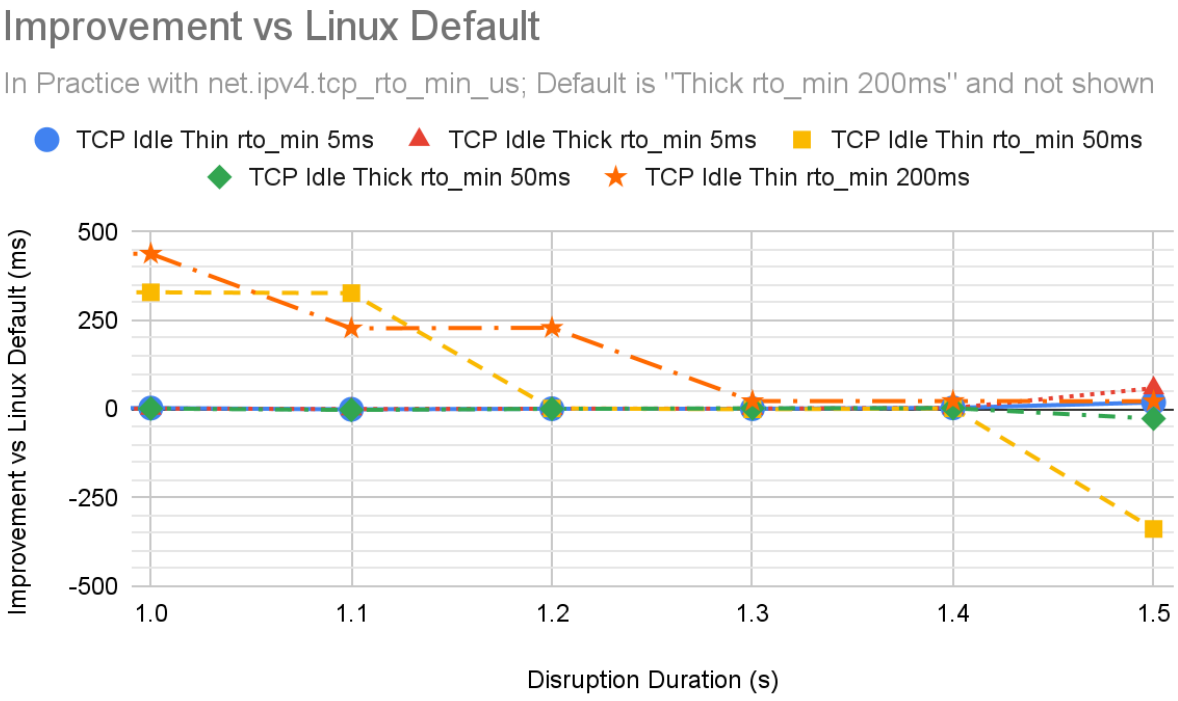
Rather than looking at the overall, we will again go from "region to region" of simulated Disruption Duration:



The “5ms thin” line rather nicely shows how net.ipv4.tcp_thin_linear_timouts works with the improvement changing by ~5ms or so at a time until the doubling-range is reached. Now on to the middle “region” of Disruption Duration:



We start to see some "convergence" of the different values. A reminder about the paucity of datapoints hiding the stair-stepping of reality. We can see that for 5 ms "rto_min" there are a few regions where we will have TCP idle a little longer than it would have with the default ("thick" retransmissions and an "rto_min" of 200 milliseconds).



By this final region of Disruption Duration, we see that things have largely "converged. There will still be some degree of leapfrog going-on.

Additional Considerations

In Improving Network Availability with Protective ReRoute, "PRR" (Protective ReRoute) is presented. In addition to the "Hypervisor Mode" of PRR there is a "Guest Mode" as well, wherein if one's VMs either use IPv6, or use gvNIC they will gain the benefits of PRR. And tuning the value of tcp_rto_min_us allows that mechanism to kick-in sooner. The pre-requisites are:

- The VM must run Linux with kernel 4.20 or later.
- The traffic must be TCP.
- The traffic must be IPv6 (irrespective of the virtual NIC driver) or gVNIC driver must be used (irrespective of the IP version).
- The net.ipv6.auto_flowlabels sysctl should be set to a value of 1 (one) though this should be the default on a 4.20 or later kernel.

Discussion

We’ve just looked at a lot of charts, and you may be wondering “How long will a disruption last?” It is an excellent question, but one right up there with “How long is a piece of string?” Or in this case perhaps, “How long will a cow stand on a network cable in a field?” Unfortunately, the answer is always the same - “It depends.” Use your experience as you gain it. Certainly Google seeks to minimize disruption duration wherever it can. The range selected for this paper was really an arbitrary one picked by the author(s).

Keep in mind these settings are useful for the odd, individual packet loss too, so let’s look at that for a moment. If you are running a request/response application, where you have a desired maximum response time, you can use that and your typical/expected service time to select a value for rto_min to ensure that in the event of a packet loss, TCP will have an opportunity to retransmit before your deadline. For example, if you want an answer within 50 milliseconds, and your server takes no more than, oh, 35 milliseconds to generate the answer once it has the question, then you might consider an rto_min of something like 10 or 15 milliseconds to “cover” for the occasional individual packet loss.[6]

In terms of whether to use the per-route rto_min or the sysctl, here is what Linux’s documentation has to say about net.ipv4.tcp_rto_min_us[7]:

```
tcp_rto_min_us - INTEGER
        Minimal TCP retransmission timeout (in microseconds). Note that the
        rto_min route option has the highest precedence for configuring this
        setting, followed by the TCP_BPF_RTO_MIN socket option, followed by
        this tcp_rto_min_us sysctl.

        The recommended practice is to use a value less or equal to 200000
        microseconds.


        Possible Values: 1 - INT_MAX

        Default: 200000
```

You can choose between picking different rto_min values on a per-route (in-guest route in the guest Linux stack rather than Google Cloud Routes), or just setting the sysctl. Or even a mix of both. There should not be too much worry about setting an rto_min “too small” because TCP is adept at measuring round-trip-times, and it should get a pretty good idea of the round-trip-time during the three-way handshake establishing the connection. And *that* is still covered by a one-second retransmission timeout.

[6] Per this presentation, Google has been using 5ms internally for quite some time.
[7] You *are* going to upgrade to a kernel with support for that sysctl, right?-)

Tables

Even with charts looking at just "regions" of Disruption Duration, and different point shapes and … it can still be difficult to see what's happening, with many lines one on top of another. So, here are the numbers behind the charts.

The units for TCP Idle Duration are milliseconds. "Thick" means net.ipv4.tcp_thin_linear_timeouts was at its default value of 0 (zero). "Thin" means it was at a value of 1 (one). "rto_min <value>" denotes the value to which the corresponding version of "rto_min" was set.

The values in the tables which follow are the medians of the measured data points. With few exceptions, generally involving a 5 ms "rto_min" and simulating a 0.010s (10 millisecond) Disruption Duration, there was really no variation in the measured results.

Theory

Disruption Duration (s)	TCP Idle Thin rto_min 5ms	TCP Idle Thick rto_min 5ms	TCP Idle Thin rto_min 50ms	TCP Idle Thick rto_min 50ms	TCP Idle Thin rto_min 200ms	TCP Idle Thick rto_min 200ms
0.01	10	10	50	50	200	200
0.02	20	30	50	50	200	200
0.03	30	30	50	50	200	200
0.04	40	70	50	50	200	200
0.05	60	70	50	50	200	200
0.06	60	70	100	150	200	200
0.07	100	70	100	150	200	200
0.08	100	150	100	150	200	200
0.09	100	150	100	150	200	200
0.1	100	150	100	150	200	200
0.2	340	310	200	350	200	200
0.3	340	310	300	350	400	600
0.4	660	630	400	750	400	600
0.5	660	630	600	750	600	600
0.6	660	630	600	750	600	600
0.7	1300	1270	1000	750	800	1400
0.8	1300	1270	1000	1550	800	1400
0.9	1300	1270	1000	1550	1000	1400
1	1300	1270	1000	1550	1000	1400
1.1	1300	1270	1800	1550	1200	1400
1.2	1300	1270	1800	1550	1200	1400
1.3	1300	2550	1800	1550	1600	1400
1.4	2580	2550	1800	1550	1600	1400
1.5	2580	2550	1800	1550	1600	3200

Practice

These results were measured with the 6.5 kernel and used the route metric method for “rto_min.” Thus they do not include 5 ms “rto_min,” either thick or thin. You can see clearly how for Disruption Durations less than rto_min, the actual idle time ended-up being a bit more than rto_min thanks to various conversions and such in the kernel.

Disruption Duration (s)	TCP Idle Thin rto_min 50ms	TCP Idle Thick rto_min 50ms	TCP Idle Thin rto_min 200ms	TCP Idle Thick rto_min 200ms
0.01	59.9	59.8	207.9	207.9
0.02	59.8	59.8	207.8	207.9
0.03	59.8	59.9	207.8	207.9
0.04	59.8	59.9	207.8	207.7
0.05	59.8	59.9	207.8	207.8
0.06	119.5	119.4	207.8	207.9
0.07	119.8	119.9	207.8	207.9
0.08	119.9	119.8	207.9	207.9
0.09	119.8	120.0	207.9	208.0
0.1	119.9	119.9	207.8	207.9
0.2	239.9	235.9	207.8	207.9
0.3	359.8	463.8	415.9	415.9
0.4	419.8	463.8	415.7	415.8
0.5	595.8	939.8	623.8	853.1
0.6	823.8	941.6	623.8	851.9
0.7	823.9	942.2	832.0	850.1
0.8	823.9	940.6	831.7	852.0
0.9	1298.3	939.6	1039.8	1682.8
1	1297.7	1865.6	1039.8	1684.2
1.1	1298.6	1865.5	1247.9	1684.5
1.2	1296.3	1868.6	1248.0	1684.2
1.3	2227.7	1868.1	1455.8	1683.9
1.4	2228.0	1864.1	1455.8	1681.3
1.5	2226.4	1867.3	1663.8	1681.1

Near Present

As a reminder, this is with the 6.11 kernel, though similar results are *likely* to be had with the Ubuntu backport to 6.8.

Disruption Duration (s)	TCP Idle Thin rto_min 5ms	TCP Idle Thick rto_min 5ms	TCP Idle Thin rto_min 50ms	TCP Idle Thick rto_min 50ms	TCP Idle Thin rto_min 200ms	TCP Idle Thick rto_min 200ms
0.01	14.9	14.6	52.0	52.1	206.4	206.8
0.02	28.0	27.1	52.1	52.0	206.5	206.4
0.03	35.1	52.0	52.1	52.1	206.6	206.5
0.04	49.0	52.1	52.1	52.0	206.7	206.7
0.05	56.1	101.0	104.0	104.1	206.5	206.7
0.06	69.1	101.1	104.1	104.1	206.6	206.2
0.07	94.1	101.1	104.1	104.1	206.4	206.6
0.08	94.2	101.2	104.1	104.1	206.6	206.7
0.09	142.4	101.3	104.1	104.0	206.5	206.6
0.1	143.1	202.6	155.6	204.4	206.6	206.6
0.2	244.8	402.8	208.0	212.5	209.0	209.0
0.3	446.8	404.7	312.1	421.7	416.9	416.5
0.4	446.6	615.6	416.1	421.3	416.8	416.7
0.5	615.8	615.5	525.8	615.7	616.6	616.6
0.6	615.9	615.6	616.0	615.7	617.0	616.8
0.7	838.9	796.9	733.7	837.7	832.6	824.7
0.8	838.6	1478.3	1149.6	837.8	832.7	824.8
0.9	1475.2	1477.7	1149.7	1476.4	1040.8	1476.8
1	1476.2	1477.4	1149.6	1477.5	1040.8	1478.8
1.1	1477.2	1476.5	1149.4	1478.9	1248.8	1476.3
1.2	1477.1	1476.8	1476.3	1477.4	1248.7	1477.8
1.3	1477.3	1476.9	1478.9	1476.2	1455.9	1478.1
1.4	1475.8	1476.4	1477.4	1476.7	1456.6	1479.1
1.5	1667.9	1627.7	2024.9	1713.7	1664.8	1686.6

Methodology

We create two, n2-standard-128 VMs in a Google Cloud Availability Zone. Here it happens to have been us-east5-c but we would expect similar results in any Availability Zone. We would also expect similar results with other VM types/sizes. One VM called "ping." The other called "pong." In each we run Ubuntu 22.04.4 LTS with a 6.5.0-1025-gcp kernel, or a Beta version of Ubuntu 24.10 with a 6.11.0-1001-gcp kernel, and use the gvnic vNIC type, though vNIC types don't really play any part in this.

We ensure that a netperf benchmark netserver process is running on "pong." We also set things up so a script running in "ping" may ssh to "pong" and execute commands.

On "ping" we start a script which can simulate a communication disruption between ping and pong by blocking packets with the specific TCP port number we use for the netperf TCP_RR test runs between "ping" and "pong." Then on "ping" we start another script which will choose at random between 5, 50 and 200 ms for rto_min, and "0" and "1" for net.ipv4.tcp_thin_linear_timeouts and set those values on both "ping" and "pong" in a kernel-version-appropriate way so they will affect a given run of a netperf TCP_RR benchmark.

Neither of these scripts should be considered the acme of how to do such things. They were, however, sufficient to the task at hand.

simulate_blackout_nft.sh

Based on results with 5ms rto_min it is likely that this script doesn't do "great" simulating a 10 millisecond Disruption Duration. It probably goes a little long. But it has done well enough for our purposes here.

```
#!/usr/bin/bash

BLACKOUT_FILE=$1

#set -x

NFT_INPUT="/tmp/nft_input"
mkfifo ${NFT_INPUT}

# set up our pipe to feed the nft command in "interactive" mode
nft -i < ${NFT_INPUT} > /tmp/nft_output.log 2>&1 &
#tail -f ${NFT_INPUT} > /dev/null &
sleep 999999999 > ${NFT_INPUT} &

while [ 1 ]
do
        if [ ! -f ${BLACKOUT_FILE} ]
        then
```

```
                echo "`date`: Waiting on BLACKOUTFILE"
                printf "flush table filter\n" > ${NFT_INPUT}
                BLACKOUT_TIME=0
        else
                BLACKOUT_TIME=`cat ${BLACKOUT_FILE}`
                echo "`date`: BLACKOUT_TIME ${BLACKOUT_TIME}"
                if [ ${BLACKOUT_TIME} != "0" ]
                then
                        # echo "t1 `date +%s.%N`"
                        printf "add rule ip filter INPUT tcp dport 65432 counter drop\n" > ${NFT_INPUT}
                        # echo "t2 `date +%s.%N`"
                        printf "add rule ip filter OUTPUT tcp dport 65432 counter drop\n" >
${NFT_INPUT}
                        # echo "t3 `date +%s.%N`"
                        sleep ${BLACKOUT_TIME}
                        # echo "t4 `date +%s.%N`"
                        printf "flush table filter\n" > ${NFT_INPUT}
                        # echo "t5 `date +%s.%N`"
                else
                        printf "flush table filter\n" > ${NFT_INPUT}
                fi
        fi
        # subtract-out how long we slept for blackout
        SLEEP_TIME=`echo $BLACKOUT_TIME | awk '{print 5-$1}'`
        sleep ${SLEEP_TIME}
done
```

probe_blackouts.sh

This is as it was for the runs with a 6.5-based kernel. The changes for the “Near Present” results using the 6.11 kernel are simply to add another RTOS, and then replace the route commands setting the “rto_min” metric with sysctl commands setting the net.ipv4.tcp_rto_min_us sysctl to the desired number of microseconds.

```
#!/usr/bin/bash

RTOS[0]=50
RTOS[1]=200
NUM_RTOS=2

BLACKOUTS[0]=0.010
BLACKOUTS[1]=0.020
BLACKOUTS[2]=0.030
BLACKOUTS[3]=0.040
BLACKOUTS[4]=0.050
BLACKOUTS[5]=0.060
```

```
BLACKOUTS[6]=0.070
BLACKOUTS[7]=0.080
BLACKOUTS[8]=0.090
BLACKOUTS[9]=0.100
BLACKOUTS[10]=0.200
BLACKOUTS[11]=0.300
BLACKOUTS[12]=0.400
BLACKOUTS[13]=0.500
BLACKOUTS[14]=0.600
BLACKOUTS[15]=0.700
BLACKOUTS[16]=0.800
BLACKOUTS[17]=0.900
BLACKOUTS[18]=1.000
BLACKOUTS[19]=1.100
BLACKOUTS[20]=1.200
BLACKOUTS[21]=1.300
BLACKOUTS[22]=1.400
BLACKOUTS[23]=1.500
BLACKOUTS[24]=0
NUM_BLACKOUTS=25

OUTPUT="-o
THROUGHPUT,ELAPSED_TIME,RT_LATENCY,MIN_LATENCY,MAX_LATENCY,P50_LATENCY,P90_LATENCY,P99_LATENCY,MEAN_LAT
ENCY,STDDEV_LATENCY,local_transport_retrans,remote_transport_retrans,result_brand"

for i in `seq $1 $2`
do
        RTO=${RTOS[`expr $RANDOM % $NUM_RTOS`]}
        BLACKOUT=${BLACKOUTS[`expr $RANDOM % $NUM_BLACKOUTS`]}
        DO_THIN=0
        if [ $RANDOM -lt 16384 ]
        then
                DO_THIN=1
        fi
        THIN_CMD="sudo sysctl --quiet -w net.ipv4.tcp_thin_linear_timeouts=${DO_THIN}"
        sudo ip route replace 10.202.0.20/32 rto_min ${RTO}ms dev ens4 via 10.202.0.1
        ${THIN_CMD}
        ssh pong "${THIN_CMD}; sudo ip route replace 10.202.0.19/32 rto_min ${RTO}ms dev ens4 via
10.202.0.1"
        echo "0" > blackout
        # the blackout script running on the other side will take up to five seconds to notice
        sleep 5.5
        netperf $HDR -H pong -t TCP_RR -l 60 -B "${RTO},${BLACKOUT},${DO_THIN},${i}" -- -P 65432 -r 1,2
$OUTPUT | sed 's/\"//g' &
        sleep 1
        echo $BLACKOUT > blackout
        wait
        HDR="-P 0"
done
```

APPENDIX - Tuning SYN Retransmissions

Problem - The Initial Timeout

The retransmission logic for TCP is described in RFC 6298 - "Computing TCP's Retransmission Timer". Specifically, it specifies that the packet retransmission timeout for any TCP connection should be computed dynamically, based on recent observations of packet RTT (round-trip time) for the particular connection.

This presents a problem for the initial three-way handshake to connect - because when sending the initial SYN (or SYN-ACK), by definition the particular connection has not yet observed a single packet-round-trip. Thus, the stack is forced to choose a relatively arbitrary value for this initial timeout, chosen large enough to avoid network-clogging spurious retransmissions. RFC 6298 specifies using one full second as an initial timeout as sufficient for nearly all real-world network paths - and this value has long been hardcoded into the Linux kernel.

While this "works" in the general sense of universally limiting the possibility of spurious SYN retransmissions, it significantly (and unnecessarily) harms the performance of connections where the system designer can know a priori that the network round-trip time is well below one second - e.g. when making connections within the same availability zone or even the same region in a cloud deployment. For example, even if one can be *certain* that the maximum round-trip time of a packet between endpoints is significantly less than, say, 20ms - when a single SYN packet is lost, the connecting TCP stack must still wait a full second before attempting to restart the handshake with a SYN retransmit.

Overriding the Initial RTO

Luckily, starting with kernel version 4.13, Linux has added functionality to override this default initial RTO (retransmission time-out) on a per-connection basis through an eBPF (extended Berkeley Packet Filter) hook which can be dynamically installed.

In order to leverage this functionality, your Linux deployment must:

- use kernel version 4.13 or later (or backport the functionality)
- use cgroups v2

The precise details of necessary packages to construct and install such a program can vary slightly from distro to distro - and there are a variety of methods to load and configure eBPF programs as well - but for purposes of illustration, we'll describe one such implementation on an Ubuntu 24.04 instance in detail.

eBPF Example

Setup

Strictly for the purposes of example, we set up a VM in us-central1 with the Ubuntu 24.04 LTS (Noble) image. For convenience, the VPC is an "auto mode" network with subnets for each region - with 10.128.0.0/20 configured for us-central-1, and we know from inspection that 10.128.0.4 is an unprovisioned address in this range (we will attempt to connect to this IP to demonstrate SYN retransmit behavior).

Because we know that the 10.128.0.0/20 refers to endpoints within the same region as the VM, we know that the network paths to them should not be subject to as much RTT uncertainty as those reachable over the backbone WAN, or Internet. Thus, we can safely reason that round-trips will remain significantly below, say, 100ms to this range - and we'd like to override the initial SYN RTO specifically for this CIDR.

To help build our eBPF program, we'll install clang and libbpf-dev:

```
sudo apt-get update
sudo apt-get install -y clang libbpf-dev
```

To load, attach, and configure our hook, we'll also be using the "bpftool" utility - which for Ubuntu is available as part of the "linux-tools" package and is generally included in the image.

[NOTE that as of August 2025 the Ubuntu 24.04 LTS image has been upgraded to kernel 6.14 and the package system has a bug which leaves out bpftool. There are bugs (see this) open to this effect, but for now one can simply install the linux-tools-generic-6.8 package and use the bpftool installed in the 6.8 subdirectory under /usr/lib/linux-tools - symlinking it to "bpftool" under the /usr/lib/linux-tools/linux-tools-$(uname -r) subdirectory]

Create set_syn_rto.c

Despite nominally being written in "C", we'll compile this program to eBPF and load it into our kernel.

The structure is simple - we define a longest-prefix-match trie lookup map (for storing CIDR ranges where we'd like to override the default initial TCP RTO) in a ".maps" section, and a function in a "sockops" section which will be called from various hooks within the TCP stack.

The "sockops" function checks whether it was called for the purposes of providing a non-default initial timeout - and if so, for connections using IPv4 (the "AF_INET" address family) it will check and see whether a non-default initial timeout has been provided for a CIDR range which includes that address, and if so return that value. NOTE that this value should be in "jiffies" - which means you need to know the CONFIG_HZ value for the kernel you're using (the Ubuntu 24.04 image is built with CONFIG_HZ=1000, so jiffies and milliseconds are interchangeable in this instance)

```
/*
  bpf.h defines the type "bpf_sock_ops" and various enum values used below
*/
#include <linux/bpf.h>
/*
  bpf_helpers.h from libbpf-dev defines things like:
    bpf_map_lookup_elem (and other kernel helper functions)
    map definition helpers __uint(name, val) etc…
    SEC(name) macro
*/
#include <bpf/bpf_helpers.h>

#define AF_INET 2 // IPv4 address family

// Define the key structure for IPv4 lpm lookup
struct ipv4_lpm_key {
    __u32 prefixlen; // Prefix length (e.g., 32 for a /32 IPv4 address)
    union {
    __u32 ip;            // The IPv4 address (stored in network byte order)
    __u8 ip_byte[4];
    };
};

// Define the LPM map
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE); // The map type
    __type(key, struct ipv4_lpm_key);    // The key structure
    __type(value, __u32);                 // The value - the RTO in jiffies
    __uint(map_flags, BPF_F_NO_PREALLOC); // Must be set when creating this map type
    __uint(max_entries, 255);             // Maximum number of entries
} ipv4_lpm_map SEC(".maps");              // Define the map within the ".maps" section


SEC("sockops")
int bpf_get_tcp_initial_rto(struct bpf_sock_ops *skops)
{
  struct ipv4_lpm_key key4;
  __u32 *p_timeout_jiffies;

  const int op = (int) skops->op;

  if (op == BPF_SOCK_OPS_TIMEOUT_INIT) {
    if (skops->family == AF_INET) {
       key4.prefixlen = 32;
       key4.ip = skops->remote_ip4;

       p_timeout_jiffies = bpf_map_lookup_elem(&ipv4_lpm_map, &key4);
       if (p_timeout_jiffies) {
         skops->reply = *p_timeout_jiffies;
       }
     }
  }

  return 1;
}

char _license[] SEC("license") = "GPL";
int _version SEC("version") = 1;
```

Build, Load, and Attach

Use "clang" to compile "set_syn_rto.c" to eBPF:

```
clang -target bpf \
  -I/usr/include/$(uname -m)-linux-gnu \
  -g -O2 \
  -c set_syn_rto.c \
  -o set_syn_rto.o
```

Then use "bpftool" to load it into the kernel, pinned to the filesystem under "sys/fs/bpf":

```
sudo bpftool prog load set_syn_rto.o /sys/fs/bpf/set_syn_rto
```

This will load both our eBPF "hook" routine in the "sockops" section, and the LPM trie lookup we added to the ".maps" section. We can query the loaded program via:

```
sudo bpftool -f prog show pinned /sys/fs/bpf/set_syn_rto
```

Which produces output similar to the following:

```
55: sock_ops  name bpf_get_tcp_initial_rto  tag 2da043aebfc52ee9
       loaded_at 2025-08-21T17:13:20+0000  uid 0
       xlated 176B  jited 115B  memlock 4096B  map_ids 1
       pinned /sys/fs/bpf/set_syn_rto
       btf_id 40
```

Importantly, it shows that it references a map with ID 1, which we can also query:

```
sudo bpftool -f map show id 1
```

which produces

```
1: lpm_trie  name ipv4_lpm_map  flags 0x1
       key 8B  value 4B  max_entries 255  memlock 0B
       btf_id 40
```

(i.e. the map we defined in the ".maps" section of our object file)

With the program now loaded, before the kernel will invoke it, we must attach it to a cgroup. Here, we attach it to the root cgroup so that will effectively be inherited (and invoked) in all cgroups:

```
sudo bpftool cgroup attach /sys/fs/cgroup/ sock_ops pinned "/sys/fs/bpf/set_syn_rto"
```

At this point, we have not yet added any RTO override entries to our "ipv4_lpm_map", but this is a good time to verify that SYN retransmits are indeed using the default of 1 second.

Connecting to our unprovisioned IP, using python3:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("10.128.0.4",80))
```

We can observe the initial SYN and the first retransmit with tcpdump:

```
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), snapshot length 262144 bytes
2025-08-21 17:33:52.884082 IP 10.128.0.16.39558 > 10.128.0.4.80: Flags [S], seq 1723423107,
win 65320, options [mss 1420,sackOK,TS val 1745905507 ecr 0,nop,wscale 7], length 0
2025-08-21 17:33:53.913078 IP 10.128.0.16.39558 > 10.128.0.4.80: Flags [S], seq 1723423107,
win 65320, options [mss 1420,sackOK,TS val 1745906536 ecr 0,nop,wscale 7], length 0
...
```

Here we clearly see the first retransmit sent after one full second.

At this point, we can add an entry to our LPM map via bpftool:

```
sudo bpftool map update id 1 key 20 0 0 0 10 128 0 0 value 100 0 0 0
```

(NOTE that keys and values here are entered per byte, in big-endian order - and given our struct defined for the key - this is a 32-bit prefixlen of 20 for 10.128.0.0, with a 32-bit value of 100 jiffies)

We can check that the value was added/interpreted properly via:

```
sudo bpftool map dump id 1
```

which produces:

```
[{
        "key": {
            "prefixlen": 20,
            "": {
                "ip": 32778,
                "ip_byte": [10,128,0,0]
            }
        },
        "value": 100
    }
]
```

And then when we try to connect to our unprovisioned IP again, we get:

```
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), snapshot length 262144 bytes
2025-08-21 17:44:31.290657 IP 10.128.0.16.51232 > 10.128.0.4.80: Flags [S], seq 1839542872,
win 65320, options [mss 1420,sackOK,TS val 1746543913 ecr 0,nop,wscale 7], length 0
2025-08-21 17:44:31.393048 IP 10.128.0.16.51232 > 10.128.0.4.80: Flags [S], seq 1839542872,
win 65320, options [mss 1420,sackOK,TS val 1746544016 ecr 0,nop,wscale 7], length 0
...
```

and see that the SYN is now retransmitted after only a 100ms wait.

Trying now to an unreachable IP outside of 10.128.0.0/20:

```
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on ens4, link-type EN10MB (Ethernet), snapshot length 262144 bytes
2025-08-21 17:46:52.414919 IP 10.128.0.16.41236 > 1.2.3.4.80: Flags [S], seq 603472496, win
65320, options [mss 1420,sackOK,TS val 1687417832 ecr 0,nop,wscale 7], length 0
2025-08-21 17:46:53.433068 IP 10.128.0.16.41236 > 1.2.3.4.80: Flags [S], seq 603472496, win
65320, options [mss 1420,sackOK,TS val 1687418851 ecr 0,nop,wscale 7], length 0
...
```

we see that these connections still use the default timeout of one second.

Removing the eBPF Hook

To remove the eBPF hook, simply detach from the cgroup:

```
sudo bpftool cgroup detach /sys/fs/cgroup/ sock_ops pinned "/sys/fs/bpf/set_syn_rto"
```

and remove the reference to the program from the filesystem:

```
sudo rm /sys/fs/bpf/set_syn_rto
```

Acknowledgements

The author(s) would like to thank the following for their assistance and feedback:

- George Powers
- Neal Cardwell
- Eric Dumazet
- Yuchung Cheng
- Kevin Young
- Philip Wells
- Sumit Singh