Mininet /Netem Emulation Pitfalls
A Multipath TCP Scheduling Experience

Alexander Frömmgen

November 17, 2017

Abstract

Do you use Mininet for network experiments? This article summarizes our experience with a notable differences between Mininet and real-world measurements. During our work on Multipath TCP scheduling, we experienced systematic and reproducible differences between real-world measurements and Mininet experiments. Our investigation shows that these differences are caused by the interplay of Mininet, netem and the queue management in the networking stack, i.e., TSQ.

1 Introduction

Multipath TCP [3] is a recent TCP evolution which enables the usage of multiple paths for a single logical Multipath TCP connection. During our work on flexible Multipath TCP scheduling [4], we used Mininet [5] and the Linux Kernel Multipath TCP implementation [1] for development and systematic evaluations. We chose Mininet, as it is convenient to run a real network stack in user defined topologies. Mininet is widely used in current efforts to reproduce network research results [1]. We were surprised when our Mininet experiments and real-world experiments showed significantly different behavior.[2]

2 A Simple Example

For a basic MPTCP measurement example, let's consider the following topology with two paths (Figure 1). The first path has a round-trip time of 40ms, the second path has a round-trip time of 80ms. The topology in Figure 1 can be easily specified with the Mininet Python API as shown in Listing 1.

Now, let's run an application on top of this topology. Our example application starts with a short handshake and eventually sends 10kb of data. At this point, Multipath TCP managed to establish one subflow per path, so we have two subflows available. In a typical Internet setup, 10kb of data require roughly 8 packets. Thus, MPTCP has to schedule 8 packets on two subflows, and both subflows have an initial congestion window of 10 packets [2].

[1] `https://reproducingnetworkresearch.wordpress.com`

[2] This experience report is available online at `https://progmp.net/mininetPitfalls.html`

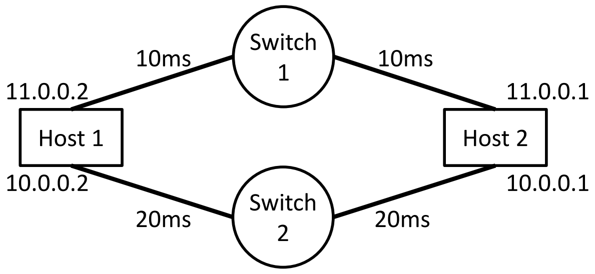


Figure 1: Basic example Mininet topology with two paths.

```
class StaticTopo(Topo):
    def build(self):
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')

        /* first path */
        s1 = self.addSwitch('s1')
        self.addLink(h1, s1, bw=100, delay="10ms")
        self.addLink(h2, s1, bw=100, delay="10ms")

        /* second path */
        s2 = self.addSwitch('s2')
        self.addLink(h1, s2, bw=100, delay="20ms")
        self.addLink(h2, s2, bw=100, delay="20ms")
```

Listing 1: Python code snippet for the topology in Figure 1.

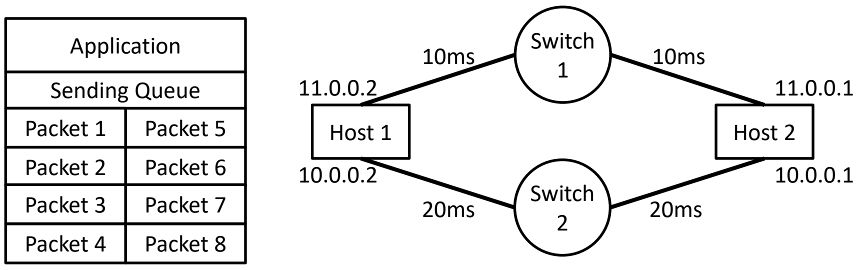


Figure 2: Sending queue with 8 packets in the Mininet example topology with two paths.

2.1 Mininet Evaluation

Using Mininet and our example application [3], we found that the default Multipath TCP scheduler schedules all 8 packets on the lowest round-trip time subflow (Figure 3). We used wireshark to analyze the sent packets (Figure 4).

[3]The source of the evaluation is available at `https://progmp.net/mininet_tsq.zip`

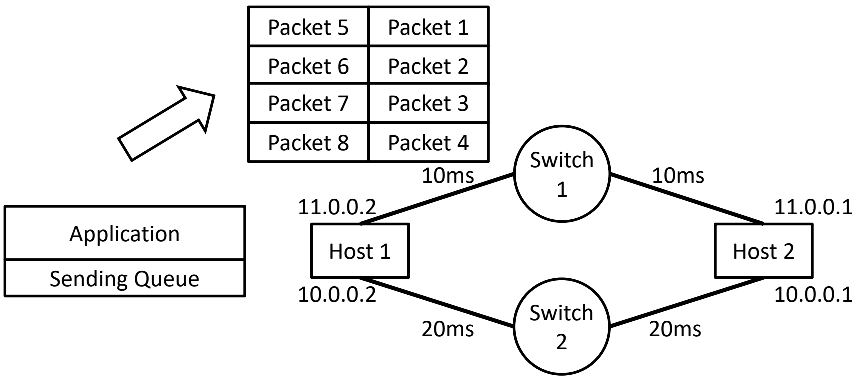
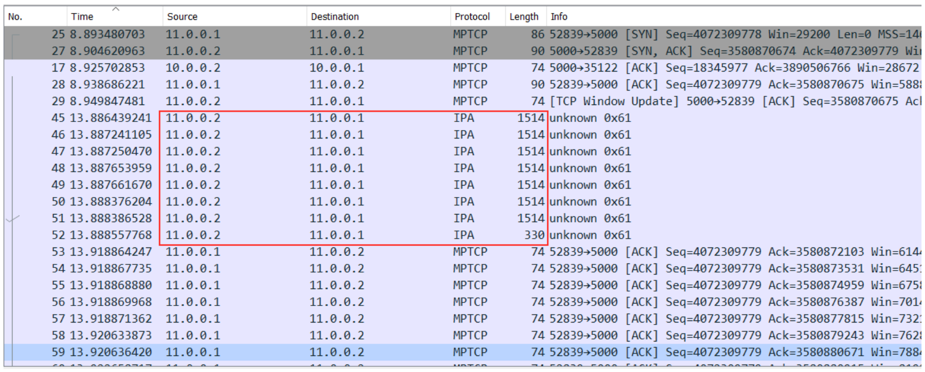


Figure 3: In the Mininet setup, all 8 packets are scheduled on the first subflow.



No.	Time	Source	Destination	Protocol	Length	Info
25	8.893480703	11.0.0.1	11.0.0.2	MPTCP	86	52839→5000 [SYN] Seq=4072309778 Win=29200 Len=0 MSS=14
27	8.904620963	11.0.0.2	11.0.0.1	MPTCP	90	5000→52839 [SYN, ACK] Seq=3580870674 Ack=4072309779 Wi
17	8.925702853	10.0.0.2	10.0.0.1	MPTCP	74	5000→35122 [ACK] Seq=18345977 Ack=3890506766 Win=28672
28	8.938686221	11.0.0.1	11.0.0.2	MPTCP	90	52839→5000 [ACK] Seq=4072309779 Ack=3580870675 Win=588
29	8.949847481	11.0.0.2	11.0.0.1	MPTCP	74	[TCP Window Update] 5000→52839 [ACK] Seq=3580870675 Ac
45	13.886439241	11.0.0.2	11.0.0.1	IPA	1514	unknown 0x61
46	13.887241105	11.0.0.2	11.0.0.1	IPA	1514	unknown 0x61
47	13.887250470	11.0.0.2	11.0.0.1	IPA	1514	unknown 0x61
48	13.887653959	11.0.0.2	11.0.0.1	IPA	1514	unknown 0x61
49	13.887661670	11.0.0.2	11.0.0.1	IPA	1514	unknown 0x61
50	13.888376204	11.0.0.2	11.0.0.1	IPA	1514	unknown 0x61
51	13.888386528	11.0.0.2	11.0.0.1	IPA	1514	unknown 0x61
52	13.888557768	11.0.0.2	11.0.0.1	IPA	330	unknown 0x61
53	13.918864247	11.0.0.1	11.0.0.2	MPTCP	74	52839→5000 [ACK] Seq=4072309779 Ack=3580872103 Win=614
54	13.918867735	11.0.0.1	11.0.0.2	MPTCP	74	52839→5000 [ACK] Seq=4072309779 Ack=3580873531 Win=645
55	13.918868880	11.0.0.1	11.0.0.2	MPTCP	74	52839→5000 [ACK] Seq=4072309779 Ack=3580874959 Win=675
56	13.918869968	11.0.0.1	11.0.0.2	MPTCP	74	52839→5000 [ACK] Seq=4072309779 Ack=3580876387 Win=701
57	13.918871362	11.0.0.1	11.0.0.2	MPTCP	74	52839→5000 [ACK] Seq=4072309779 Ack=3580877815 Win=732
58	13.920633873	11.0.0.1	11.0.0.2	MPTCP	74	52839→5000 [ACK] Seq=4072309779 Ack=3580879243 Win=762
59	13.920636420	11.0.0.1	11.0.0.2	MPTCP	74	52839→5000 [ACK] Seq=4072309779 Ack=3580880671 Win=788

Figure 4: Wireshark trace for the Mininet example of Figure 3. Note that all packets are scheduled on the first path (IP 11.0.0.2).

2.2 Real-World Evaluation

When we used the same application in a comparable real-world setup, we experienced different behavior. In the real-world experiments, packets were distributed over multiple paths, although the round-trip times showed the same differences as in our emulation setup (Figure 5).

We repeated the experiment multiple times, showing reproducible results. So why do our Mininet experiments and the real-world measurements show significantly different behavior?

3 Explanation

Digging into the Multipath TCP implementation, we found that the scheduler stops using the first path due to TCP small queues (TSQ) [4]. TSQ is an effort to reduce queue sizes in the network stack. Large queues in the network stack are known to increase latency due to queuing delays. During the development of Multipath TCP, a commit was added to avoid sending packets on subflows

[4]https://lwn.net/Articles/507065

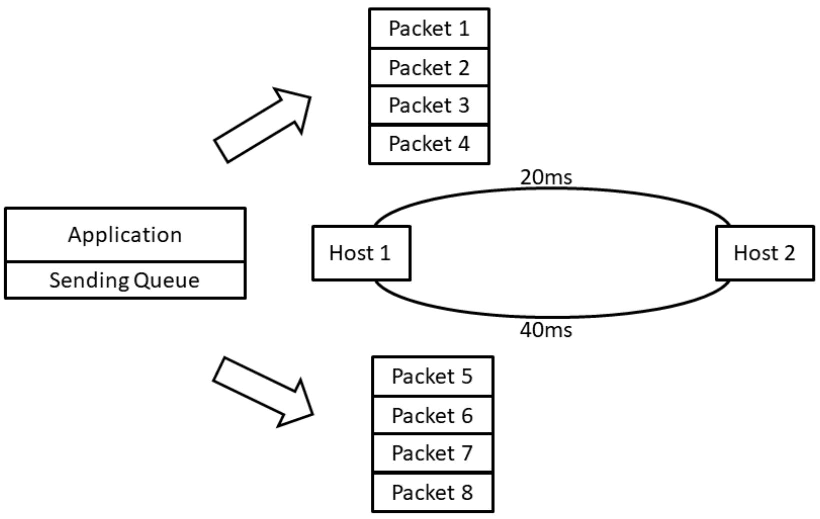


Figure 5: In the real-world environment, packets are split on both subflow.

which are considered throttled by TSQ (Listing 2) [5]. For many application scenarios, this is a reasonable design decision for Multipath TCP.

```
/* If TSQ is already throttling us, do not send on this subflow. When
 * TSQ gets cleared the subflow becomes eligible again.
 */

if (test_bit(TSQ_THROTTLED, &tp->tsq_flags))
    return 0;
```

Listing 2: The Multipath TCP default scheduler checks if a subflow is already throttled by TSQ.

But why don't we see this behavior in our Mininet emulation? This requires a more detailed analysis of the interactions of Mininet, TSQ and Multipath TCP. Mininet relies on netem [6] for delay emulation. Checking the implementation of netem [7], we found that packets which are delayed by netem are not considered queued by TSQ (Listing 3).

```
/* If a delay is expected, orphan the skb. (orphaning usually takes
 * place at TX completion time, so _before_ the link transit delay)
 */
if (q->latency || q->jitter || q->rate)
    skb_orphan_partial(skb);
```

Listing 3: The netem implementation removes delayed packets from the sending queue.

Thus, Multipath TCP never experiences a TSQ throttled subflow in Mininet setups with delayed links at the sender. To substantiate our claim, we repeated

[5]https://github.com/multipath-tcp/mptcp/commit/5c278893b37fe48c66ff226793607687b8482ba9

[6]https://wiki.linuxfoundation.org/networking/netem

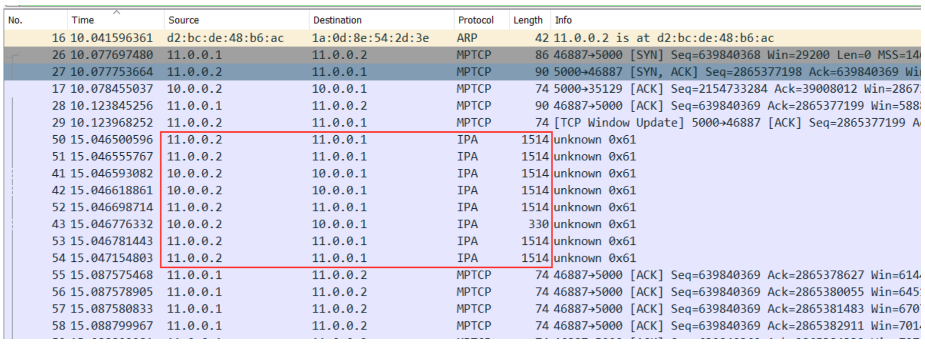
[7]https://elixir.free-electrons.com/linux/v4.14/source/net/sched/sch_netem.c#L462

the previous Mininet emulation with a slightly modified topology (Listing 4). In this topology, the first link at the sender does not use netem.

```
class StaticTopo(Topo):
    def build(self):
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')

        /* first path */
        s1 = self.addSwitch('s1')
        self.addLink(h1, s1, bw=100)
        self.addLink(h2, s1, bw=100, delay="20ms")

        /* second path */
        s2 = self.addSwitch('s2')
        self.addLink(h1, s2, bw=100)
        self.addLink(h2, s2, bw=100, delay="40ms")

```

Listing 4: Python code snippet for a modified topology which has no netem delay on the sender's link.

The wireshark trace in Figure 6 shows the expected behavior, which is comparable to our real-world measurements.



```
No.        Time            Source             Destination        Protocol  Length Info
        16 10.041596361    d2:bc:de:48:b6:ac  1a:0d:8e:54:2d:3e  ARP           42 11.0.0.2 is at d2:bc:de:48:b6:ac
        26 10.077697480    11.0.0.1           11.0.0.2           MPTCP         86 46887→5000 [SYN] Seq=639840368 Win=29200 Len=0 MSS=14
        27 10.077753664    11.0.0.2           11.0.0.1           MPTCP         90 5000→46887 [SYN, ACK] Seq=2865377198 Ack=639840369 Wi
        17 10.078455037    10.0.0.2           10.0.0.1           MPTCP         74 5000→35129 [ACK] Seq=2154733284 Ack=39008012 Win=2867
        28 10.123845256    11.0.0.1           11.0.0.2           MPTCP         90 46887→5000 [ACK] Seq=639840369 Ack=2865377199 Win=588
        29 10.123968252    11.0.0.2           11.0.0.1           MPTCP         74 [TCP Window Update] 5000→46887 [ACK] Seq=2865377199 A
        50 15.046500596    11.0.0.2           11.0.0.1           IPA         1514 unknown 0x61
        51 15.046555767    11.0.0.2           11.0.0.1           IPA         1514 unknown 0x61
        41 15.046593082    10.0.0.2           10.0.0.1           IPA         1514 unknown 0x61
        42 15.046618861    10.0.0.2           10.0.0.1           IPA         1514 unknown 0x61
        52 15.046698714    11.0.0.2           11.0.0.1           IPA         1514 unknown 0x61
        43 15.046776332    10.0.0.2           10.0.0.1           IPA          330 unknown 0x61
        53 15.046781443    11.0.0.2           11.0.0.1           IPA         1514 unknown 0x61
        54 15.047154803    11.0.0.2           11.0.0.1           IPA         1514 unknown 0x61
        55 15.087575468    11.0.0.1           11.0.0.2           MPTCP         74 46887→5000 [ACK] Seq=639840369 Ack=2865378627 Win=614
        56 15.087578905    11.0.0.1           11.0.0.2           MPTCP         74 46887→5000 [ACK] Seq=639840369 Ack=2865380055 Win=645
        57 15.087580833    11.0.0.1           11.0.0.2           MPTCP         74 46887→5000 [ACK] Seq=639840369 Ack=2865381483 Win=670
        58 15.088799967    11.0.0.1           11.0.0.2           MPTCP         74 46887→5000 [ACK] Seq=639840369 Ack=2865382911 Win=701
```

Figure 6: Mininet experiment without netem on the sender's links. This Mininet experiment shows the same behavior as our real world measurements, i.e., packets are send from both source IP adresses 11.0.0.2 and 10.0.0.2.

4 Conclusion

Our experiments show that we have to be careful when transferring emulation results to real-world applications. Evaluation results should always be (at least) supported by real-world measurements. We are still convinced that Mininet is great for network experiments and reproducible network research, but we feel confirmed that sanity checks are essential.

Did we misuse Mininet? As our results show that our initial Mininet setup does not reproduce real-world situations, we probably used it wrongly. However, we just followed established online examples here, so chances are high that others will fall into the same pitfalls as well. Note that this problem is not specific to Mininet, but might appear in comparable netem setups as well.

Are there more pitfalls? Yes. We are preparing some more posts. If you struggled with Mininet pitfalls, feel free to contact us, we might prepare a collection :-)

Talking about MPTCP. How can I get the behavior MPTCP showed in Mininet in the real-world? There are a lot of options. You might change the queue sizes considered by TSQ or change the MPTCP scheduler, e.g., using ProgMP [8].

5 Acknowledgments

This work has been funded by the German Research Foundation (DFG) as part of the project C2 in the Collaborative Research Center (SFB) 1053 MAKI.

We would like to thank the Mininet Mailing list replier [9], who supported our analysis.

References

[1] S. B. C. Paasch. Multipath TCP in the Linux Kernel. available from `http://www.multipath-tcp.org`.

[2] J. Chu, N. Dukkipathi, Y. Cheng, and M. Mathis. Increasing TCP's Initial Window. RFC 6928.

[3] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824.

[4] A. Frömmgen, A. Rizk, T. Erbshäusser, M. Weller, B. Koldehofe, A. Buchmann, and R. Steinmetz. A Programming Model for Application-defined Multipath TCP Scheduling. In *ACM/IFIP/USNIX Middleware*, 2017.

[5] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments using Container-based Emulation. In *CoNEXT*, 2012.

[8] `https://progmp.net`

[9] `https://mailman.stanford.edu/pipermail/mininet-discuss/2017-April/007425.html`