Something

IN3260

Oskar Haukebøe

December 12, 2023

Contents

1 Introduction

Network performance issues are a pervasive nuisance that can often leave users perplexed about their origin—be it within their local WiFi or due to external factors such as distant servers. Identifying these bottlenecks is critical in maintaining a proficient network experience.

One project that attempts to solve this challenge is NETHINT [1], which leverages passive WiFi traffic analysis to evaluate latency and packet loss, providing insights into whether the congestion lies in the local network or not.

This assignment builds upon the work of NETHINT by empirically testing its capability to discern the location of Bottlenecks within a controlled environment. Utilization of a TEACUP testbed [2] facilitates a systematic approach to simulating various network scenarios with different congestion points. Through automated experiments on real machines, this assignment aims to critically evaluate NETHINT's proficiency in

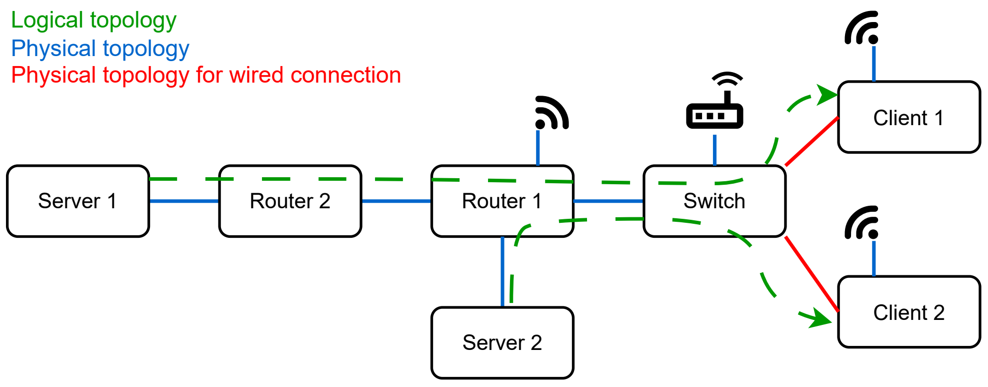


Figure 1: Topology of the testbed

accurately identifying whether local WiFi is the limiting factor. The TEACUP configuration used is available at [3].

2 Testbed configuration

The primary distinctive feature of NETHINT is its utilization of passive network measurements, ensuring no interference with the network [4]. It achieves this by monitoring WiFi traffic and analyzing packet timings for each device on the network. This allows for a comparative assessment of the packet delay across different devices, which then can be used to determine whether the devices on the network have a common bottleneck or not. Should a common bottleneck be identified, it likely indicates that congestion exists within the local area network (LAN).

To assess NETHINT's ability to recognize local WiFi bottlenecks, we configured a network topology as shown in Figure 1. All nodes in this topology are physical computers running Debian Linux, aside from the switch. They are managed by another computer not depicted in the figure, using TEACUP. This setup enables automated generation and control of network traffic throughput and delay at routers, as well as traffic logging.

2.1 Network Devices and Configuration

- *Servers:* Two servers (Server 1 and Server 2) for generating traffic.
- *Clients:* Two clients (Client 1 and Client 2) to act as traffic endpoints.
- *Routers:* Two routers with Router 1 being the entry point for both clients.
- *Switch:* A switch to connect the clients with Router 1, enabling bandwidth limitation at the router level.
- *WiFi AP:* An access point for the wireless segment of the network, set to channel 3 without encryption to accommodate NETHINT's capabilities.
- *Monitor Mode:* Router 1's WiFi card, set to monitor mode for traffic sniffing, is also configured to channel 3 to monitor the WiFi AP.

This setup models a scenario where Client 2 streams video from Server 2, while Client 1 consumes bandwidth for different activities from Server 1. The objective is to discern whether traffic between Server 1 and Client 1 interferes with the traffic between Server 2 and Client 2.

2.1.1 Configuration details

We use a switch to connect the two clients to the first router, allowing both clients to access Router 1 through the same network interface for wired connections. This setup facilitates bandwidth limitation at the router, enabling both clients to share the available bandwidth.

The WiFi AP is connected to the switch and provides an unencrypted WiFi network. The lack of encryption is due to NETHINT's inability to decrypt WiFi traffic. Both clients connect to this network, thus gaining access to the two servers. Router 1's WiFi card, configured in monitor mode, is employed by NETHINT to capture all the WiFi traffic between the clients and the WiFi AP. To enable traffic monitoring, the WiFi interface on Router 1 and the WiFi AP are both set to channel 3.

For the machines to recognize how to reach each other, they had to be configured to know where the packets should be sent. Client 2, for example, was configured in the file `/etc/network/interfaces.d/vlan11-iface` as follows:

```
auto enp36s0
iface enp36s0 inet static
      address 172.16.12.5/24
      up route add -net 172.16.10.0 netmask 255.255.255.0 gw
          ↪ 172.16.12.254 || true
      up route add -net 172.16.11.0 netmask 255.255.255.0 gw
          ↪ 172.16.12.254 metric 10 || true
      up route add -net 10.10.12.0 netmask 255.255.255.0 gw
          ↪ 172.16.12.254 metric 10 || true
```

The IP address `172.16.12.254` corresponds to the network interface of Router 1 that faces the switch. The address `172.16.11.0` is linked with Server 2, and `10.10.12.0` with Server 1. This configuration indicates to Client 2 that it can reach the servers via Router 1. The address `172.16.10.0` represents the interface on Router 1 facing Router 2 and was necessary for TEACUP operations, as it requires initial ping tests between certain nodes.

While the above outlines wired connectivity, the main tests needed to run wirelessly, hence the specification of `metric 10` in the routing commands. Given that clients can reach servers both via the WiFi AP and wired connections, it is crucial to ensure they utilize the correct network path. The following commands configure client preferences for network interfaces:

```
sudo ip route add 172.16.11.0/24 via 172.16.13.1 metric 2
sudo ip route del 172.16.11.0/24 via 172.16.13.1 meric 2
```

Here, `172.16.13.1` is the IP of the WiFi AP. These commands alternate the traffic route between the wired and wireless interfaces on the clients.

2.2 Bottleneck configuration

The bottlenecks are placed at either Router 1, Router 2, or at the WiFi AP. This corresponds to the two users at Client 1 and Client 2 either having a common bottleneck or not having a common bottleneck. When the bottleneck occurs at Router 2, the user at Client 2 is not expected be significantly affected, as Router 1 should still have a surplus of bandwidth.

2.2.1 Configuration of throughput limitations

In order to configure the bottlenecks, we change the throughput at Router 1 and Router 2. The throughput at the WiFi AP is set to the lowest value it supports which is 54 Mbps. Depending on the desired bottleneck location, the throughput at Router 1 and Router 2 was adjusted either above or below this threshold, depending on where the bottleneck should be. Table 1 lists throughput values used in each bottleneck scenario.

Table 1: Throughput at the two routers in the different bottleneck configurations

Bottleneck at:	Router 1 (Mbps)	Router 2 (Mbps)
Router 1	15	70
Router 2	70	15
WiFi AP	70	70

2.2.2 TODO Additional variable adjustments

In addition to throughput, bot queue length and delay were varied on Router 1. The queue lengths were set to:

- 0.5 BDP
- 1 BDP
- 1.5 BDP
- 2 BDP

Delays of 50ms and 10ms were tested at Router 1, with Router 2 consistently set at a 10ms delay. The actual buffer-sizes used for the tests are listed in Table 2.

Table 2: Buffer sizes used (in number of packets) for the different delay and bdp configurations

	10ms delay	50ms delay
0.5 BDP	6	31
1 BDP	12	62
1.5 BDP	18	93
2 BDP	24	124

2.2.3 TEACUP configuration

In order to set these limitations, we used the built-in functions of TEACUP which allows us to set these limitations for the router, and then it runs the tests for each of the values specified. This is done by setting the following variables in the TEACUP configuration:

```
# Emulated delays in ms
TPCONF_delays = [25,]

# Emulated bandwidths (downstream, upstream)
TPCONF_bandwidths = [
    ('70mbit', '70mbit'),
]

# Buffer size
TPCONF_buffer_sizes = [31, 62, 93, 124]
```

This will run 4 different tests, as there are 4 different buffer sizes specified. Note that the delay here is set to 25ms, as TEACUP sets the delay for both downstream and upstream traffic.

While running the tests, TEACUP creates a matrix from these lists and executes the tests accordingly. However, this approach did not align with our objectives, as we intended to use distinct sets of buffer sizes for each delay configuration. Specifying multiple values for buffer sizes alone prevented an excessive number of test runs by limiting the variable combinations. We then change the other variables by using multiple configuration files and swapping them for each test-run.

These values are set for Router 1 using the configuration below. This sets the values for traffic passing through the network interface facing the switch.

```
TPCONF_router_queues = [
  # Set same delay for every host
    ('1', " source='0.0.0.0/0', dest='172.16.12.0/24', delay=V_delay, "
     " loss=V_loss, rate=V_up_rate, queue_disc=V_aqm, queue_size=V_bsize
        ↪  "),
    ('2', " source='172.16.12.0/24', dest='0.0.0.0/0', delay=V_delay, "
     " loss=V_loss, rate=V_down_rate, queue_disc=V_aqm, queue_size=
        ↪ V_bsize "),
]
```

TEACUP does not support setting different configurations for multiple routers. This meant that we had to limit the bandwidth for Router 2 differently.

2.2.4 Configuring Router 2 with `tc`

To configure the delay for Router 2, we utilized the `tc` program to define qdisc rules. TEACUP provides a variable `TPCONF_host_init_custom_cmds` that executes specified commands on a designated machine. We used the following configuration to automatically enforce the desired constraints on Router 2.

```
TPCONF_host_init_custom_cmds = {
   'pc02' : ['tc qdisc del dev enp13s1 root',
             'tc qdisc add dev enp13s1 root handle 2: netem delay %s' %
                ↪ tc_delay,
             'tc qdisc add dev enp13s1 parent 2: handle 3: htb default
                ↪ 10',
             'tc class add dev enp13s1 parent 3: classid 10 htb rate %s'
                ↪  % tc_rate,
             'tc qdisc add dev enp13s1 parent 3:10 handle 11: bfifo
                ↪ limit %s' % tc_bsize],
}
```

The `tc_rate` values were aligned with those specified in 1, and `tc_delay` was consistently set to `10ms`, in accordance with the delay used for all tests. The `tc_bsize` was configured to equal 1 BDP.

2.3 Traffic generation and types

For all tests, we use one VoIP traffic connection between Client 2 and Server 2, while we use a few different types of traffic between Server 1 and Client 1. We vary the type of TCP flows by changing the congestion control (CC) algorithm, and the type of traffic. These are:

- Triple flow using CC algorithm Reno
- Triple flow using CC algorithm BBR
- Triple flow using CC algorithm Cubic
- Mixed flow consisting of one each of Cubic, BBR, and Reno
- VoIP plus triple Reno
- VoIP plus triple Cubic
- VoIP plus triple BBR
- VoIP combined with one each of Cubic, BBR, and Reno

We emulate VoIP traffic, for which we send 20 UDP packets per second with a packet size of 100 bytes (this mimics Skype, which will use TCP when UDP does not work and was found to send at roughly this rate and packet size with occasional outliers [5]). This is done using iperf with the flags `-b 16k -l 100 -i 0.05` at the client. This sends packets of size 100B every 0.05 seconds, which adds up to 16Kb per second. These flags are described in [6].

The configuration for VoIP traffic was consistent with that used between Server 2 and Client 2. Different TCP congestion control algorithms were configured in iperf with the `-Z` flag. We also initially had some tests with just web or VoIP traffic, without any normal iperf traffic, but this did not generate sufficient traffic to cause any congestion.

The below source block details how the traffic generation was configured. It features three BBR streams between Server 1 and Client 1, and one VoIP stream between Server 2 and Client 2. `pc01` and `pc04` corresponds to Server 1 and Client 1 respectively, while `pc03` and `pc05` corresponds to Server 2 and Client 2 respectively.

```
traffic_iperf = [
    # pc01 -> pc04
    ('0.0', '1', " start_iperf, client='pc04', server='pc01', port=5001,
        ↪ "
     " duration=V_duration, extra_params_client='-R',
         ↪ extra_params_server='-Z bbr' "),
    ('0.0', '2', " start_iperf, client='pc04', server='pc01', port=5002,
        ↪ "
     " duration=V_duration, extra_params_client='-R',
         ↪ extra_params_server='-Z bbr' "),
    ('0.0', '3', " start_iperf, client='pc04', server='pc01', port=5003,
        ↪ "
     " duration=V_duration, extra_params_client='-R',
         ↪ extra_params_server='-Z bbr' "),

    # pc03 -> pc05
    ('0.0', '5', " start_iperf, client='pc05', server='pc03', port=5001,
        ↪ "
    " duration=V_duration, extra_params_client='-b 16k -l 100 -i 0.05 -
       ↪ R' "),
]
```

By default, iperf transmits data from the client to the server, but the '-R' flag reverses this direction. We utilize the '-R' flag instead of swapping server and client roles due to the home router functioning as the WiFi AP and employing NAT. This makes it essential to ensure that the clients are the ones to initiate the connection.

2.4 Collecting data

When running tests with TEACUP, it automatically collects pcap files using tcpdump, on the network interfaces that have an IP address. In addition, we set up NETHINT on Router 1 to listen to the WiFi traffic using the wireless network interface. As NETHINT writes the collected data to a file, we had to make a custom function in the TEACUP program to set up this as a logger.

```
@parallel
def start_nethint_logger(file_prefix='', remote_dir='', local_dir='.'):
    # log queue length and queue delay
    logfile = remote_dir + file_prefix + '_' + \
            env.host_string.replace(":", "_") + "_nethint.log"

    # For listening to wireless interface
    pid = runbg('/home/teacup/oskar/NETHINT/src/main.py --wireless -l %s
       ↪  6' % logfile)

    bgproc.register_proc_later(
           env.host_string,
           local_dir,
           'nethintlogger',
           '00',
           pid,
           logfile)
```

And then call in in the `start_loggers` function in `loggers.py` as:

```
execute(
    start_nethint_logger,
    file_prefix,
    remote_dir,
    local_dir,
    hosts=config.TPCONF_router)
```

NETHINT operates as a logger on Router 1. Since NETHINT is monitoring a wireless network card, it must be executed with the `--wireless` flag. Additionally, it is configured to use the 6th network interface, which corresponds to the wireless network interface.

3 Changes made to NETHINT

In order to calculate the correlation between the network streams, we needed to make a few small changes to the NETHINT program. We also found a miscalculation which had to be fixed. These changes were:

- Enabled logging of the raw OWD, rather than solely the variations in OWD.

- Corrected the OWD calculation.
- Modified it to save information about all packets, not just those with a valid RTT.

NETHINT produces JSON files, each with a JSON object representing the data for an individual packet. Within each object, there is information about the destination and source IP addresses and ports, alongside data pertaining to RTT and OWD among other details. Initially, the program was designed to save only the packets for which it could calculate an RTT. However, other metrics, such as OWD, may be valuable even when RTT data is unavailable. The way that NETHINT calculates the RTT only works for outgoing traffic, which meant that it did not log any packets belonging to incoming traffic. Consequently, we have made certain modifications to NETHINT that expand the range of data it logs. Details of these changes can be found in [7]. To change this, it was only one line which had to be changed:

```
- if isset_log() and valid_rtt:
+ if isset_log():
```

Additionally, there was a mistake in the calculation of the OWD. The OWD is calculated by subtracting the tsval in the timestamp field from the capture time. However, the capture time is given in seconds, while the tsval is in milliseconds, hence, the capture time had to be converted to milliseconds. This was also a rather easy change:

```
- owd = self.cap_time - self.tsval
+ owd = int(self.cap_time * 1000) - self.tsval
```

The value given in tsval is not usually the actual time, but rather a value given by a “timestamp clock”, nevertheless it should be approximately proportional to real time [8]. This means that the calculated OWD is not really the actual time it takes for the packets to travel one way, but that it still is a number which says something about how the OWD changes. This is fine for our purpose as we are considering how the OWD changes with time, and not the actual values. Though we still need to have the actual OWD values for

The value in tsval is typically not the actual time but a value from a “timestamp clock,” which should be approximately proportional to real time [8]. Therefore, while the calculated OWD may not reflect the precise time it takes for packets to traverse one way, it is still a number indicative of relative changes in OWD. For our purposes, this is sufficient, however, we still the raw OWD values for our analysis. NETHINT was originally programmed to record the variations in OWD, necessitating a modification to enable the logging of raw OWD values.

4 Other limitations experienced with NETHINT

- Not able to capture all data
- High RAM usage
- Long processing time of the pcap files (longer that it took to collect them)

5 Data

In order to compare how the OWD changes between the different bottleneck location configurations, we first calculate the correlation coefficient between the two traffic paths (Server 1 to Client 1 and Server 2 to Client 2). For this we collect the data from the JSON objects by comparing the destination address. This means that we don't differentiate between the different TCP streams between Server 1 and Client 1, but that we look at all data going towards Client 1 as a whole. This gives us two datasets, one for traffic towards Client 1 and one of traffic towards Client 2. The x here are the capture times, while the y values are the OWD values.

When calculating the correlation coefficient between these two datasets, we get a number representing how much they move together. That is, when one dataset increases, how much does the other increase. In order to calculate this, it is necessary that both datasets are of equal length, and have corresponding x values. This is almost guaranteed not to be the case. To solve this, we interpolate the smaller dataset using numpy, by setting:

```
y2 = np.interp(x1, x2, y2)
```

where `x2` and `y2` correspond to the x and y values of the smaller dataset, while `x1` is the x values of the larger dataset.

After interpolating the dataset, we calculate the correlation coefficient using `scipy.stats`:

```
corr, _ = stats.pearsonr(y1, y2)
```

We calculate this correlation coefficient for each test. Then we calculate the CDF for the correlations for each bottleneck configuration. This is done using numpy:

```
x = np.sort(y_values)
y = np.arange(1, len(x) + 1) / len(x)
```

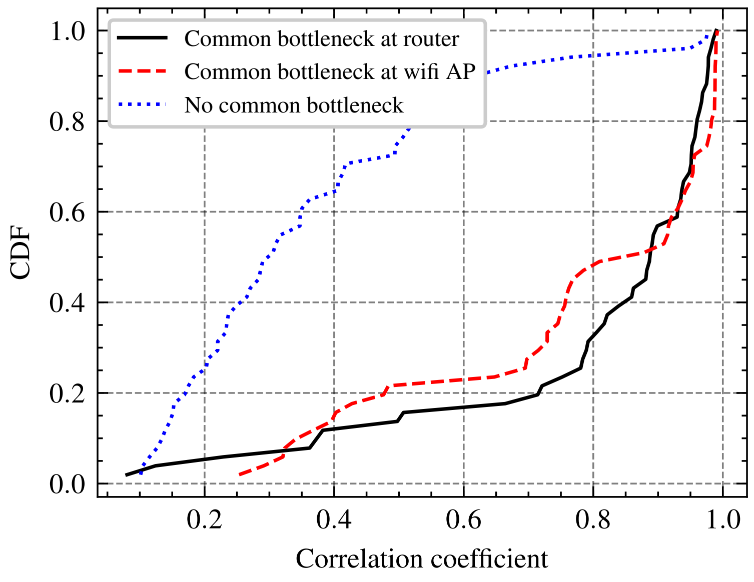


Figure 2: CDF graph for all test cases

Figure 2 shows the CDF for the correlation coefficient for all the tests. It shows that there is a clear difference in how the OWD changes over time between when the two traffic routes have a common bottleneck and when they do not. It can also be seen that if the correlation coefficient is greater than 0.7, it is very likely that there is a common bottleneck. However, there are still some tests, where also the configuration with no common bottleneck have a very high correlation coefficient, so this value being high does not guarantee that there is a common bottleneck.

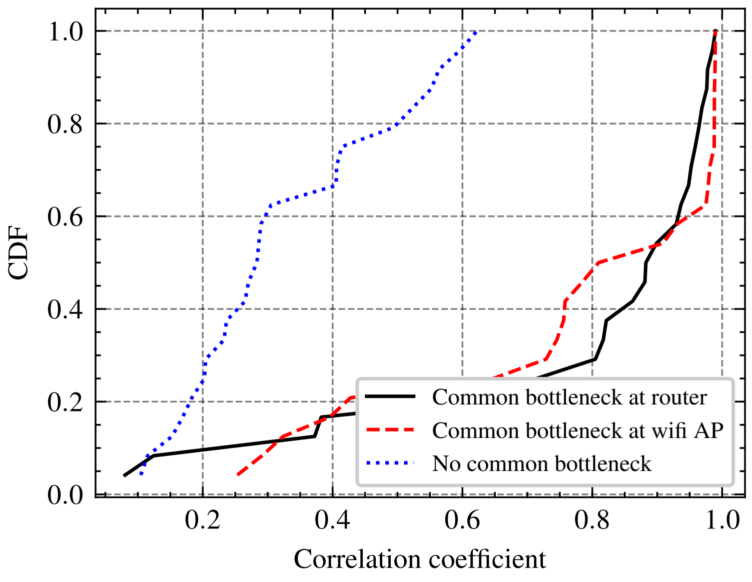


Figure 3: CDF graph for all tests without VoIP traffic

An interesting aspect is that all tests where the correlation coefficient for no common bottleneck in Figure 2 was above 0.7 also had VoIP traffic between Server 1 and Client 1. When excluding all tests that also had VoIP traffic between these, the highest correlation coefficient for no common bottleneck was 0.623. The entire CDF plot for this is shown in Figure 3 where it is a clear gap between the highest correlation values for when there is no common bottleneck and when there is.

5.1 Only VoIP traffic

A possible explanation for why having VoIP traffic between Server 1 and Client 1 causes such high correlation could be that having identical traffic patterns directed towards both clients at simultaneously can make it seem more correlated than it should be, despite the presence of other concurrent traffic. We also ran some tests with just VoIP traffic between Server 1 and Client 1, meaning that the traffic going to the two Clients should be now identical. When only running VoIP traffic like this, we did not generate enough traffic to reach any of the configured throughputs, meaning that essentially all tests like this was without any common bottleneck. For this reason, we have combined all the tests into just one CDF plot shown in Figure 4, where it shows that this traffic tends to seem either very correlated or not very correlated at all, which confirms that the type of traffic being similar can be enough to get very high correlation values.

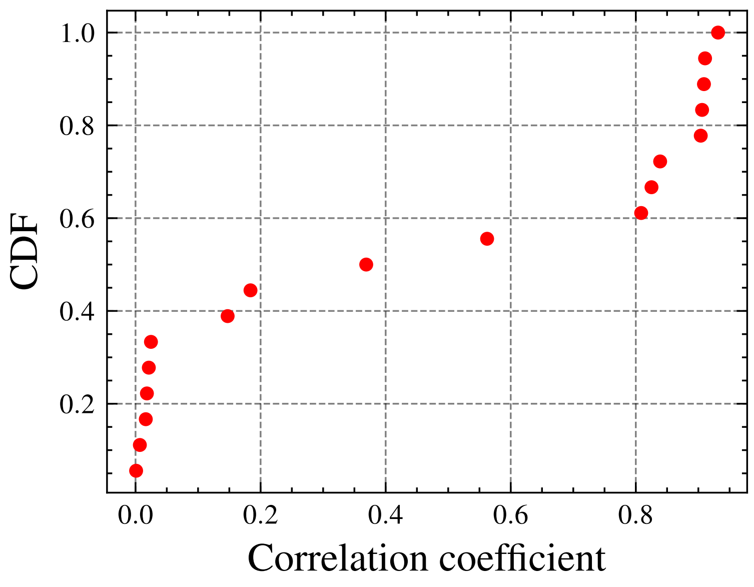


Figure 4: CDF graph for traffic only containing of VoIP traffic

5.2 Double CDF

Another way to compare the traffic is to first calculate the CDF for the raw OWDs for traffic directed towards the two Clients. This allows for comparing the distribution of OWD values instead of how the OWD values changes over time. Since the OWD values are not actual real clock time, they will vary significantly in actual values. It is therefore necessary to align the OWD values. We have done this by moving all OWD values such that the lowest OWD value for each client becomes 0. This means that all other OWD value essentially represent how much longer the OWD is compared to the lowest OWD, and that the calculated correlation represent how similar this distribution is between the two clients. As can be seen in Figure 5 this did not actually affect the CDF for no common bottleneck, except for decreasing the highest CDF values slightly compared to in Figure 2. However, the values for both configurations with a common bottleneck now appear significantly more correlated.

References

[1] P. J. Barhaugen, “NETwork Home INTerference.” May 08, 2023. Accessed: Nov. 30, 2023. [Online]. Available: `https://github.com/petternett/NETHINT`

[2] S. Zander and G. Armitage, “CAIA Testbed for TEACUP Experiments Version 2,” 2015.

[3] Oskar, “Ohaukeboe/teacup-nethint.” Dec. 10, 2023. Accessed: Dec. 10, 2023. [Online]. Available: `https://github.com/ohaukeboe/teacup-nethint`

[4] P. Juterud Barhaugen, “Home Network Interference Detection with Passive Measurements,” University of Oslo, 2023.

[5] M. Mazhar Rathore, A. Ahmad, A. Paul, and S. Rho, “Exploiting encrypted and tunneled multimedia calls in high-speed big data environment,” *Multimed tools appl*, vol. 77, no. 4, pp. 4959–4984, Feb. 2018, doi: 10.1007/s11042-017-4393-7.

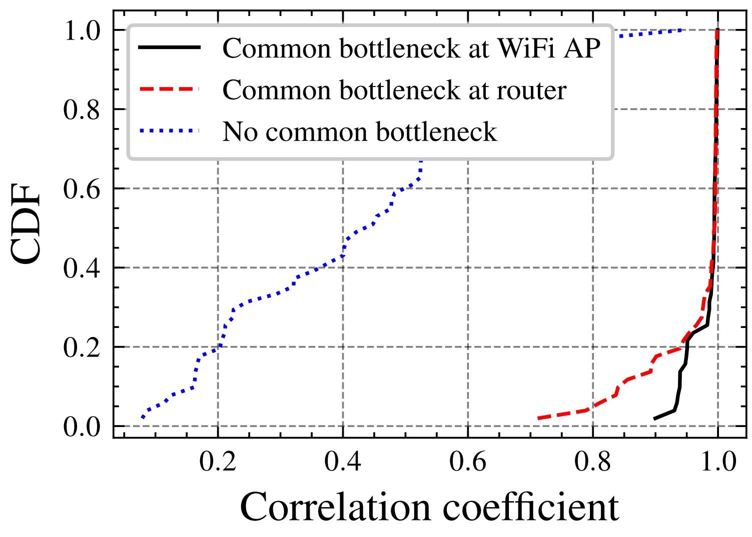


Figure 5: CDF of correlation coefficients of CDF

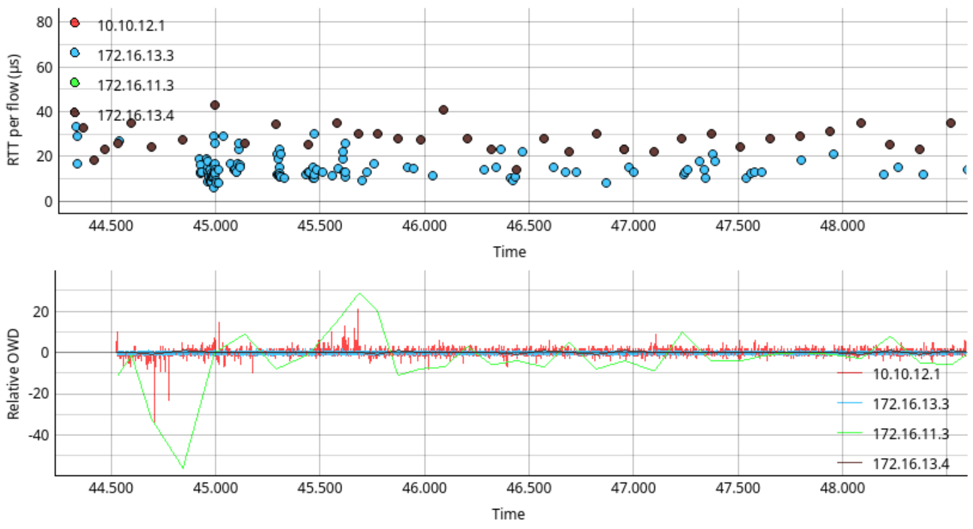
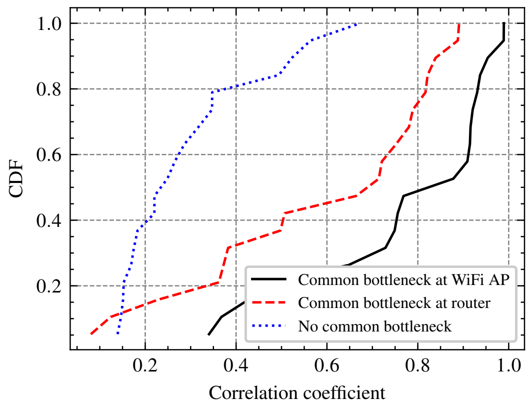
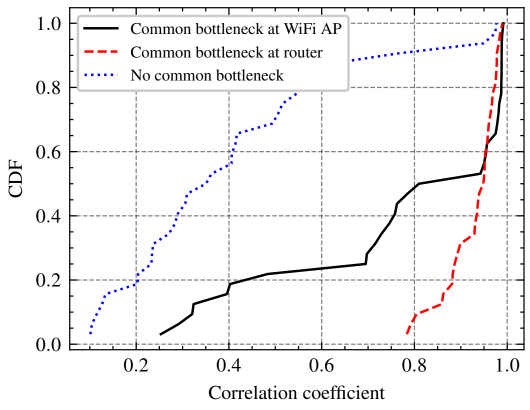


Figure 6: Image of NETHINT

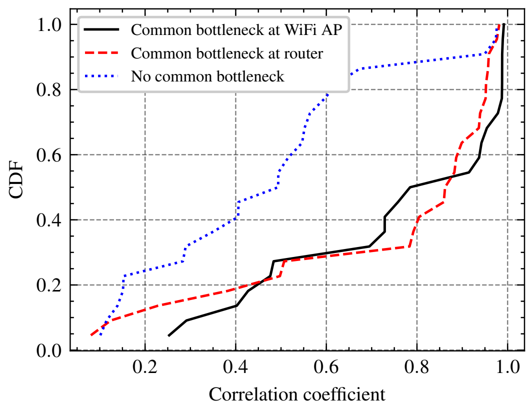


(a) Delay of 10ms

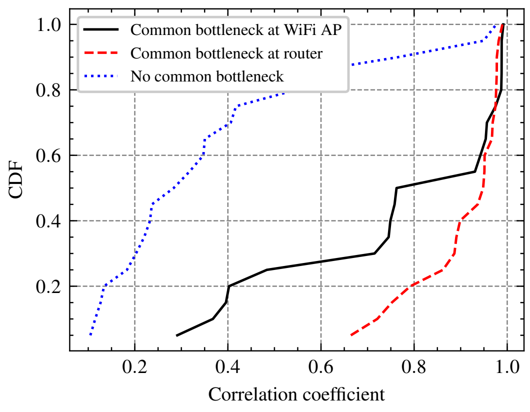


(b) Delay of 50ms

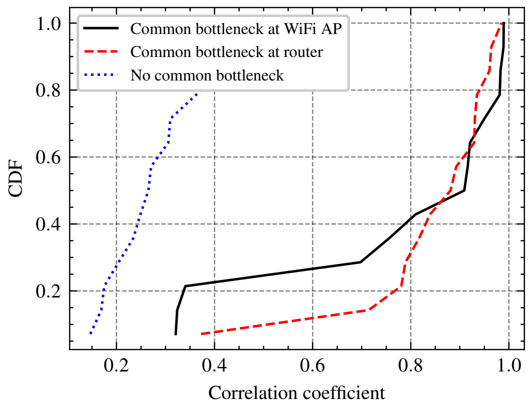
Figure 7: CDF plots for each delay configuration



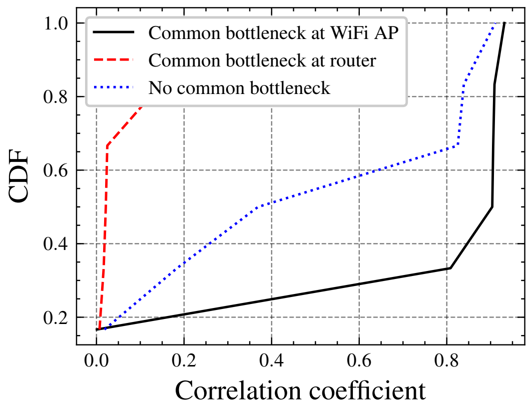
(a) 0.5 BDP



(b) 1 BDP



(c) 1.5 BDP



(d) 2 BDP

Figure 8: CDF plots for each BDP configuration

[6] “Manpage of IPERF.” Accessed: Dec. 07, 2023. [Online]. Available: `https://iperf2.sourceforge.io/iperf-manpage.html`

[7] Oskar, “Ohaukeboe/NETHINT.” Dec. 08, 2023. Accessed: Dec. 11, 2023. [Online]. Available: `https://github.com/ohaukeboe/NETHINT`

[8] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger, “TCP Extensions for High Performance,” Internet Engineering Task Force, Request for Comments RFC 7323, Sep. 2014. doi: 10.17487/RFC7323.