Detection of shared network bottleneck using passive WiFi traffic analysis

IN3260

Oskar Haukebøe

December 13, 2023

1 Introduction

Network performance issues are a widespread 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 [2] testbed facilitates a systematic approach to simulating various network scenarios with different points of congestion. Through automated experiments on real machines, this assignment aims to critically evaluate NETHINT's proficiency in 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 configure 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 of network traffic, as well as throughput and delay limitations at the routers. Additionally, it enables automatic logging of network traffic.

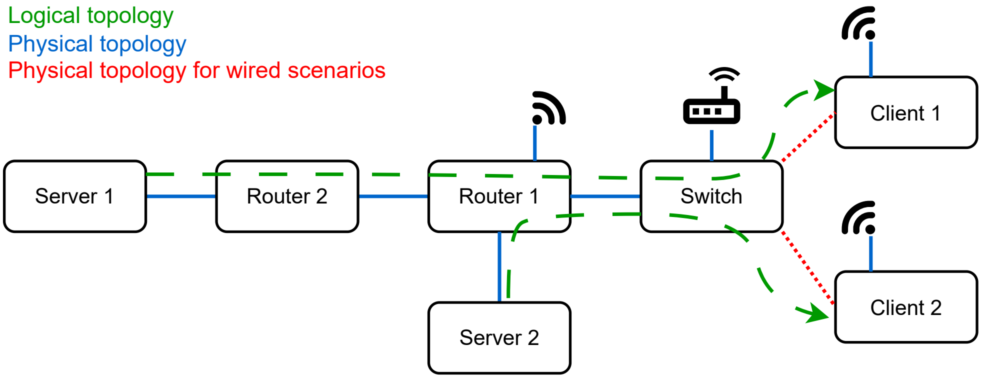


Figure 1: Topology of the testbed

2.1 Network Devices and Configuration

Figure 1 shows how all the devices are connected, and how the traffic flows between them. The logical topology shows how the traffic between Server 1 and Client 1 passes both routers, while the traffic between Server 2 and Client 2 only passes one router. This allows us to position the bottleneck at either Router 1 or Router 2, which gives either a shared bottleneck or no shared bottleneck between the two logical topologies. The roles of the individual nodes are the following:

- *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. Network congestion is generated at the routers.
- *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.
- *WiFi interface at Router 1:* Router 1's WiFi card, set to monitor mode for traffic sniffing, also configured to channel 3 to monitor the WiFi AP.

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

2.1.1 Address configuration

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 also 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 over WiFi. Router 1's WiFi card, configured in monitor mode, is used 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 via the 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, allowing us to easily switch between running the tests over WiFi or over Ethernet.

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. 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 capacity 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 capacity at Router 1 and Router 2 was adjusted either above or below this threshold. Table 1 lists capacity values used in each bottleneck scenario.

Table 1: Configured capacity 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 Additional variable adjustments

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

- 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. These values were calculated using 15Mbps as the lowest capacity. However, an oversight led us to not update these values for when the WiFi AP was the bottleneck where the lowest throughput was 45Mbps. This caused the BDP values used for this traffic to be around $\frac{1}{3}$ of what it was supposed to be.

Table 2: Buffer sizes used on Router 1 (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 one of the routers, 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 to create an RTT of 50ms, as the delay is set for both downstream and upstream traffic.

While running the tests, TEACUP creates a matrix from these lists and executes the tests accordingly, meaning that we could also specify both delays used and have it run the tests for all combinations between buffer sizes and delays. 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 traffic queue configurations for the 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 between Client 2 and Server 2, while we use a few different types of traffic between Server 1 and Client 1. For NETHINT to be able to collect information about the packet streams, the traffic must use TCP. We vary the type of TCP flows by changing the congestion control (CC) algorithm, and the type of traffic. The following are the traffic configurations used between Server 1 and Client 1:

- Three flows using CC algorithm Reno
- Three flows using CC algorithm BBR
- Three flows using CC algorithm Cubic
- Mixed traffic consisting of one flow each of Cubic, BBR, and Reno
- VoIP plus 3x Reno
- VoIP plus 3x Cubic
- VoIP plus 3x BBR
- VoIP combined with one flow each of Cubic, BBR, and Reno

These are 8 different types of traffic that are run on two different delay configurations as well as 4 different BDP configurations. This makes 64 different tests that are run for each of the three bottleneck configurations, giving a total of 192 test runs. Additionally, each test ran for 300 seconds as this produced 20 sawteeth when using TCP Reno, which is the slowest, with the largest queue and BDP values.

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], and exact way this is set is shown in the configuration below.

The configuration for VoIP traffic between Client 1 and Server 1 was consistent with that used between Server 2 and Client 2. Different TCP congestion control algorithms were configured using the `-Z` flag in iperf. 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 configuration 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` correspond to Server 1 and Client 1 respectively, while `pc03` and `pc05` correspond 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 having NAT enabled. This makes it essential to ensure that the clients are the ones to initiate the connection as the server machines cannot reach the client IP addresses.

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 this up 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.

When running NETHINT, it is possible to make it render a GUI which shows the data as in Figure 2. But for our analysis, it is easier to have all this data collected in a file instead. NETHINT supports logging all data into a JSON file by running it with the flag `-l`, which we then can use for our analysis of the data.

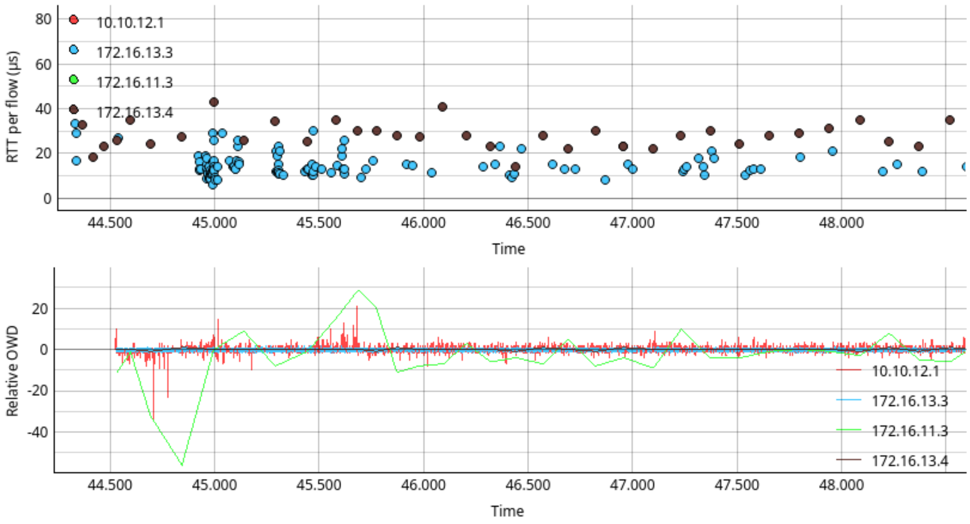


Figure 2: Screenshot of NETHINT

Unfortunately, we later found that this yielded very few packets to be collected by NETHINT. It would therefore likely yield better results to capture a pcap file for the traffic on Router 1s WiFi card, and then analyze this data with NETHINT later. This could be done either by using tcpdump, or by running NETHINT with the `-w` flag which enable writing to a pcap file. As we did not get the opportunity to run these tests again with this change, we use the pcap files generated by TEACUP on the network interfaces of both Client 1 and Client 2. This should produce similar data except that we cannot guarantee that the real time clocks of the two clients are perfectly synchronized.

It was also found that out of all 192 test runs, 13 of them seem to have failed as no packets were sent between Server 2 and Client 2 apart from SYN and ACK. Upon inspecting the pcap files generated at Client 2, it appears that a TCP RST packet got sent, stopping the TCP transmission. It is unclear why these tests failed, but when using the data in the next sections, we will be excluding these tests. All failed tests had configured delay of 10ms and were the following:

- 0.5 BDP and VoIP plus one flow each of BBR, Cubic and Reno with bottleneck at WiFi AP
- 0.5 BDP and three Cubic with bottleneck at WiFi AP
- 1 BDP and one flow each of BBR, Cubic and Reno with no common bottleneck
- 1 BDP and VoIP plus three BBR with no common bottleneck
- 1 BDP and three BBR with no common bottleneck
- 1 BDP and three Reno with no common bottleneck
- 1.5 BDP and VoIP plus flow each of BBR, Cubic and Reno with no common bottleneck
- 1.5 BDP and three Cubic with no common bottleneck

- 2 BDP and one flow each of BBR, Cubic and Reno with no common bottleneck
- 2 BDP and VoIP plus three BBR with no common bottleneck
- 2 BDP and VoIP plus three Cubic with no common bottleneck
- 2 BDP and three BBR with no common bottleneck
- 2 BDP and three Reno with no common bottleneck

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 the program 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 still 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. These changes are available at [7]. To achieve 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, which is given by scapy, 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 that can be used to say 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. We still need to have the actual OWD values for our calculations. NETHINT originally only logged how the OWD changed (Current OWD - Previous OWD), however, we need the actual raw OWD values. This meant that we also had to enable logging of the raw OWD values.

4 Results

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, allowing us to compare how the OWD changes over time between the two logical topologies in Figure 1.

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` are 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 3 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, showing that this value being high does not guarantee that there is a common bottleneck.

An interesting aspect is that all tests where the correlation coefficient for no common bottleneck in Figure 3 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 the "no common bottleneck" case was 0.62. The entire CDF plot for this is shown in Figure 4 where we can see a clear gap between the highest correlation values for when there is no common bottleneck and when there is.

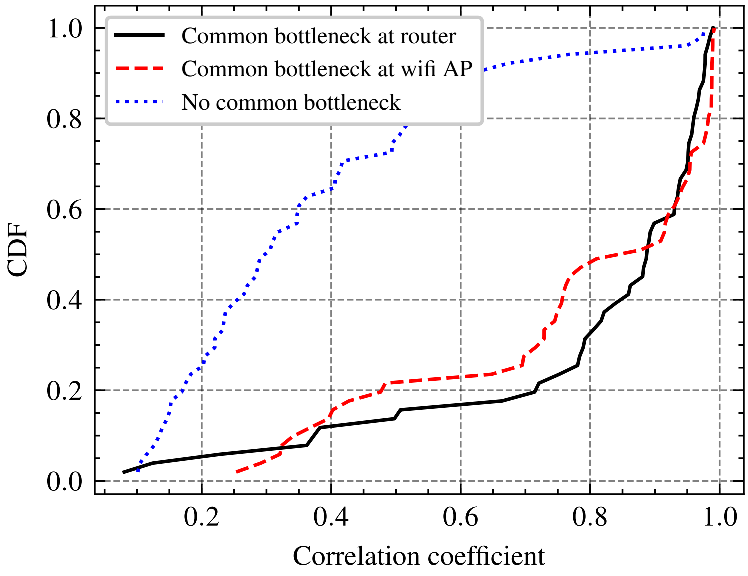


Figure 3: CDF graph for all tests

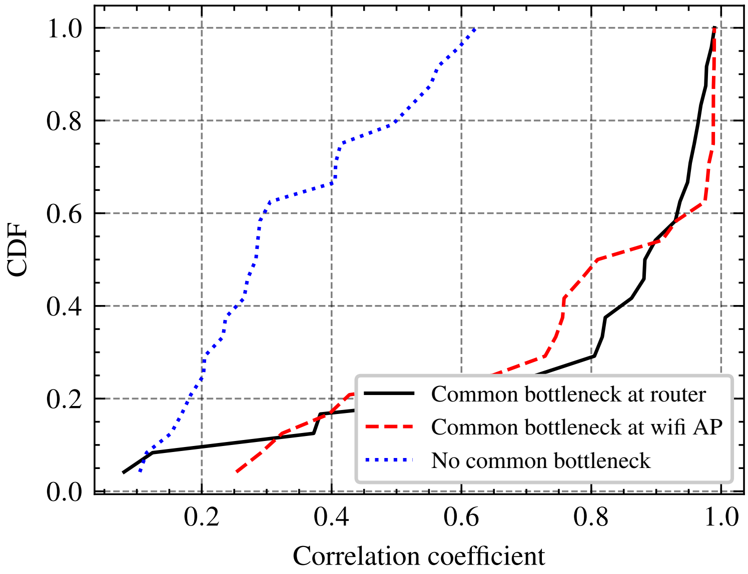


Figure 4: CDF graph for all tests without VoIP traffic

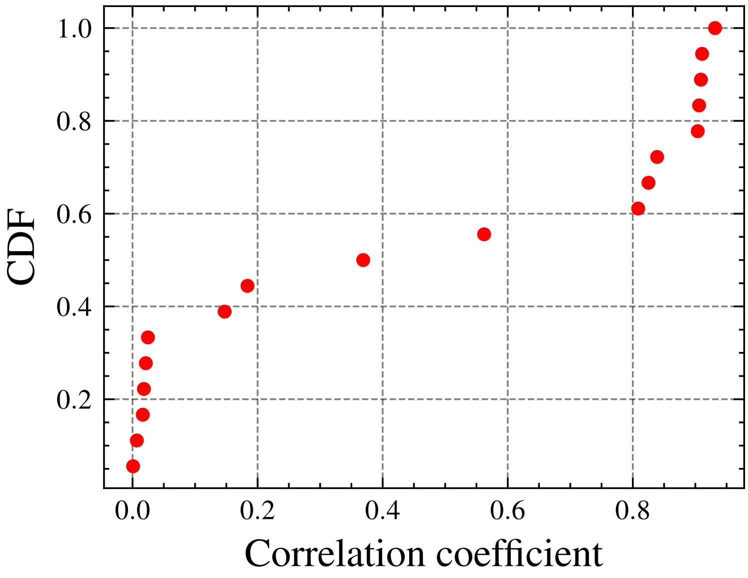


Figure 5: CDF graph for traffic only containing of VoIP traffic

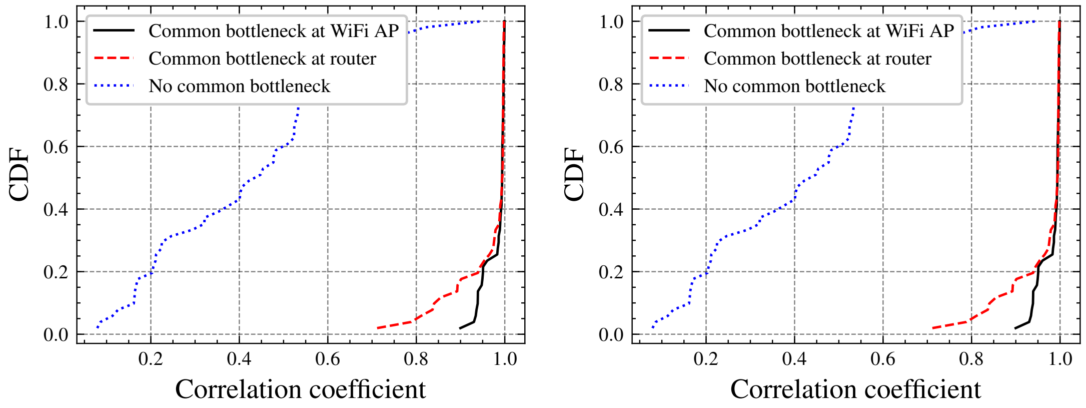
4.1 Only VoIP traffic

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 throughput bottlenecks, meaning that essentially all tests like this were without any common bottleneck. For this reason, we have combined all the tests into just one CDF plot shown in Figure 5, which 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.

The fact that the VoIP tests did not produce enough traffic to cause congestion makes Figure 3 potentially misleading - a practical system might put a lower bound limit on the used delay values, as correlations in our system are meant to reflect queue interference, not traffic patterns.

4.2 Correlation of 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 shifting 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



(a) All data

(b) No tests with VoIP

Figure 6: CDF of correlation coefficients of CDF

distribution is between the two clients. In a practical system, this would be similar to determining the correlation after carrying out long-term measurements covering various traffic patterns.

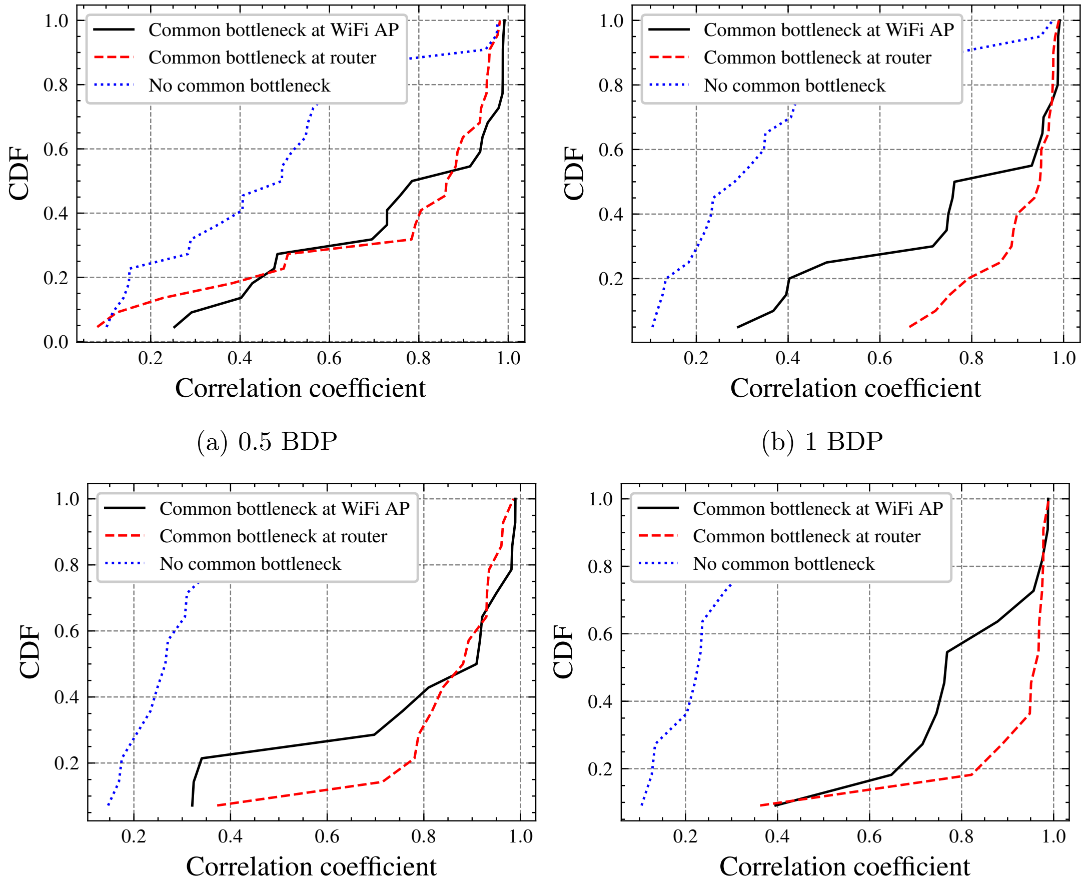
As can be seen in Figure 6 this did not significantly affect the CDF for no common bottleneck, except for decreasing the highest CDF values slightly compared to in Figure 3. However, the values for both configurations with a common bottleneck now appear significantly more correlated. With this calculation, the highest correlation coefficient for no common bottleneck is less than 0.95, while this same value only accounts for about 20% of the tests with a common bottleneck.

When doing the calculation in this way, the difference between including the tests with VoIP traffic, and not including it, was not very significant. This can be seen as the two graphs 6a and 6b are very similar.

4.3 Comparison between BDP

Figure 7 shows the CDF for each of the BDP configurations separately, rather than the complete data over all configurations that was used for the previous diagrams. This also reflects a more realistic case, as one would typically apply a monitoring tool in one particular network configuration only.

These were calculated as in Section 4. It can be seen that the correlations for no common bottleneck tend to decrease as the BDP increases. It is also noticeable how the correlations for when there is a common bottleneck at the router tend to increase as the BDP increases. We don't see the same effect for when the bottleneck is at the WiFi AP, but this is likely due to all the BDP configurations here actually being much smaller as mentioned in Section 2.2.2.



(a) 0.5 BDP

(b) 1 BDP

(c) 1.5 BDP

(d) 2 BDP

Figure 7: CDF plots for each BDP configuration

5 Conclusion and future work

In this assignment, we have demonstrated how NETHINT can be used for determining whether a network bottleneck is situated in the local WiFi or somewhere else by evaluating the OWD of the incoming packets. By calculating the correlation of the OWD of the incoming packets between two clients, we have seen a significant trend in there being a higher correlation when the bottleneck is located in a shared router, rather that when the bottleneck is situated somewhere only affecting the client using the most traffic.

The results were very promising both when correlating how the OWD changes over time, as well as when comparing the distribution of OWD values. But they also showed how the calculated correlation can be high due to similarities in the traffic pattern. This effect of having similar traffic pattern is something that can be further studied in order to better understand its effect on the perceived correlation and how to circumvent it.

In order to properly determine whether NETHINT is suitable for detecting network congestion in a home network, it would also be necessary to run tests with more than two clients as there are likely more devices on a home network.

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.

[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.