WRAP

(Wireless RTT Analysis & Performance) Tool

Introduction

This tool was developed as part of the DISCO project's effort to evaluate Wi-Fi testbed performance through passive monitoring. The core challenge addressed is how to assess Wi-Fi performance improvements from machine learning models without actively interfering with the network or biasing results. This tool enables passive observation and quantitative measurement of user experience quality.

This tool analyzes network capture files (PCAP) to get an overall sense of a network's quality. PCAP is a standardized file-format for network packet captures that one can get through tools like tcpdump, wireshark, airodump, etc..

What's special about this tool is that it works with 802.11-frames at the Link-layer of the OSI-model. Capturing frames on this layer is not trivial, it requires special hardware and software in order to capture, unlike captures at the Transport-layer which tools like tcpdump can do on most units with a network interface.

The project is part of the IN3260 - Project Assignment in Informatics: Computer Networking, led by supervisors Magnus Skjegstad and Michael Welzl.

Related Work

This work is inspired by Petter Juterud Barhaugen's master thesis. He developed NETHINT, which is a suite of tools used to get an overall sense of a network's status, discover bottlenecks and high-consuming network units by monitoring TCP.

The suite of the master-thesis is written in python relying primarily on the packet manipulation library *scapy*, used for retrieving the information from a capture-file, *numpy* to analyze those numbers, *pyqt* and *pyqtgraph* for UI and data-representation. I have instead used *pyshark*, *matplotlib* and *pandas*.

The written master thesis gave me a great start for what tools I should look into for this task, ideas for what approaches I could take for calculating round trip times, and other useful insights.

Other similar tools include *Synthetic Packet Pairs (SPP)*, *Passive Ping (PPING)* and *eBPF* which enables round trip time measurements by monitoring TCP in the kernel.

Determining the 'Health' of the network

The most important factors for determining the status of a network provided by this tool is the round trip times and retransmissions. It can be useful to run the Stats tool first on the csv file in order to spot any outlier pairs.

Data Retransmissions

Data retransmissions are simply showing the amount of retried attempts of sending a single data frame. Multiple data retransmissions are a sign of bad network communication between two units.

ACK Retransmissions

Similar to data retransmissions, ACK retransmissions are the amount of retried attempts of sending an ACK for a specific data frame. Multiple ACK retransmissions are a sign of bad network communication between two units.

Round Trip Time (RTT)

The round trip time is a time measurement measured by ‘starting the timer’ on the first timestamp of the first Data-Frame of a sequence-number sent, and ‘stopping the timer’ when the first corresponding ACK for that Frame is recorded. A shorter RTT is indicates good performant communication.

The RTT is probably the most interesting indicator of a Network’s health, as communication performance is all about speed. However, note that this RTT-calculation might not be entirely accurate - which brings me to the next part of this report, this tool's weaknesses and possible anomalies.

Weaknesses / Irregularities / Anomalies

Missing Packets in capture

The capture-file isn’t always perfect, as the monitor might not capture every single frame during recording. This means that there might have been Data-Frames and ACK’s that did in fact get sent and / or even received that the monitor simply didn’t pick up.

First of all, a lost frame in the capture will give us a wrong / inaccurate retransmission count, regardless of Data-Frame og ACK.

In the case of a Data-Frame, we can actually verify this by checking the Data-Frame's retry-flag. If the first Data-Frame we encounter, meaning the first time seeing a Data-Frame with a given source, destination and sequence-number, has the retry-flag set to False, we can know that this is in fact the first sent Data-Frame of this sequence number and not a retransmission.
If however, the retry-flag is set to True, but it's the first time we've encountered a Data-Frame with this source, destination and sequence-number, we can be certain that an earlier Data-Frame has been sent and that we simply haven't caught it in the capture. This is what the "First Recorded Frame is Retransmission (bool-flag)" annotates in the csv file.

In the case of an ACK-frame we might get the wrong RTT-calculation, as there might have been an earlier ACK that actually made it to the receiver. Even worse, if the sender of the ACK never resends that ACK, we will get an 'infinite' RTT, crossing the timer-threshold, even though the ACK actually made it through.

An ACK might not actually correspond to the assumed sender

As described earlier, an ACK's source is not stated in the frame itself. The source-address is looked up through an algorithm that maintains a fixed-size history of the last 128 sequence numbers per sender, using a deque data structure. This allows accurate matching of ACKs with their data frames even when nodes communicate with multiple partners simultaneously, and is particularly important for handling BlockACKs which can acknowledge up to 64 previous frames at once.

Recorded ACK might not actually be received

The RTT is calculated by first_ack_timestamp - first_data_frame_timestamp. However, we have no way of validating that the ACK-frame actually made it through to the receiver, meaning that the RTT might be invalid. This is very likely the case where there are many ACK-retransmissions.

Capturing Data

Turris Omnia

The Turris Omnia is an open-source network router, which is in many ways in the heart of this project. It has the unique ability to sniff 802.11-frames from the MAC-layer of the OSI-model, as mentioned earlier, by setting up the router in promiscuous-, / 'monitor'-mode. This piece of hardware has been essential to get capture-files to analyze and experiment in order to develop this software-tool.

As the pcap-file with 802.11-frames is central to this project, and not trivial to get, I will now describe the process of capturing a pcap-file at the Link-layer in detail, using the Turris Omnia.

1. SSH into the Turris Omnia

   ```
   ssh turris01
   ```
2. Check available physical interfaces

   ```
   iw list
   ```
3. Setup a monitor interface

   ```
   iw phy0 interface add mon0 type monitor
   ```

   ```
   ip link set mon0 up
   ```
4. Take a network-dump

   ```
   tcpdump -i mon0 -w capture.pcap
   ```

Tools used, and tools ditched

Airodump - ditched

The Turris Omnia uses the OpenWRT Operating System, which is a Linux OS targeting embedded systems. The OpenWRT OS supports a bunch of Linux packages, and some OpenWRT specific packages. Airodump is one of these specific packages, and is described as follows;

"Airodump-ng is used for packet capture, capturing raw 802.11 frames. It is particularly suitable for collecting WEP IVs (Initialization Vector) or WPA handshakes for the intent of using them with aircrack-ng. If you have a GPS receiver connected to the computer, airodump-ng is capable of logging the coordinates of the found access points.

Additionally, airodump-ng writes out several files containing the details of all access points and clients seen, which can be used for scripting, or creating custom tools"

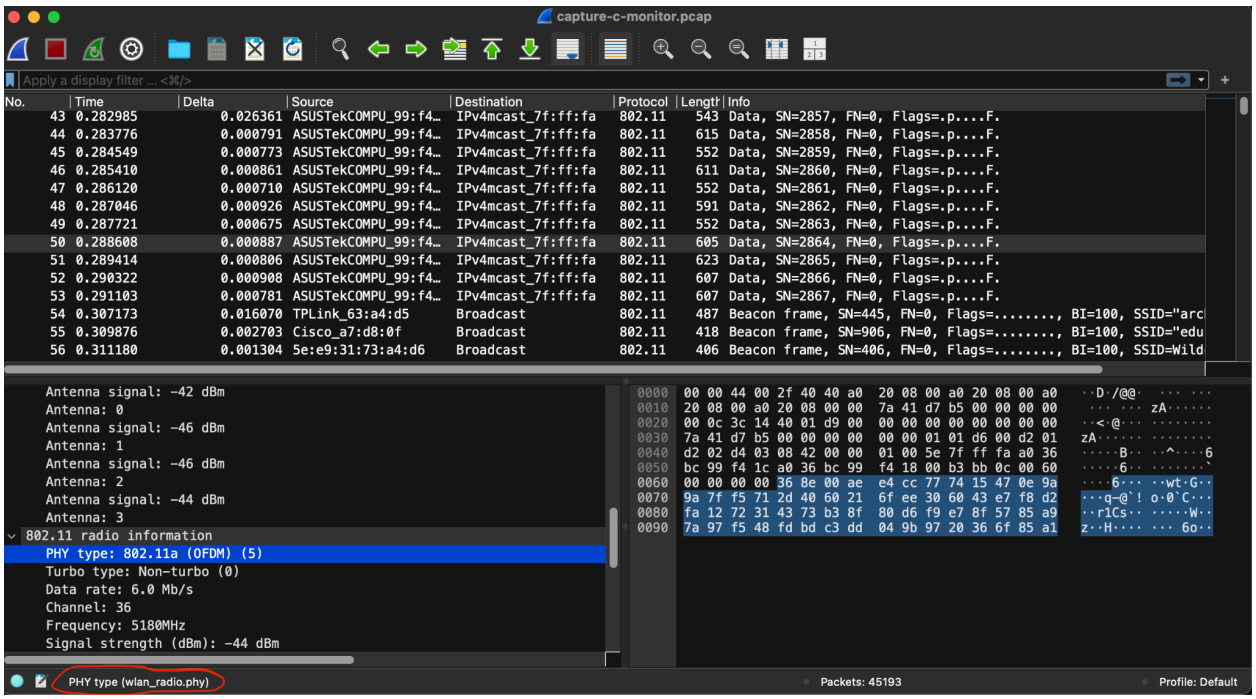
Airodump seemed like the perfect choice to begin with, but I actually ended up using tcpdump, as tcpdump is a more established tool and Airodump seemed to be more suited for 'hacking'-purposes, which is not what we're interested in here.

Scapy - ditched

As I briefly mentioned, I started out using the scapy-library to retrieve information from the pcap-file, as in the NETHINT master-thesis. However, as I found the GUI of Wireshark to be so useful for researching, developing and troubleshooting, I decided to use the pyshark-library, which is a python library based on tshark, giving me access to all the fields available in Wireshark.

Wireshark

Wireshark is a large tool with a GUI making it easy to analyze pcap-files. It has extensive tooling and filters in order to extract wanted information, and export information in different formats. One particularly useful trait of the software is it showing what filter-command a selected field corresponds to, making it easy to filter on a specified field. This same filter / field can be applied to the pyshark-library API.



As indicated at the bottom left marked with red, by selecting a specific row of information, one can get the corresponding tshark syntax for filtering on that property, either in wireshark as a filter, or with the pyshark library as shown below;

```
def parse_phy_type(pkt):
    phy_num_tot = safe_get_field(pkt, 'wlan_radio', 'phy')

    if not phy_num_tot:
        return "Unknown PHY type", -1
```

TCPDump

TCPDump is an even more established tool than Wireshark. For this workflow, I've only used tcpdump to get packet-captures from the Turris Omnia. By addressing the monitor interface on the Turris Omnia, we're able to capture 802.11-frames.

PyShark

PyShark is a python wrapper for tshark, making it very convenient to fetch information in the python script corresponding to the Wireshark interface.

Pandas

Pandas is a data analysis python library with C-optimized code making crunching numbers from the csv file in the plotter.py and stats.py scripts very efficient.

The Tech-Stack

The WRAP-tool consists of three parts;

- PCAP-to-CSV
- Plotter
- Stats

PCAP-to-CSV

The PCAP-to-CSV script is the meat and potatoes of this tool. It takes the pcap-capture file with recorded 802.11-frames and converts it into an easily human-readable csv file. Each line is supposed to represent one attempted packet transmission, including retransmissions and corresponding ack and ack-retransmissions. The headers for the csv file are:

- Source MAC-address
- Destination MAC-address
- Sequence Number
- Capture Length
- PHY Network Standard
- Frame-Type
- Bit-Rate
- Channel
- Frequency
- Timestamp
- Delta Time
- Amount of Data Frames
- Amount of ACK Frames
- Data Retransmissions
- ACK Retransmissions
- Round Trip Time
- First Recorded Frame is Retransmission (bool-flag)

The script is using a packet's source mac-address, destination mac-address and sequence-number to make a 'key' to aggregate on. The Data-Frame types hold all of this information in the frame, making it pretty straight forward to store the relevant information to that key.

The ACK-frames are not as straight-forward, as they only contain the destination MAC-address and not their source in order to stay lean. This is unlike TCP-layer frames, which include this information. To handle this, the tool maintains a rolling history of the last 128 sequence numbers for each sender, enabling accurate matching

of ACKs to their corresponding data frames even when nodes are communicating with multiple partners or using BlockACKs.

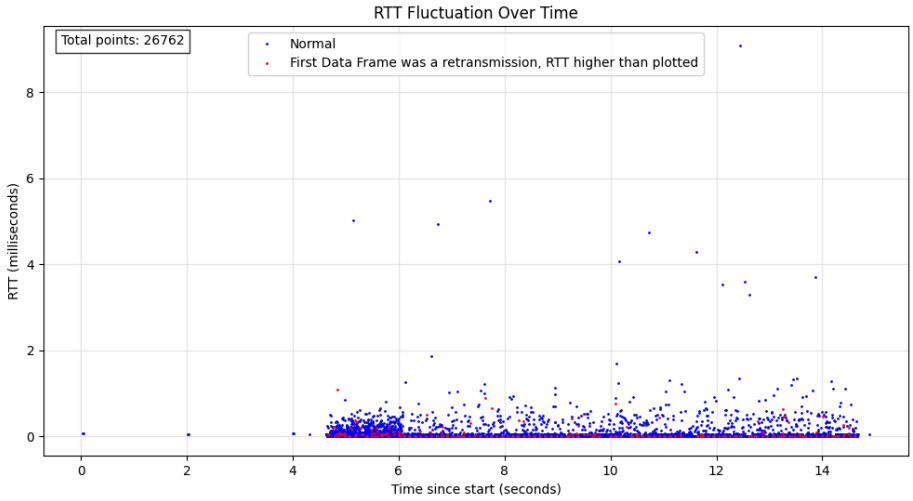
Another challenge is that the sequence-numbers wrap around after 4096. If not handled correctly, the wraparound would overwrite / layer upon an earlier entry before the entry. This is solved by simply checking if the last sequence number by a given source / destination is the same as the current frame. If the sequence-number is the same, it's either a first-time ACK, or a retransmission of either kind. If the sequence number is different from last time, we can move the last analyzed aggregated frame to a “completed frames” list, so that next time we see a sequence number, that frame will be treated as a separate line in the csv file.

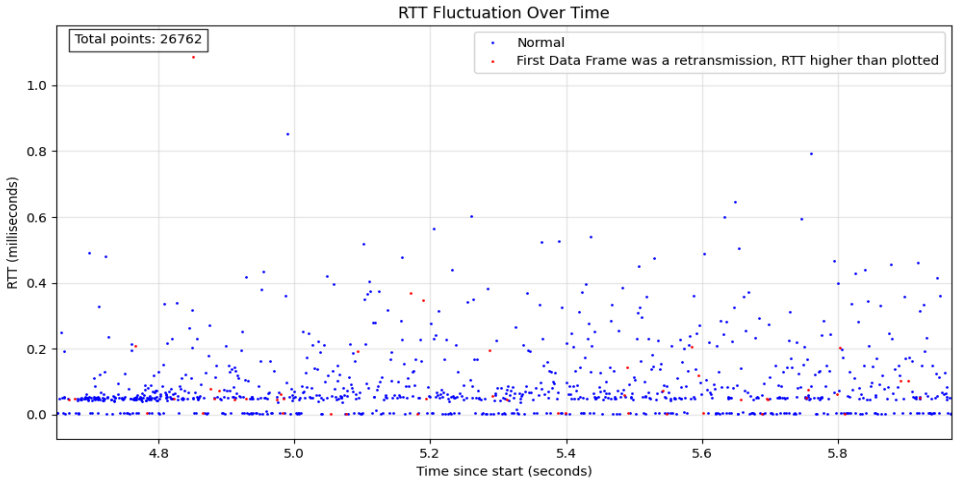
ACK’s also come in the form of BlockACK’s, which is a bundled frame of multiple ACK’s. The script ‘unwraps’ and interstates the BlockACK’s contents, turning every individual ACK into an ACK-frame using the algorithm described in the paragraph above.

The PCAP-to-CSV python-script uses pyshark to analyze the given pcap-file and the csv-library to write the csv file. To begin with, I started using scapy to analyze the pcap-file, like in the master-thesis, however I found the Wireshark GUI to be very useful for research, and I figured I might as well use the pyshark-library to get the corresponding fields that i needed and saw in Wireshark. While Wireshark clearly displays the PHY Network Standards (ranging from 802.11 FHSS to 802.11be), extracting this information programmatically through Scapy proved to be challenging.

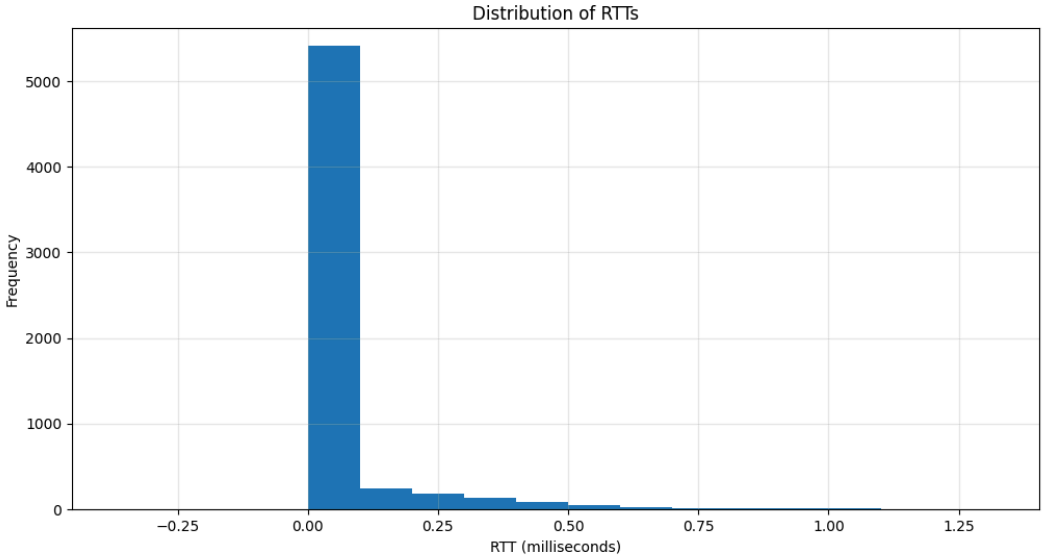
Plotter

Once the csv file is made, we can represent the data in a graph with the plotter. All the plotter does is make a graph of the csv file by using timestamp as the X-axis and Round Trip Time in milliseconds as the Y-axis. Here we use the “First Recorded Frame is Retransmission”-bool flag to mark a data-point either blue or red. If the flag is set to true, the point is marked red, meaning that the Round Trip Time is definitely higher, as the first Data-Frame recorded was a retransmission, meaning that the capture is missing the initial data-frame for some reason.





One can optionally display the plotter as a histogram, showing the frequency of RTT's on the y-axis and the RTT's on the x-axis.



Stats

We can also open the csv file with stats.py. This tool aggregates frame-type, source-, and destination-mac-addresses to give an overall insight of the status between particular network unit pairs. Its one line per pair per direction, The output might reveal interesting facts about a given pair of network-units, giving us a hint of what we should filter the csv file using a tool like grep.

```
(venv) richardmax@Richards-Air-2 submission % python3 stats.py output.csv

Metrics by Source-Destination pair:
                                        Frame Type  Total Packets  Total ACKs  Avg RTT  Bit Rate
Source            Destination
00:b7:71:b8:d7:c0 01:0b:85:00:00:00           Data              1           0      NaN       6.0
48:8b:0a:88:10:cf 48:8b:0a:88:10:cf           Data              7           0      NaN       6.0
5c:e9:31:63:a4:d6 01:00:5e:00:00:01           Data              1           0      NaN       6.0
                  a8:41:f4:e1:29:0e        QoS Data           842           0      NaN     960.8
                  ff:ff:ff:ff:ff:ff           Data              2           0      NaN       6.0
64:d6:9a:9e:0a:7f 6c:b2:ae:a7:c8:af        QoS Null             1           9   0.0551       6.0
6c:b2:ae:a7:d8:00 01:0b:85:00:00:00           Data              1           0      NaN       6.0
6c:b2:ae:ae:d6:80 01:0b:85:00:00:00           Data              1           0      NaN       6.0
Unknown           a8:41:f4:e1:29:0e        QoS Data            34           0      NaN    1080.9
a0:36:bc:99:f4:18 01:00:5e:7f:ff:fa           Data             22           0      NaN       6.0
a0:36:bc:99:f4:1c 01:80:c2:00:00:00           Data              8           2      NaN       6.0
                  01:80:c2:00:00:0e           Data              2           0      NaN       6.0
a8:41:f4:e1:29:0e 5c:e9:31:63:a4:d5  Null function          25837        6293   0.0579       6.0

Overall Statistics:
Total Packets: 26759
Total Retransmissions: 414
Average RTT: 0.0579 seconds
Overall Success Rate: 0.236
```

Installation

To install the WRAP-tool we need to initialize a python virtual environment and install the required packages. The easiest way to do this is by running the setup_venv.sh shell script and then activate the environment with `source venv/bin/activate`.
Alternatively, we can do it manually;

1. Create and activate a virtual environment.

   ```
   python3 -m venv env
   source venv/bin/activate
   ```

2. Install requirements

   ```
   pip3 install -r requirements.txt
   ```

Usage

PCAP-to-CSV

Run the program by providing a pcap-file as an argument:

```
python3 pcap_to_csv.py <capture_file.pcap> <desired_name_of_output_csv_file>
```

The desired name of the output csv file is optional, and will default to output.csv.

Plotter

Once we have the csv file, we can use the plotter like this;

```
python3 plotter.py output.csv
```

Stats

Likewise with Stats, we feed the csv file to get an overview like this;

```
python3 stats.py output.csv
```

Future Work

There are several ‘obvious’ possibilities for improvements of this tool. Given its novel nature, and my inexperience in the field of Network Communication, it's hard to determine whether the results of the tool actually are accurate, useful or straight up buggy.

More Accurate ACK source look up

The algorithm for determining an ACK frame's source address now uses a fixed-size data structure storing the last 128 sequence numbers per sender. This helps accurately match ACKs with their corresponding data frames, even when nodes are communicating with multiple partners simultaneously, and is essential for matching BlockACK’s. Future improvements could include correlating timestamps between ACKs and data frames to further validate these matches, or dynamically adjusting the sequence history size based on network conditions.

Compare to TCP-analyzers

As there are some developed tools like SPP and NETHINT, it might be useful to compare the results from these TCP-analyzers to see if there are any major discrepancies or similarities to the results of this tool.

Detailed Plotter

The plotter could encompass more features, showcasing different dimensions like throughput. It would be nice to differentiate between the units communicating with each other with different color schemes. The current plotter offers two visualization modes: a time-series plot showing RTT values over time with color-coding for retransmissions, and a histogram view showing the distribution of RTT values across different time intervals, helping identify patterns in network latency.

Performance, Python

The tool is written in python, which in itself is a slow language compared to lower level programming languages. Even though the tool uses pandas for performance gains, the script still takes a while to process larger pcap-files. It would be ideal to see the analysis-tool in a live-stream of data, giving a status of a network in real-time.

To achieve this the tool would probably have to be re-written in a more performant language like C, Mojo or Rust. I was actually looking into rewriting it in Rust, some quick research does support a bunch of pcap analysis libraries. Another possible solution is to wrap the Python code with Mojo syntax, to enable better performance.

Conclusion

This is novel and difficult work, but the useful potential of this tool might be huge. As we access the frames on the lower level of the OSI we can get more accurate readings and analysis of a network's status than the TCP-layer.

WRAP could be good for identifying bottlenecks in a network, and which MAC-adresses might be causing issues, or at least which MAC-adresses communicate badly with each other.

Acknowledgments

A huge thanks to my supervisors Magnus Skjegstad and Michael Welzl who made all this work possible. As a novice Network-student and software-developer, they've shown me the way through the dark from beginning until the end. A lot of this work is pretty novel, meaning that it's hard to predict what results to expect, and without their expertise, I would be completely lost.

References

DISCO - https://prosjektbanken.forskningsradet.no/en/project/FORISS/338740

TCPDump - https://www.tcpdump.org/

WireShark - https://www.wireshark.org/

Airodump - https://www.aircrack-ng.org/doku.php?id=airodump-ng

NETHINT - https://www.duo.uio.no/bitstream/handle/10852/103924/Petter-Juterud-Barhaugen_Master-Thesis.pdf?sequence=1&isAllowed=y

Scapy - https://scapy.net/

PyQT - https://wiki.python.org/moin/PyQt

PyQTGraph - https://www.pyqtgraph.org/

PyShark - https://pypi.org/project/pyshark/

MatPlotLib - https://matplotlib.org/

Pandas - https://pandas.pydata.org/

Synthetic Packet Pairs - http://caia.swin.edu.au/tools/spp/

Passive Ping - https://github.com/pollere/pping