TSG IT Systems

project report

Machine-Learning-based Link Fault Detection:

Date	Version	Author	Validation status	Checker
04.08.2020	1.0	Galina Blokh	Not validated	L. Schvartser Samuel Guerchonovitch

TSG IT Systems

1. Introduction

1.1 Purpose

The purpose of this project is the same it was in first time - **to find a way to predict with a low cost a link fault in a complex network, changing its shift very quickly.** The main idea is to be able, by watching the flows in the network and knowing the initial state of the topology, to determine the entrance or the fault of a link in this network by simply monitoring (1).

There are several main points to discover in this project:

- Understand when and where is a fault appearing, in order to deploy an immediate correction (1)
- Understand how the model is facing the change in the topology due to the integration of a new node (1)
- Be able to predict the impact of a new node integrating the current topology using Random Forest, XGBoost ML models. Evaluating ML model with ROC curve, Area Under the Curve, Precision, Recall, Confusion matrix
- Stress test the network on more complex architectures (with 30/60 nodes). The topology can be defined as the blocks with same structure connected by definite links. The same test as on the small network using Random Forest, XGBoost ML models
- Test the network on an architecture closer to the client's.

1.2 Context & Assumptions

We would like to restore previous works topology. Therefore, we use a lab version, which implies several assumptions:

- All the nodes used in lab are virtual machines, built the same way, depending on a personal computer (1)
- The behavior of the virtual nodes is necessarily different from a behavior of several physical routers of different age, craftmanship and use (1)
- All the monitoring has been done in automatic way, using python script, what is **different from (1),** where the the monitoring has been done partly hands-on.
- We are assuming that only one link fault is occurring at a time. As the model may, on term, deal with the scale of the microsecond, modeling a double link fault at the microsecond seems rare enough to consider this assumption valid (1)

1.3 Definitions & Data to use

As the paper suggested and it was used in the first try, we will use the most basic information a capture of the network would provide:

- **Packet Flow Rate** is the most important information, as we are expecting the flow rate to change according to the dynamic routing (1)
- **Time to Live** may have a pretty high entropy when dealing with dynamic routing. If TTL changes in average of more than 1, we can expect that at least 1 hop is added in the routing. Hence avoiding the case where the delay remains slightly the same, but the route is changing (1)
- **Delay** is also an important feature as a different route may lead to different cost in delay (1)
- **Packet Loss** can be a good way of identifying link drop event as the packets sent during the route recalculation will eventually be dropped and resent in most cases. It can then help the model discriminate between the real faults and other changes in the traffic (such as a usage peak) (1)

2. Models & Principle & Experiment

2.1 Model presentation

The simple symmetric network topology to restore, which looks like:

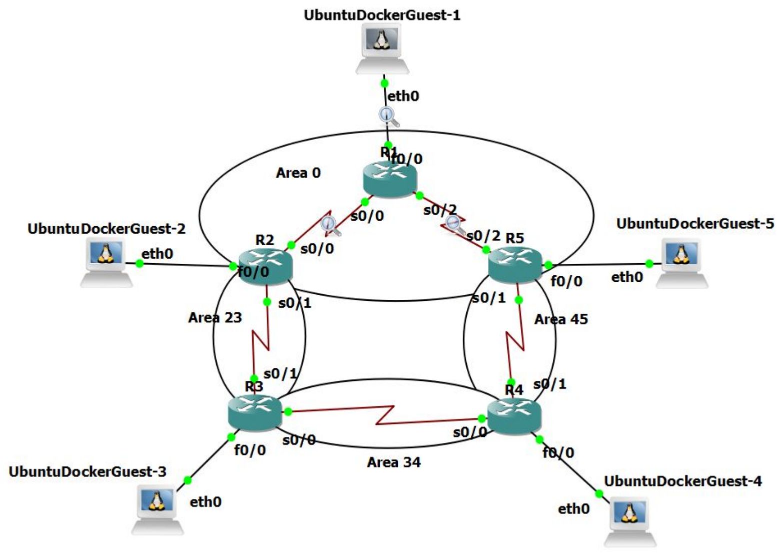


Fig.1 5-nodes loop model in GNS3 (1)

We used GNS3 2.02 to simulate network nodes, mainly Cisco routers c3640-a3js-mz.124-25d.image. These routers allows configure multiple types of links, connections and choose the optimal with speed traffic.

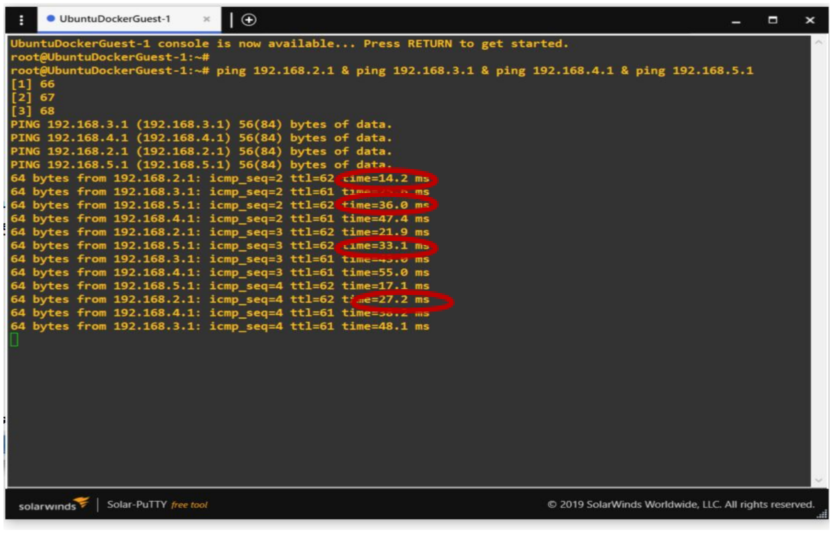
As it shown on the Fig.1, we used Docker containers as end points, but this time it was such as AlpineLinux, ipterm, Kali Linux CLI, Network Automation, not only Ubuntu Docker Guest (as on the Fig.1).

All new models of docker containers this time were built in the topology for more convenient use: they are containing Ubuntu/Kali for parallel requests and have already pre-installed such modules as python 2, python 3, python scapy, python paramiko, thus we could check automatical capturing traffic around the node 1 and were able to do our stats before creation additional handle docker images for transferring outside the network.

We deployed the OSPF protocol for dynamic routing between the routers, dealing with the redirection of the flow in the link fault scenario. A pro version_15 of VMWare is used for the virtual machines in Linux (1)

2.2 Operating principle

We then request a ping from every endpoint to every other endpoint in the same way as it was made in previous try. When the network becomes stable , we identify the delay encountered by the ping in the red circles (Fig.2). We can then establish a mean delay per request per destination (1):



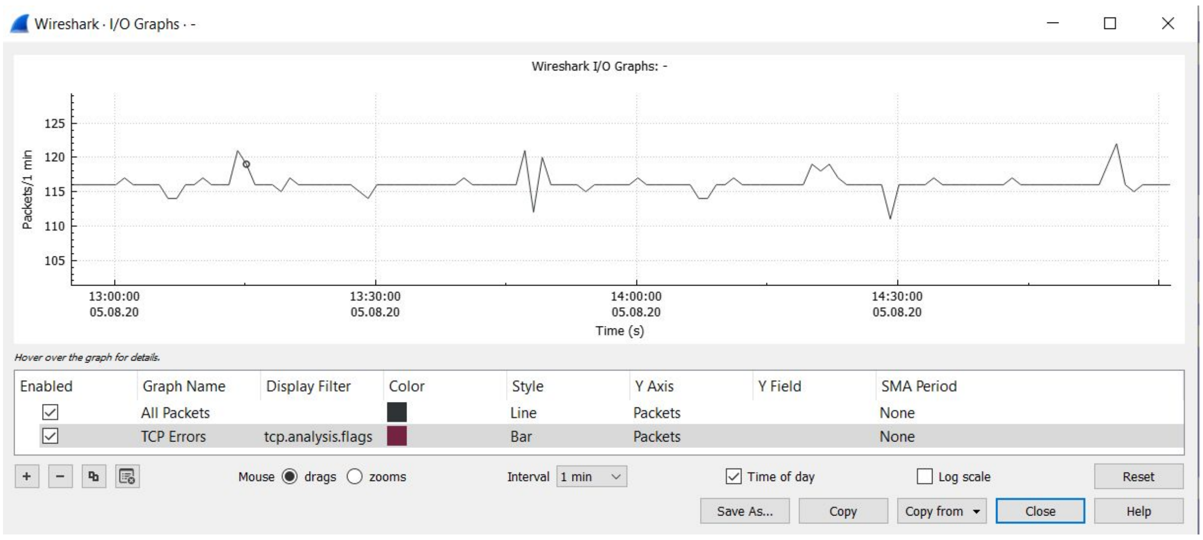
```
UbuntuDockerGuest-1 console is now available... Press RETURN to get started.
root@UbuntuDockerGuest-1:~#
root@UbuntuDockerGuest-1:~# ping 192.168.2.1 & ping 192.168.3.1 & ping 192.168.4.1 & ping 192.168.5.1
[1] 66
[2] 67
[3] 68
PING 192.168.3.1 (192.168.3.1) 56(84) bytes of data.
PING 192.168.4.1 (192.168.4.1) 56(84) bytes of data.
PING 192.168.2.1 (192.168.2.1) 56(84) bytes of data.
PING 192.168.5.1 (192.168.5.1) 56(84) bytes of data.
64 bytes from 192.168.2.1: icmp_seq=2 ttl=62 time=14.2 ms
64 bytes from 192.168.3.1: icmp_seq=2 ttl=61 time=[illegible] ms
64 bytes from 192.168.5.1: icmp_seq=2 ttl=62 time=36.0 ms
64 bytes from 192.168.4.1: icmp_seq=2 ttl=61 time=47.4 ms
64 bytes from 192.168.2.1: icmp_seq=3 ttl=62 time=21.9 ms
64 bytes from 192.168.5.1: icmp_seq=3 ttl=62 time=33.1 ms
64 bytes from 192.168.3.1: icmp_seq=3 ttl=61 time=[illegible] ms
64 bytes from 192.168.4.1: icmp_seq=3 ttl=61 time=55.0 ms
64 bytes from 192.168.5.1: icmp_seq=4 ttl=62 time=17.1 ms
64 bytes from 192.168.2.1: icmp_seq=4 ttl=62 time=27.2 ms
64 bytes from 192.168.4.1: icmp_seq=4 ttl=61 time=[illegible] ms
64 bytes from 192.168.3.1: icmp_seq=4 ttl=61 time=48.1 ms
```

Fig. 2 -one of 5 end-points console, model in GNS3 from Fig.1 (1)

Next step is to get datetime information and make our collected data as TimeSeries using scapy library from python 3.

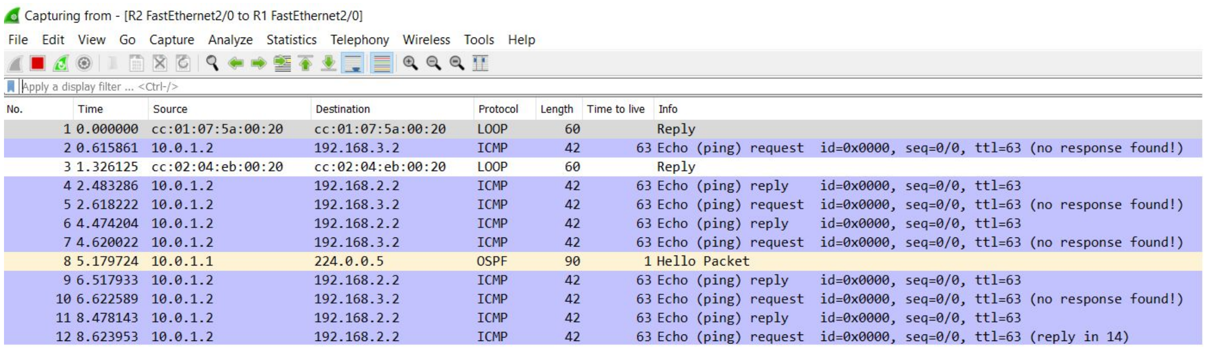
But before that we will monitor it with Wireshark.

This application provides some way of observing the flow rate, which allows to pick the values for each destination (1)



Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period
☑	All Packets		⬛	Line	Packets		None
☑	TCP Errors	tcp.analysis.flags	🟥	Bar	Packets		None

Fig.3 -whole end-points sending packages each other ,Graphique I/O



Capturing from - [R2 FastEthernet2/0 to R1 FastEthernet2/0]
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time      Source             Destination        Protocol  Length  Time to live  Info
   1 0.000000  cc:01:07:5a:00:20  cc:01:07:5a:00:20  LOOP      60               Reply
   2 0.615861  10.0.1.2           192.168.3.2        ICMP      42            63 Echo (ping) request  id=0x0000, seq=0/0, ttl=63 (no response found!)
   3 1.326125  cc:02:04:eb:00:20  cc:02:04:eb:00:20  LOOP      60               Reply
   4 2.483286  10.0.1.2           192.168.2.2        ICMP      42            63 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63
   5 2.618222  10.0.1.2           192.168.3.2        ICMP      42            63 Echo (ping) request  id=0x0000, seq=0/0, ttl=63 (no response found!)
   6 4.474204  10.0.1.2           192.168.2.2        ICMP      42            63 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63
   7 4.620022  10.0.1.2           192.168.3.2        ICMP      42            63 Echo (ping) request  id=0x0000, seq=0/0, ttl=63 (no response found!)
   8 5.179724  10.0.1.1           224.0.0.5          OSPF      90             1 Hello Packet
   9 6.517933  10.0.1.2           192.168.2.2        ICMP      42            63 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63
  10 6.622589  10.0.1.2           192.168.3.2        ICMP      42            63 Echo (ping) request  id=0x0000, seq=0/0, ttl=63 (no response found!)
  11 8.478143  10.0.1.2           192.168.2.2        ICMP      42            63 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63
  12 8.623953  10.0.1.2           192.168.2.2        ICMP      42            63 Echo (ping) request  id=0x0000, seq=0/0, ttl=63 (reply in 14)
```

Fig.4 - Wireshark shows the signals with lost packets (having no response)

With this hands-on method, by observing data, we won't be collecting it, but we will check, does our automatisation program (which written in python for data collection) collects the correct data.

2.3 Experiment

As a reminder, we are aiming to scale the experience on more complex, not symmetrical and node-independent network structure (1), which clear we can see on the Fig.5:

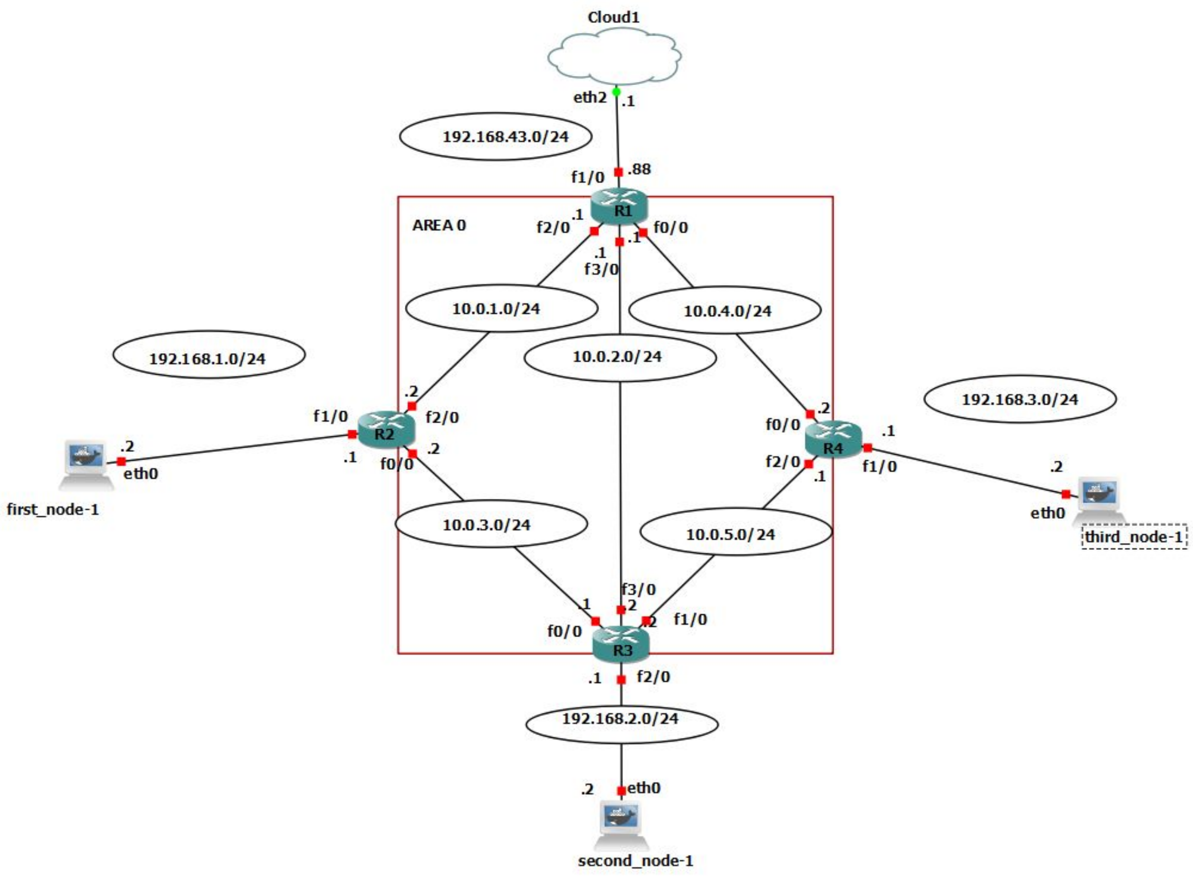


Fig.5 - A more complex, non-symmetric architecture

Therefore, we wasn't going working hands-on. The following has been deployed:

- The data capture must be done on the node itself instead of observing its *In/Out* transfers,

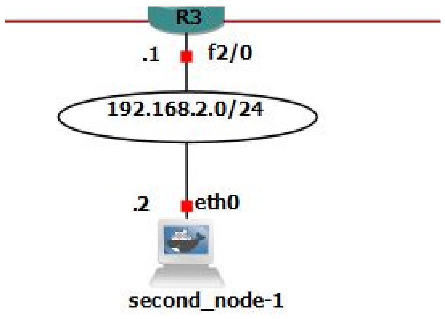
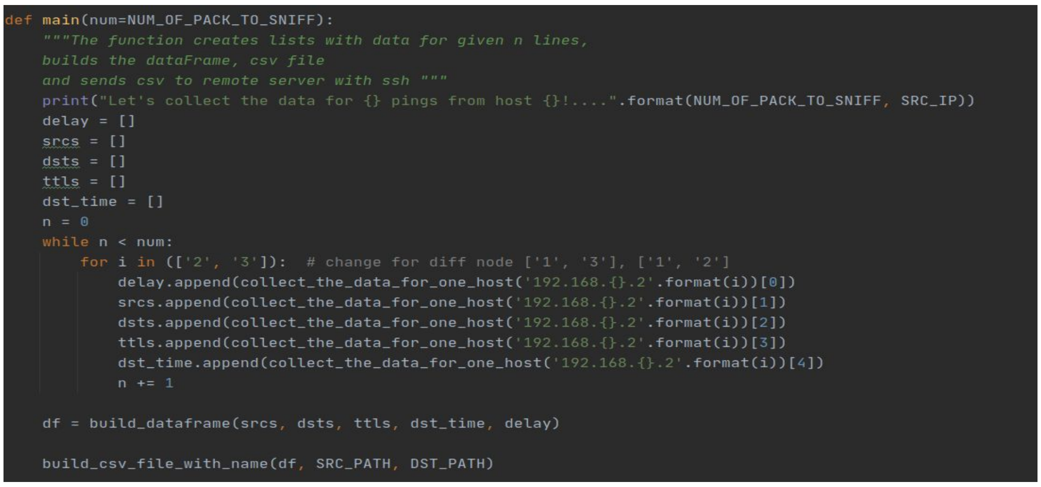


Fig.6 - One of 3 end-points sending and receiving packets in non-symmetric architecture

- The monitoring of the data must be centralized and sent regularly, create list of objects to restructure into data frames



```
def main(num=NUM_OF_PACK_TO_SNIFF):
    """The function creates lists with data for given n lines,
    builds the dataFrame, csv file
    and sends csv to remote server with ssh """
    print("Let's collect the data for {} pings from host {}!....".format(NUM_OF_PACK_TO_SNIFF, SRC_IP))
    delay = []
    srcs = []
    dsts = []
    ttls = []
    dst_time = []
    n = 0
    while n < num:
        for i in (['2', '3']):  # change for diff node ['1', '3'], ['1', '2']
            delay.append(collect_the_data_for_one_host('192.168.{}.2'.format(i))[0])
            srcs.append(collect_the_data_for_one_host('192.168.{}.2'.format(i))[1])
            dsts.append(collect_the_data_for_one_host('192.168.{}.2'.format(i))[2])
            ttls.append(collect_the_data_for_one_host('192.168.{}.2'.format(i))[3])
            dst_time.append(collect_the_data_for_one_host('192.168.{}.2'.format(i))[4])
            n += 1

    df = build_dataframe(srcs, dsts, ttls, dst_time, delay)

    build_csv_file_with_name(df, SRC_PATH, DST_PATH)
```

Fig.7 - Function which send and receives packets from 1 node, saves data into csv file and sends it to the local machine (4)

In this function was implemented conditions for creating, monitoring and saving the traffic flow

- The network must be more realistic, simulating *"real"* conversations (TCP conversations for example) instead of ICMP messages such as the pings - for this we were using scapy python library, in which we able to put any needed protocol as a parameter.

- Labelling for work/fault:

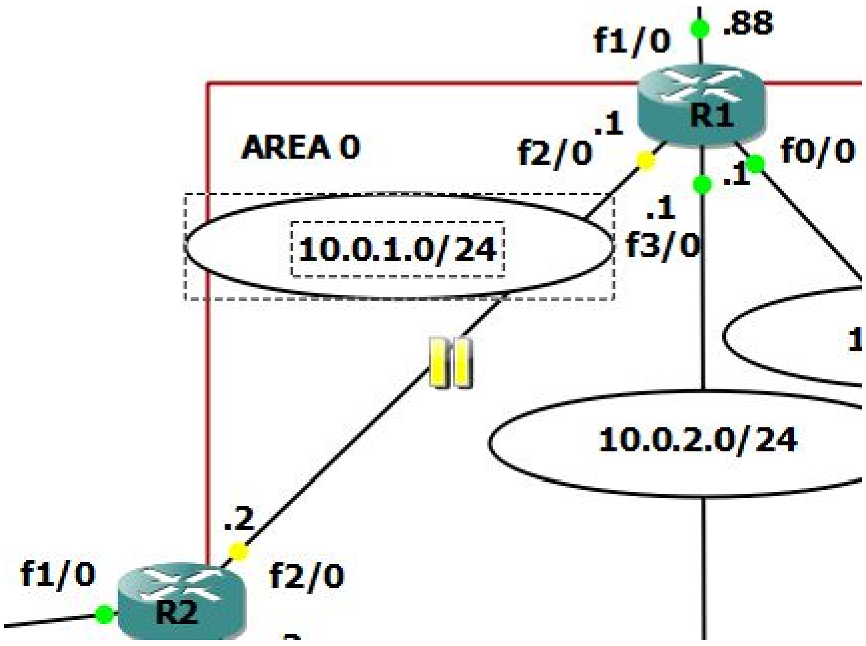
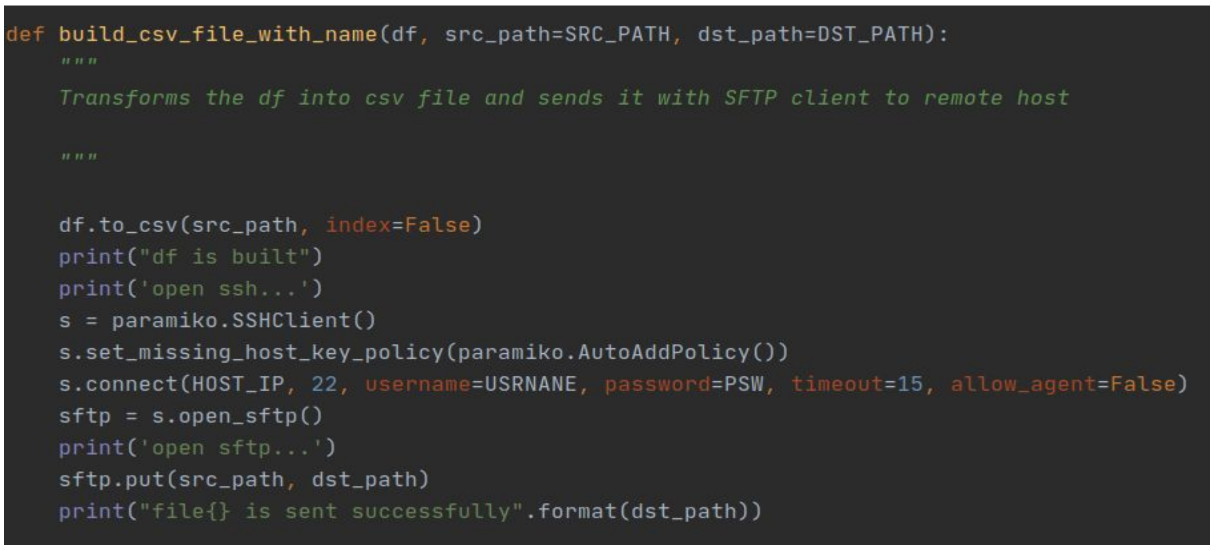


Fig.8 - Suspend option in GNS3 application imitates fault in network connection (yellow "pause")

A workaround has been to extract the data on the fly after collecting it. For this, we have done the following via Python scapy and paramiko libraries:

- Transform the data into csv (using the Python *Pandas* library) - code implementation on Fig.9 (4)
- Transfer the data through the internet to the local machine via paramiko python library, as GNS3 allows internet



```
def build_csv_file_with_name(df, src_path=SRC_PATH, dst_path=DST_PATH):
    """
    Transforms the df into csv file and sends it with SFTP client to remote host

    """

    df.to_csv(src_path, index=False)
    print("df is built")
    print('open ssh...')
    s = paramiko.SSHClient()
    s.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    s.connect(HOST_IP, 22, username=USRNANE, password=PSW, timeout=15, allow_agent=False)
    sftp = s.open_sftp()
    print('open sftp...')
    sftp.put(src_path, dst_path)
    print("file{} is sent successfully".format(dst_path))
```

Fig.9 - Function which transforms data frame into csv file and send it to the local machine using sftp connection (4)

In the **(1)**, were used transfer **to GitHub** using python library paramiko**.**

This time the data were transferred **to the local machine** using sftp connection with the same python library.

- connection via NAT**.** - Here we decided **improve (1)** and **used Cloud** image **instead NAT**. Cloud node, comparing with **NAT node, allows both side connection** and transfer in/out network. NAT gives only access outside the network.

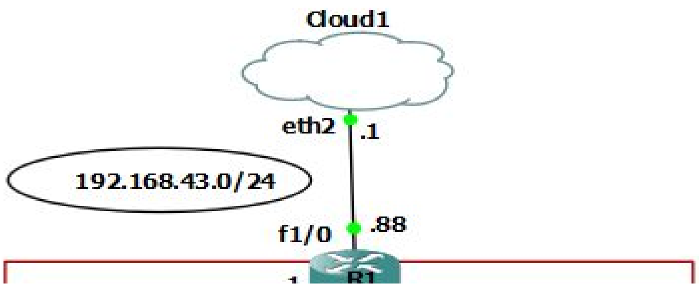
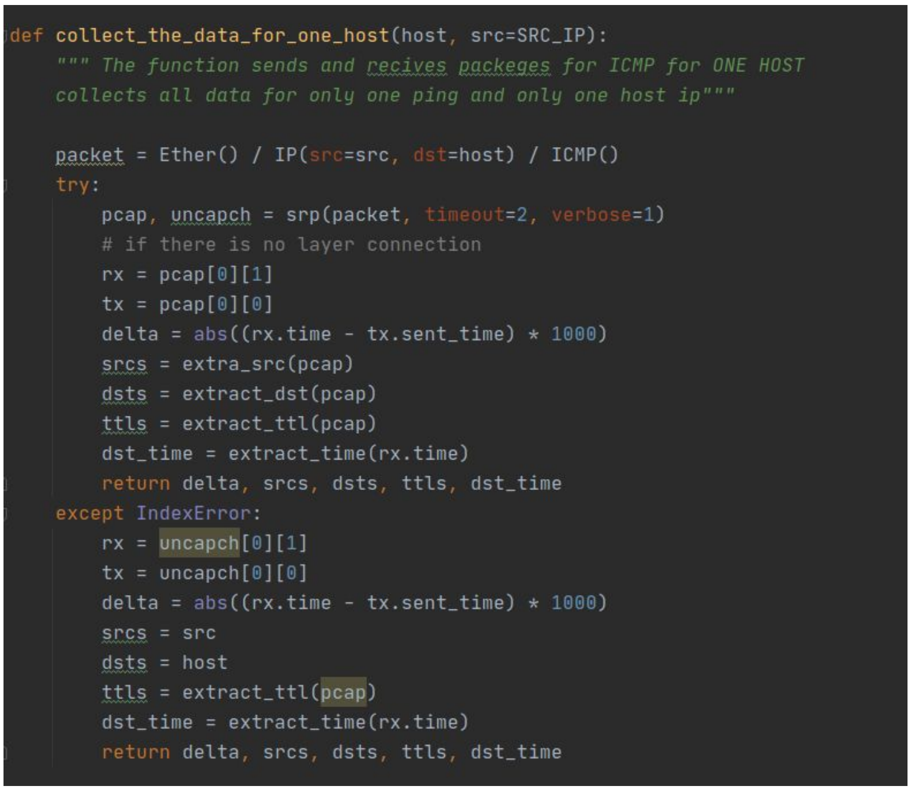


Fig.10 - Cloud node instead NAT node allows both side connection and transfer in/out network

- Sniffing up to a specified number of packets and return a pcap file - in this point **we didn't used sniff()** functions from scapy python library (as it was in the first try). Sniff() just scanning whole network from one node.

  The condition was "The data capture must be done on the node itself instead of observing its *In/Out* transfers", thus **we used srp()** function, which sends and receives packets. This allowed to get time data of sending and receiving packets.

  It wasn't made in (1) by the objective reason.



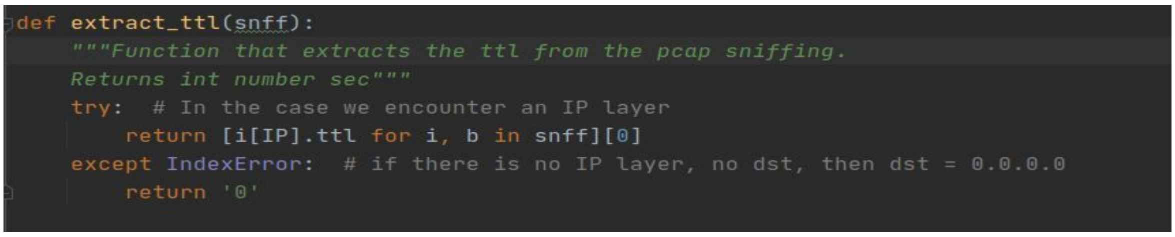
```
def collect_the_data_for_one_host(host, src=SRC_IP):
    """ The function sends and recives packeges for ICMP for ONE HOST
    collects all data for only one ping and only one host ip"""

    packet = Ether() / IP(src=src, dst=host) / ICMP()
    try:
        pcap, uncapch = srp(packet, timeout=2, verbose=1)
        # if there is no layer connection
        rx = pcap[0][1]
        tx = pcap[0][0]
        delta = abs((rx.time - tx.sent_time) * 1000)
        srcs = extra_src(pcap)
        dsts = extract_dst(pcap)
        ttls = extract_ttl(pcap)
        dst_time = extract_time(rx.time)
        return delta, srcs, dsts, ttls, dst_time
    except IndexError:
        rx = uncapch[0][1]
        tx = uncapch[0][0]
        delta = abs((rx.time - tx.sent_time) * 1000)
        srcs = src
        dsts = host
        ttls = extract_ttl(pcap)
        dst_time = extract_time(rx.time)
        return delta, srcs, dsts, ttls, dst_time
```

Fig.11 - Function which collects the data from 1 node (4)

In this function, on the check stage, we were using ICMP() protocol as parameter, to capture the flow without internet connection in 5 nodes network (Fig.1). For the final run we suppose to use TCP() protocol as parameter and network with internet access (Fig.5).

Also, srp() function allows access to sent packets with 'no response' - this gave us time delay mesure. At first there was an idea to dump pcap file and extract the data from it, but with static file we lost access to time information of 'no response' packets, it became '00:00:00'. So we captured and **extracted all the data 'on the fly' directly on each node and from each sent packet.**

- Access the ttl of all the sniffed packets



```
def extract_ttl(snff):
    """Function that extracts the ttl from the pcap sniffing.
    Returns int number sec"""
    try:  # In the case we encounter an IP layer
        return [i[IP].ttl for i, b in snff][0]
    except IndexError:  # if there is no IP layer, no dst, then dst = 0.0.0.0
        return '0'
```

Fig.12 - Function which collects the data from pcap file (ttl value) (4)

In the same way we extracted destination ip, sending source ip, source time for each sent packet

- Integrate the python script and the scp service in the GN Software - GNS3 allows the use of Docker containers, we are built the docker containers on a Virtual Machine and ran directly the script on start , replacing each end point with our handle way created docker image with python script inside.

We **didn't used Ostinato** /traffic generator. All **traffic generation was implemented with docker images** on each end point using python script (Fig.5, Fig.6), as well as data capturing and transferring.

- To finish to parametrize correctly the transfer.

  Up to this part we got without any issues (1). The program (all previous steps together) was creating, collecting, extracting correct traffic flow and data.

3. Issues & Solutions

3.1 Issues

So far as the ‘Labelling for work/fault’ stage was the last for data collecting, we got a serious issue at that time - **OSPF routing protocol stopped work in correct way**. Empirically we determined that topology without internet connection and Cloud node has a very good, stable OSPF routing run; but with providing internet connection with Cloud/NAT node, OSPF routing information on each router became partially lost. The Issue became clear visible only when we started suspend the links, imitating fault connection.

The **reasons** why this issue appeared may be different:

- new version of GNS3 application, which we used to create the topology and to imitate the network work, has bugs;
- the router settings was changed with time progress (in global worldwide meaning)
- OSPF routing protocol incompatible with Internet access in GNS3 application
- some other reasons which can explain only very experienced network engineer

This point has a key importance: **impossibility to collect the correct data blocks the main goal of this project**: Machine-Learning-based Link Fault Detection (data science part)

Apart from this issue, everything is set and running. We are able to collect the data without “fault connections” and we missing only data with ‘fault connections’.

3.2 Solutions

Solutions can be also several:

- To involve in this question a very very experienced network engineer;
- To change the application for network work imitation;
- To use automatisation: with python libraries telnet and paramico to configure network without internet access with correct working OSPF routing protocol; run docker images and collect the data; include into python script (using python libraries telnet and paramico) functions which will configure internet access in the topology when the data (correct data with 'faults') will be collected; and send this data to the local machine.

4. Conclusions

- We have done a great work with improving the (1) in the engineer part:
  - we were using more modern/advanced tools for data collecting and network configuration
  - observed data was capturing on each node with docker image in automatic way using python script
  - got all needed types of data to create timeSeries object in ML part except target variable
- The Machine Learning based technology to detect and localize the link faults in IoT architectures is that what wasn't succeeded in this second try of the project
- To go through this we have to use the solution from the paragraph 'Solutions' or find others
- After the solution will be implemented and the issue will disappear, it will be possible to:
  - Predict the impact of a new node integrating the current topology using Random Forest, XGBoost ML models. Evaluating ML model with ROC curve, Area Under the Curve, Precision, Recall, Confusion matrix (1)
  - Make stress test for network on more complex architectures (with 30/60 nodes).
  - Have advanced option: to test ML model on an architecture closer to the client's.

5. References & Links

1. “Proof of Concept: Machine-Learning-based Link Fault Detection” , “TSG IT Systems”, Samuel Guerchonovitch (08.04.2020)
2. TE-Based Machine Learning Techniques for Link Fault Localization in Complex Networks, Srinikethan Madapuzi, Tram truong-Huu, Mohan Gurusamy, NU Singapore (2016)
3. Machine Learning-based Link Fault Identification and Localization in Complex Networks, Srinikethan Madapuzi, Tram truong-Huu, Mohan Gurusamy, NU Singapore (2019)
4. Gitlab repository https://gitlab.com/amproyGit/nfd/-/tree/Galina_Dev
5. GNS3 Tutorial https://www.youtube.com/watch?v=Ibe3hgP8gCA&list=PLhfrWIlLOoKNFP_e5xcx5e2GDJIgk3ep6