Technical Report

UCAM-CL-TR-1000
ISSN 1476-2986

Number 1000

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Application identification in data centres: a traffic driven approach

Mihnea-Stefan Popeanga

August 2025

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 2025 Mihnea-Stefan Popeanga

This technical report is based on a dissertation submitted June 2025 by the author for the degree of Master of Philosophy (Advanced Computer Science) to the University of Cambridge, St Edmunds College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-1000*

Abstract

Modern data centre (DC) operators cannot tune and secure what they cannot see. However, application identification from network traces is held back by two obstacles: public packet captures are scarce because commercial workloads and user data are confidential; the few datasets that exist do not focus on DC specific workloads, and do not allow others to reproduce the experiments. This dissertation tackles both these issues. I designed and implemented an end-to-end framework that can systematically capture traffic with nanosecond timestamps, demultiplex flows, and compute a set of 203 features. Each flow is coupled with extensive metadata detailing the exact setup that generated the traffic, allowing any researcher to reproduce the experiments under identical conditions. Using this workflow, I created the first public DC-focused dataset, unencumbered with personal or confidential information, that spans three representative workloads. Machine learning classification techniques demonstrate the utility of the data: traditional feature-based models achieve perfect accuracy when identifying the three workloads. A core novelty is that besides strict identification, the collected data includes significant metadata. To demonstrate this, I tackled performance estimation as well for one of the workloads. A 1D CNN can distinguish between flows corresponding to different performance metrics with an accuracy of 95%.

Contents

Chapter 1

Introduction

Modern data centres are the backbone of the most used digital services, processing and storing large quantities of information. Within data centres at any point there are a vast number of applications running. These are interconnected, and the data exchanges between them generate complex and high-volume network traffic. Effectively managing and optimising data centre operations necessitates a deep understanding of this traffic. The first component of this understanding is the ability to identify the running applications accurately. This dissertation explores this task of application identification for data centre specific workloads, using a traffic-driven approach. It presents a novel framework for creating comprehensive datasets that capture the different behaviours, and evaluates machine learning techniques for flow level traffic classification.

1.1 Motivation

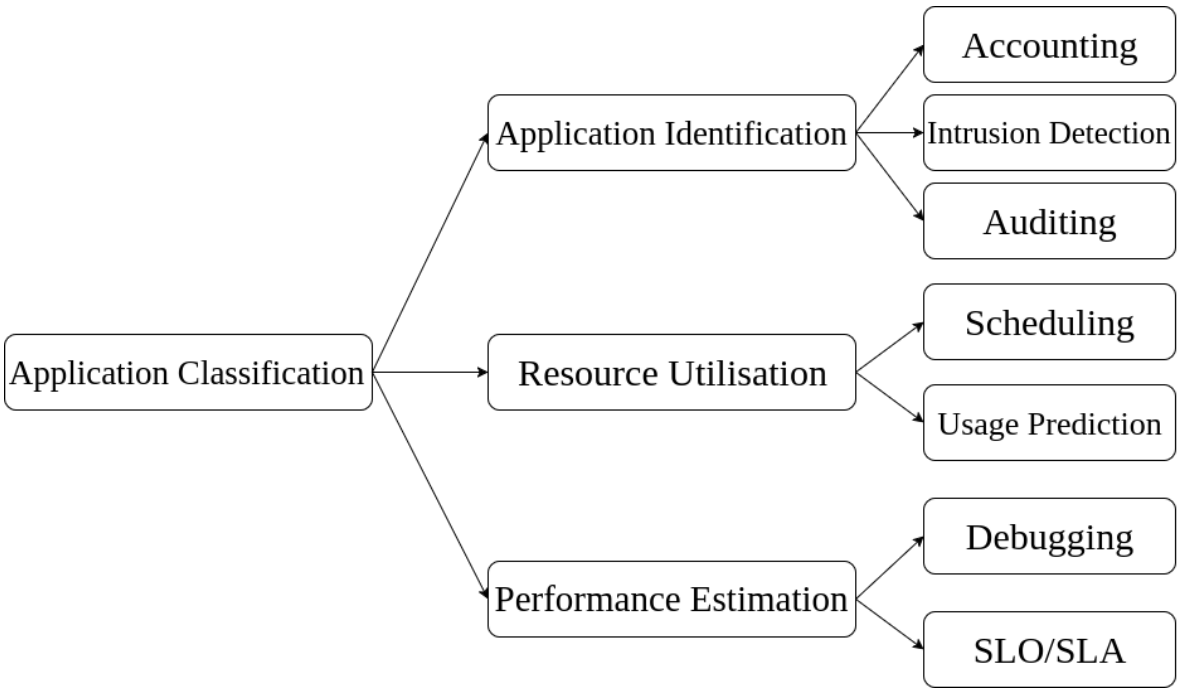

Figure 1.1: Example uses of application classification based on the classification target.

Application classification is an important problem in today's computing landscape, par-

ticularly in data centres. The targets of classifications may vary depending on the desired usage. Figure 1.1 presents a few different targets and common uses for them.

In data centres, by leveraging classification, operators can perform critical tasks such as intrusion detection, capacity planning, and QoS assurance [1, 2, 3]. Given the diverse nature of workloads in data centres and their specific requirements, application level classification is especially crucial for maintaining predictable performance. It can enable advanced performance diagnosis by correlating traffic patterns with varying load (e.g. different measured queries-per-second), guiding targeted optimisations [4]. Moreover, detailed traffic characteristics can serve as signals for workload placement and resource allocation in cluster scheduling [4, 5, 6]. Accurate classification of workloads, based on their resource usage, can also lead to lower carbon footprints for data centres [7].

1.2 Research Gap

This project is concerned with traffic classification within the scope of data centre networks. Upon inspection of existing literature (detailed in Chapter 3), two significant gaps appear.

Firstly, despite the unique characteristics of data centre networks, there is no existing literature dedicated to classifying application traffic within this environment.

Secondly, the limited research existing on data centre network characteristics is reliant on closed proprietary datasets. This lack of public data acts as a significant impediment to reproducible research and comparative analysis of network traffic classification techniques. Furthermore, the reasons for the lack of public network traces are user privacy and confidentiality concerns. Multiple approaches have been proposed to facilitate the sharing of network traces [8, 9, 10], but so far, it remains a major drawback in network research.

Therefore, a clear research gap exists in the development of network traffic classification techniques specifically designed and evaluated on realistic data centre application workloads for which the data is public and the experiments are reproducible. Furthermore, there is an unmet need for more granular classification that moves beyond the simple identification of applications. Network traffic classification can be used to understand the operational state of DC workloads (e.g. performance). The main current issue that blocks the progress is the labelling techniques used in existing datasets [11]. This project aims to cover this gap by introducing a framework enabling reproducible data centre application classification and dataset creation with integral comprehensive labelling.

1.3 Summary of Contributions

The main contributions of this project are:

- The creation of a tool-chain that can reliably run workloads on a set of distributed machines and capture their network traffic with nanosecond timestamps. This is an extensible framework that allows the collection of more than just the network traffic associated with the running workload, but instead captures the entire system configuration along with other metadata[1].

[1]Source code for all used tools can be found at `https://gitlab.developers.cam.ac.uk/msp57/dc_workload_classify`. To access it, contact my supervisor Andrew Moore.

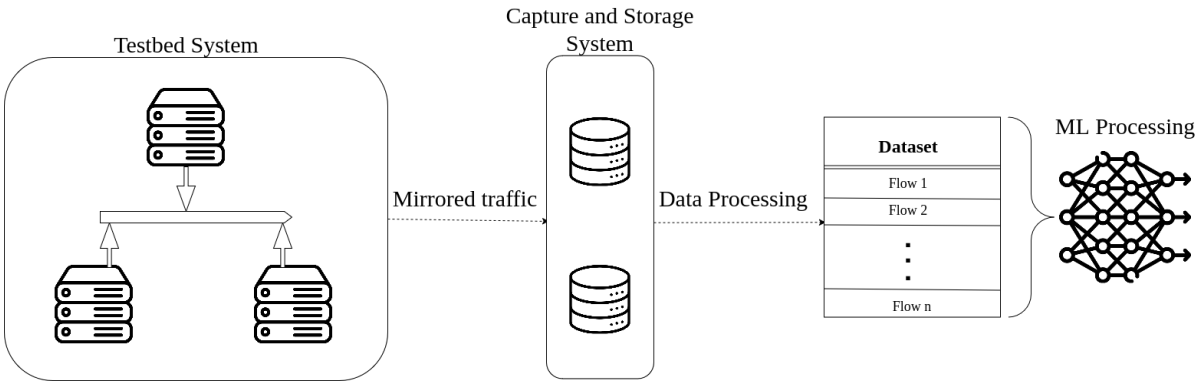

Figure 1.2: Overview of the project's components, the capture infrastructure, data processing pipeline, and the ML classifiers.

- The creation of a workflow that processes network traffic captures and creates feature-based datasets of approximately 200 features. A core novelty is that the labelling is not static, each dataset can be tagged on-the-fly using extracted information from any of the metadata associated with a workload run. This enables analyses that reach well beyond application identification.
- A comprehensive dataset built using the above tool-chains describing the network traffic of three data centre specific workloads.
- An evaluation of network traffic classification techniques using machine learning on the created dataset. Besides application identification, I also evaluated the ability to estimate a performance metric, such as queries-per-second, (QPS) for one of the selected workloads.

The components are summarised in Figure 1.2. The figure shows the designed system flow.

Chapter 2

Background

2.1 Data Centre Networks

The focus of this project is on network traffic in data centres. Modern data centres (DCs) have to accommodate unique traffic patterns of demanding workloads. Network architectures of DCs are derived from the Clos network topology [12, 13, 14], the most common being the *fat tree* [15] architecture. Servers are placed in racks and are connected to a Top-of-Rack (ToR) switch. Then, ToR switches are connected to multiple levels of spine switches, allowing for multiple paths between any two servers. This is because modern DCs are dominated by serve-to-server (East-West) traffic which is orders of magnitude higher than client-to-server (North-South) traffic [16]. A *fat tree* topology is shown in Figure 2.1.

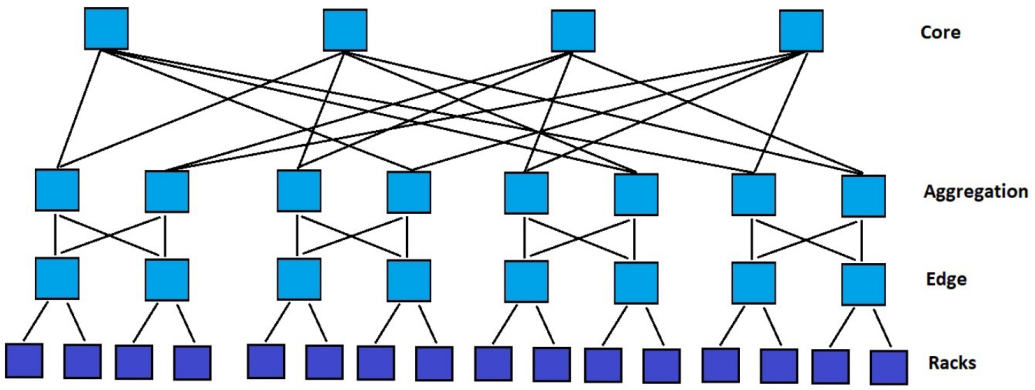

Figure 2.1: A data centre *fat tree* topology. Dark blue squares are servers and light blue squares are switches. The labelled edge switches are the ToR switches, core and aggregator are the spine switches. Figure taken from [4].

One key characteristic of data centre networks (DCNs) is the high bisection bandwidth[1], which is constantly increasing and has reached multi-petabit per second [17]. Another crucial characteristic is the low latency, as many common services require low tail latency [18]. In large DCNs, end-to-end latency is in the range of tens of microseconds [19], as increased latency may considerably decrease performance as shown by Zilberman *et al.* [20].

[1]Minimum aggregate bandwidth across a cut that divides the network into two equal halves.

2.2 Classifying Workloads

The way workloads are classified depends on the scope and on the target of the classification. The choice of data and classifying unit[2] is related to what the expected outcome is. Example targets for classification are application names, service types, performance regimes, and resource usage. This information can be used for better resource utilisation predictability and estimation, as explained in Section 1.1.

Identifying a particular application may be straightforward, especially in data centres where workload types are vastly different (Section 3.2). Learning more about them, such as how they perform and how they use the different resources, may be a more complex task. In both industry and research, tracing is used extensively to reveal information about a workload. This may be used in debugging, accounting, auditing, or verification. Many tracing tools have appeared in both academic and industry settings. Some examples are KUTrace [21], Dapper [22], and Magpie [23]. An important issue is that in distributed systems, tracing individual requests is difficult [24]. Furthermore, all these tracing systems are intrusive: they require changes at either system or application level, which may impact the performance of the traced workloads.

In this landscape, classification using network observations alone can be a valid alternative. Collecting the network traffic is non-intrusive and does not affect performance in any way. So the idea of analysing network traces to answer the same questions as system traces is a promising, yet unexplored one. Network traces could reveal even more information if used together with simple system traces.

2.2.1 Network Traffic Classification

This project is mainly concerned with TCP communication using Ethernet as the Data Link Layer and optical fibre as the physical transmission media. The packet size represents the size of the Ethernet frame excluding the preamble bytes and the mandatory IPG (inter packet gap) bytes. The inter-arrival time (IAT) between two packets represents the distance in time between the starts of two consecutive packets. The inter-packet gap (IPG) represents the distance between two consecutive Ethernet frames and is approximately equal to the IAT minus the time on wire. These concepts are shown in Figure 2.2.

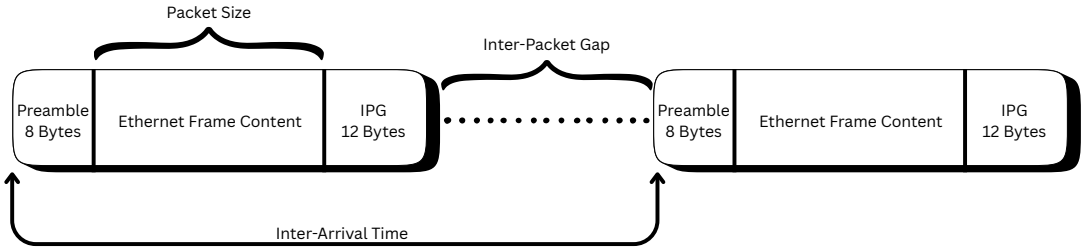

Figure 2.2: Basic definitions for Ethernet traffic. Based on a figure from [25].

When classification is considered, the first question to answer is what will be the input? In the case of network traces, the unit of classification is a part of the trace. Depending on the scope, different granularity may be needed when the trace is split. Different scopes that are relevant in a data centre environment are shown in Figure 2.3.

[2]The input to the classification model.

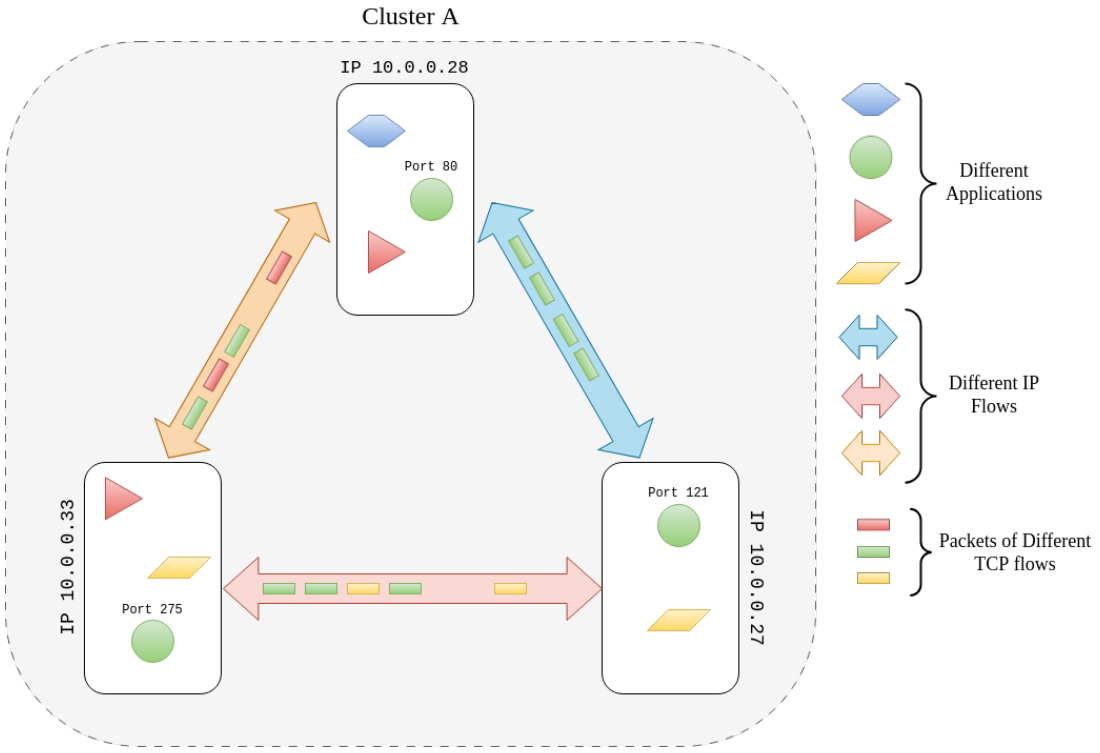

Figure 2.3: Simplified view of a computer network with three machines running distributed applications. Each shape represents a different application. In order to extract information about a particular application, we have to look at traffic originating and targeting only that application, in this case different TCP flows. If we are interested in a particular machine, we ignore the different TCP flows, and we have to look at individual IP flows, which are identified only by the machines' IP addresses. If we want to get information about the cluster as a whole, we may need to aggregate all information in the network, regardless of the target IP or network port.

From Figure 2.3, we can see that depending on the scope, the classification unit may be different, and the traffic may split at different granularities. In case the focus is on individual applications, the sensible choice is a flow defined as the 5-tuple $(protocol, src\_ip, src\_port, dst\_ip, dst\_port)$. If the focus is on individual machines, the 3-tuple definition $(protocol, src\_ip, dst\_ip)$ is enough. For an image of the whole cluster, the classification unit may not even be a flow but the whole trace over a period of time. These definitions are of course not fixed, and they depend on the individual use case and on the other information we may have about the network topology and application deployment.

2.3 Network Traffic Capture

In order to analyse the network traffic between multiple machines, this traffic has to be somehow "saved". For this, traffic capture tools are used. The simplest capture tools are software based, such as *tshark*[3]. These may have timestamping resolutions in the order of microseconds, which for low bandwidth networks is not an issue. Other capture tools use specialised hardware. One such example is the Endace DAG card [26] that has a much better resolution of 4ns. Another example of hardware is the Exablaze ExaNIC card [27] which has a 250ps resolution.

[3] `https://www.wireshark.org/`

In a 10Gbit/s network, packets arrive at short intervals, so the timestamping resolution has to be low enough. Otherwise multiple packets will correspond to the same timestamp[4]. Time on wire for packets of various sizes in a 10Gbit/s network are shown in Table 2.1. So to capture a 64 byte packet, requires a maximum resolution of less than 50ns. Out of the mentioned tools, only the specialised hardware tools are suitable for capturing a 10Gbit/s network.

Packet Size (Bytes)	**Time on Wire (ns)**
64	51.2
512	409.6
1024	819.2
1512	1209.6

Table 2.1: Time on wire for packets of different sizes, assuming 10Gbit/s connection.

Capturing traffic is a task that puts pressure on the disks and on the operating system, as each packet has to be written to disk and an interrupt is raised every few packets. Hence, collecting traces on the same machine that generates the traffic may perturb the execution of the workload. To avoid this issue when using optical fibre, traffic can be captured externally provided that it does not interfere with the system being observed.

An approach that can achieve this is the use of optical taps. These are passive devices that work by splitting a portion of the light signal from the main fibre link and diverting it to a monitor port on another machine.

Another approach is port mirroring on switches that offer this capability. This is a more sophisticated process. An example of such a switch is the Exablaze ExaLINK Fusion [28], which has layer 1 switching capabilities, meaning it can replicate traffic from any port to any other port with very low latency introduced. This device is essentially a patch panel.

The standard file format for capture files is the *pcap* format [29]. This is a simple format that captures the bytes that make each packet and add timestamps to each packet. *Pcap* files can be parsed using CLI or GUI tools like *tshark*/*wireshark* or libraries like *dpkt*[5] or *pyshark*[6].

2.4 Machine Learning Classification

Classification is a fundamental problem that machine learning (ML) can efficiently solve. Methods range from feature-based classification to deep learning methods using complex neural networks.

Feature-based classification is a cornerstone of "traditional" machine learning. The goal of the classification algorithm is to learn a function $f : X \to Y$ where X is a vector of values called "features" and Y is a discrete set of class labels. This type of classification

[4]This is what happens with the TSval option in TCP. The resolution is 1ms so multiple packets end up having the same TSval. For the created datasets (see Section 4.4.2 and Appendix C) I chose to count how many packets have consecutive equal TSval's, assuming that that can be correlated with the burstiness of the traffic.

[5]`https://dpkt.readthedocs.io/en/latest/`

[6]`https://pypi.org/project/pyshark/`

relies on a crucial step: feature engineering. In this step, domain specific knowledge is used to extract relevant characteristics from raw data. These are chosen in a way that can capture the underlying patterns of the target classes. In network classification of flows, examples of features are packet length statistics or TCP flags. Examples of ML algorithms are decision trees, support vector machines, or naive Bayes.

Deep learning is a subfield of ML that is based on neural networks with multiple layers. Deep learning (DL) methods can learn hierarchical feature representations directly from raw data, skipping the feature engineering step. A typical architecture in DL used for tasks that involve 2 dimensional data (images) or 1 dimensional data (time series or sequence data) is a convolutional neural network (CNN). In the case of network traffic classification the CNN can take as input the raw captured bytes.

Chapter 3

Related work

3.1 Network Traffic Classification

Network traffic classification could initially be solved easily by looking at the used port numbers and mapping them to the corresponding protocols. However, as applications became more complex, and developers wanted to obfuscate their activity more, dynamic ports started to be used, or applications started "tunnelling" their traffic through standard ports (e.g. 80 for HTTP). Moore and Papagiannaki [30] identified in 2005 that using port-based identification could not provide an accuracy of more than 70%, confirmed by Madhukar *et al.* [31] on a separate dataset.

Deep Packet Inspection (DPI) appeared as a response to the evolution of the used port numbers. DPI classifies the payload of network traffic by computing specific signatures that can then be matched against known signatures of applications. This method achieved high accuracy for known applications as shown by Moore *et al.* [30], Fernandes *et al.* [32] and Hubballi *et al.* [33]. As encrypted traffic became the norm, DPI became harder, and decreased in efficiency.

Feature-based machine learning (ML) methods started being used to classify network flows, leveraging statistical features extracted at packet or flow level. Moore and Zuev [34] introduced classification using Bayesian techniques which achieved an accuracy of up to 96%. Williams *et al.* [35] evaluated multiple supervised ML algorithms on network classification that achieved up to 99% accuracy. These methods have been used successfully for different target classes and different types of traffic [36, 37, 38, 39].

Deep learning methods have also been used to classify network traffic. Wang [40] was among the first to use neural networks for traffic classification, showing how the method can be used to classify each TCP flow into different protocols by using the first 1000[1] bytes of TCP payload. Wang *et al.* [41, 42] used convolutional neural networks (2-dimensional and 1-dimensional) to classify malware traffic as well as VPN traffic. They used the first 784 bytes of each flow, both of all traffic layers and only of layer 7 (application) separately. Lopez *et al.* [43] improved the model by integrating spatial and temporal characteristics as well, using LSTM, a specific Recurrent Neural Network (RNN).

So far the objectives of network traffic classification have been partially limited by the availability of datasets and their scope. The most widely used public datasets and their classification targets are listed below in Table 3.1.

[1] This threshold of approximately 1000 bytes was introduced in [30].

Dataset Name	Year	Scope
Moore [34]	2005	End-User Applications (web, BitTorrent, SMTP etc.)
ISCX [44]	2016	End-User Applications (Facebook, Chrome, Skype) / VPN-nonVPN
MAWI [45]	2000 - ongoing	Data collected at an ISP, of various applications, labelled as malware or not.
UNSW [46]	2017	Traffic coming from different IoT devices
Mirage [47]	2019	Android applications
MobileGT [48]	2018	Mobile applications
UNIBS-2009 [49]	2009	End-User Applications (web, BitTorrent, Skype etc.)
CTU-13 [50]	2014	Botnet traffic

Table 3.1: Public datasets used in network classification research and their scope.

Other existing research uses private datasets that are still scoped mostly to consumer applications or malware classification. Voice over IP (VoIP) identification is well studied [51, 52, 37, 53]. Detecting VPN traffic is another wide spread use of network traffic classification [54, 42, 55].

3.2 Data Centre Workload Characteristics

The workloads used in data centres have distinct resource usage patterns and are classified based on their communication type, latency sensitivity, and bandwidth intensity. Commonly studied workloads in literature are presented in Table 3.2.

TCP remains the most common transport layer protocol in data centres. The characteristics of TCP traffic have been a hot topic in network research given that understanding them is essential for optimising performance, diagnosing issues, and designing better protocols. Benson *et al.* [60] and Alizadeh *et al.* [56] showed the bimodal distribution of flow sizes: above 90% of traffic is represented by *mice flows*, which are short flows carrying less than 10% of bytes. The rest of the flows are *elephant flows*, which carry more than 90% of bytes.

The burstiness of TCP traffic is another well known property of DCNs. Multiple studies showed an ON/OFF pattern: periods of intensity are followed by idle periods. Benson *et al.* [60] showed that packet inter-arrivals in DCNs follow a heavy tailed distribution. Kapoor *et al.* [64] also showed that packets often arrive in "trains" at sub-RTT intervals. Woodruff *et al.* [65] showed how burst patterns differ between applications. Burstiness is an important characteristic because intense bursts could overflow buffers and cause losses even if the average utilisation is low.

An obstacle in data centre research is the limited availability of public traffic datasets. Operators are reluctant to share actual traces due to privacy concerns. Typically, researchers will only publish aggregated statistics or graphs. Facebook [14], Google [66] and Alibaba [67] have published traces of resource usage for some of their clusters, but

Workload Type	Examples	Patterns and Requirements	Reference Studies
Web Services (online queries)	Web Front-Ends, Google Search	Latency sensitive, short flows, High fan-out	Google's cluster traces [56], *DCBench* [57], Ersoz *et al.* [58]
Batch Analytics and Big Data	MapReduce, Spark	Large flows sending GB of data, causes bursts when nodes synchronise, throughput intensive, burst prone	Facebook traces [14], Microsoft reports [59]
Distributed Storage and Key-Value Stores	NoSQL DBs, *memcached*, Redis	Mixed patterns: small control messages, but also bulk data transfers.	Benson *et al.* [60], YCSB [61]
High-Performance Computing	Simulations, Modelling (e.g. financial)	Compute intensive, low network activity, bandwidth needed for IPC	Jia et al. [57],
Machine Learning	Distributed training, large model inference	Bandwidth intensive (updates have to be moved quickly), latency sensitive at synchronisation points, burst prone	MLPerf suite [62], Zerwas et al. [63]

Table 3.2: Common data centre workload types, examples and their network characteristics. "Latency sensitive" workloads generate short flows and require low latency, while "throughput intensive" workloads generate large flows and emphasize aggregate bandwidth.

none include network packet traces. Because of this data scarcity, researchers have created tools and benchmarks, such as SWIM [68] and YCSB [61], which have been used before to mimic web services. Other publications use simulations to generate traffic (using ns-2 for example), citing prior measurements for validity.

3.3 Data Handling and Privacy Challenges

It is evident that a significant challenge of network traffic classification is the scarcity of public datasets. This is a consequence of existing regulations and of privacy concerns. The lack of public data hinders the reproducibility and comparison of existing studies. Training on private data may also affect the generalisability of methodologies and introduce bias. For specialised targets, like data centres, this is an ongoing issue, as identified by Roy *et al.* [14]. At the time of that publication, the whole research community was relying on data published by Microsoft [59], which did not generalise to data observed by Facebook.

Researchers who have access to real data cannot share the raw traffic captures due to regulations such as GDPR. Anonymising the data is challenging and may degrade data quality. Companies are reluctant to share data due to the possibility of losing their competitive advantage. Even when they share traces [67, 66], they only focus on particular clusters that already use open technologies.

Regardless of its many beneficial and legitimate uses, network traffic classification has a "dark side" that may raise ethical and privacy concerns. Research in this field can be used for surveillance and monitoring of individuals by ISPs or governments. These actors may have indirect access to personal data, such as browser activity, applications usages, and communication patterns. These can be used for user profiling with objectives, such as targeted advertisements or manipulation.

This project addresses these concerns. The resulting dataset is fully open and reproducible. The systematic approach taken for data collection is ethical as all of the trace generation is controlled by myself. The scope of the project is on intra-data centre workloads. The performed analysis is, in my opinion, harder to misuse by malicious actors, as DCs already have multiple layers of security, and getting access to internal traffic may be caused by bigger security issues.

Chapter 4

System Design and Methodology

This chapter will describe the experimental setup I used to run selected the workloads and capture their network traffic at 10Gbit/s. All experiments were run in a controlled test-bed that resembled a data centre (DC) environment, as presented in Section 4.1. Following the description of the used hardware and software, I will introduce the selected workloads that will be the subject of the executed experiments. The workloads represent different application classes that appear in data centre networks (DCNs). These are introduced and motivated in Section 4.2.

To make experiments reproducible and less prone to errors, I have developed an ad-hoc orchestration system that can trigger experimental runs and traffic collection. This is presented in Section 4.3.

The main deliverable of the experiments is the resulting dataset that describes the captured traffic for each application. Section 4.4 will present how raw captured traffic is processed to obtain a feature-based dataset that parametrises each TCP flow of an application.

4.1 Experimental Setup

4.1.1 Experimental Test-bed

My project's objective is to analyse and collect data from data centre level applications. To make the data I collect relevant, the experimental setup has to be similar to what is found in DCNs. As presented in Section 2.1, modern DCs can have highly complex network architectures, but generally they are characterised by high bandwidth and low-latency connections. Since I aim to look at applications' behaviour in their "purest" form with no interference, my test-bed is relatively simple in order to minimise influences of hardware.

For transparency and reproducibility, the exact hardware setup used is presented in Appendix A.

The test-bed is represented one rack with multiple identical servers, all connected directly to one network switch using optical fibre. Throughout this report I will refer to this set of machines as the test-bed machines.

Each test-bed machine has three network connections:

- An Ethernet connection that connects the machine to the internet, used also for SSH-ing into the machines.
- An Ethernet connection that connects the machine to a control infrastructure, used for the sys-admin.
- An optical fibre connection to a switch.

The optical fibre connection is a 10Gbit/s connection. The NIC used is an Intel Corporation Ethernet 10G 2P X520 Adapter. The other two connections are irrelevant for these experiments, as all workloads use only the optical fibre connection.

All machines are connected to the same switch, making it equivalent to a "leaf" switch in a DCN architecture. Consequently, the available bisection bandwidth is 10Gbit/s. I have measured the latency and bandwidth of the connections, results are presented in Table 4.1. These are rough measurements and should be taken as estimates of the orders of magnitude. They do highlight the high bandwidth and low-latency properties. Interestingly, the measured bandwidth is not the full line rate of 10GBit/s. I suspect that *iperf*[1] does not consider the mandatory inter-packet gap and the preamble bytes (see Section 2.2.1 for definitions) when computing the measured bandwidth.

Property	**Value**
Bandwidth	9.40Gbit/s
Latency (RTT)	$\approx 50\mu$s

Table 4.1: Network Characteristics

4.1.2 Network Capture Infrastructure

To characterise the network traffic of an application, the said traffic has to be captured and stored in order to be analysed later. Section 2.3 enumerated available tools that can capture network traffic highlighting their strengths and weaknesses. In order to obtain an accurate and reliable dataset, I have chosen to use specialised hardware for the traffic captures. I used the ExaNIC X10 [27] for captures, which is an ultra low-latency 10Gbit/s network device used in domains such as high frequency trading (HFT). Each of the two ports on the ExaNIC can capture at line rate, 10Gbit/s. To capture traffic in both directions in a connection, both ports have to be used.

ExaNIC X10's are installed on a separate set of machines. These machines will be referred to as the capture machines. As mentioned in the section above, the test-bed machines are connected only to each other through the switch, so to capture the traffic on another machine, it has to be mirrored somehow. One option as suggested in Section 2.3 would be optical taps. But these suffer from potential losses of signal strength. The chosen method for traffic mirroring is the use of a high performance network switch, the ExaLINK Fusion [28].

The ExaLINK Fusion is a layer 1 switch tailored for low-latency applications. The delay introduced is around 5ns. As a layer 1 switch it can be used for patching, tapping or replication. In my setup, this switch sits between the test-bed machines and the leaf switch. All connections are passed to the leaf switch, but are also tapped. The outputs of the taps are connected using optical fibre to the ExaNIC's. Essentially, each ExaNIC

[1] `https://iperf.fr/`

captures traffic to and from a particular test-bed machine. All fibre connections in the setup are of similar lengths, so connections are symmetric from this point of view.

An overall image of the capture system can be seen in Figure 4.1.

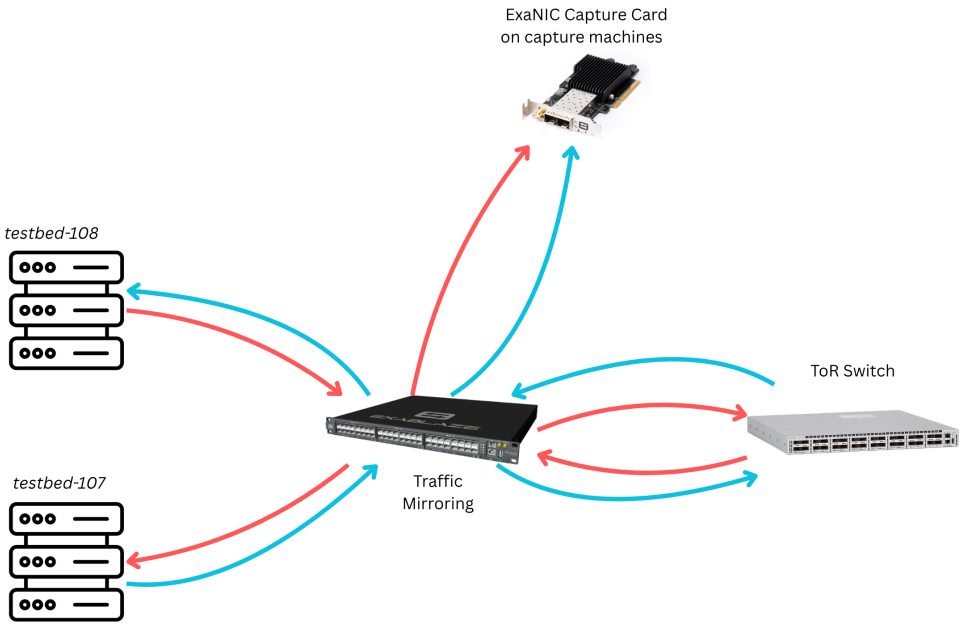

Figure 4.1: Simplified view of the experimental setup including the capture system. Red arrows represent traffic going from *testbed-108* to *testbed-107* and blue arrows represent traffic going the reverse way.

Above I mentioned that the important setup detail of the capture machines is their storage capability. This is because the ExaNIC's buffer writes to handle large bursts, but they may still end up dropping packets if the disk writes are not fast enough. The ExaNIC can capture at a rate of 10Gbit/s on both ports, for a total maximum data generation rate of 20Gbit/s. But in practice this rate is bounded by the storage system. The write speeds of the disks used on each capture machine can be seen in Table A.3. We see that the disks may be a bottleneck, as at most they can write at 3Gbit/s which would correspond to about the same maximum capture rate.

The ExaNIC may not be able to capture at full line rate on both ports due to limitations in the disk properties, but this is not a problem for the experiments I ran, as I will show in Section 4.2. All experiments "generate" traffic at rates lower than 3Gbit/s.

Another important aspect of the capture system is the timestamping capability, as explained in Section 2.3. The timestamps for the captured packets are applied in hardware on the ExaNIC's. The card provides a resolution of 250ps, but the actual resolution used in the capture files is nanosecond resolution, which is sufficient to precisely preserve the true ordering of the packets in the low-latency captured system.

Previous work [25, 69] evaluated the used hardware, showing that timestamping is precise when capturing on both ports, with maximum deviations of 750ps and mean deviation of 125ps. The mean deviation is thus less than a clock cycle while the maximum deviation is 4 clock cycles on the capture machine. This makes it suitable for my experiments.

Multiple ExaNIC's can have their clocks synchronised using different methods: synchronising with their hosts (provided the hosts are synchronised with each other), PTP synchronisation or PPS synchronisation. This way, two ExaNIC's can be used to capture

Listing 4.1: Summary of a capture using ExaNIC

```
SW Received: 24611 packets ( 0.000 MP/s )
SW Wrote: 24611 packets ( 0.000 MP/s )
Lost RX/WR:     0 packets ( 0.000 MP/s )
Dropped:        0 packets ( 0.000 MP/s )
SW Overflows:     0 times   ( 0.000 /s   )
```

traffic from the same connection (one captures one direction, the other one the other direction). This could be used to mitigate the issue coming from the disk write speeds. In my setup I chose to use a GPS-derived PPS signal, that is connected to all ExaNICs ensuring their synchronisation.

Capturing traffic using the ExaNIC's is straightforward using the tool-chain open-sourced by Exablaze [70]. Configuration of the NICs is done only once when the machines are turned on for the first time. A capture is started using a simple command, `exact-capture`, which uses a given number of CPUs that are kept busy during the run. The capture ends when the process is killed and at the end a summary is printed, as shown in Listing 4.1. Using this output I estimated the data production rate of running workloads that are captured. Given that the rate of dropped packets is outputted as well, I was able to empirically verify if the disks on the capture machines can handle the workloads.

The ExaNIC's store the captures in a novel format called *expcap*. This is an extension to the *pcap* format presented in Section 2.3. The structure of a captured packet is presented in Figure 4.2. The major addition of *expcap* is the possibility of storing picosecond timestamps. Besides that, the file also contains metadata related to the capture hardware, which in this case is not needed. Since current libraries and programs used for parsing of traffic captures do not support the *expcap* format, I converted all captures to *pcap* files with nanosecond resolution, using the software provided by Exablaze.

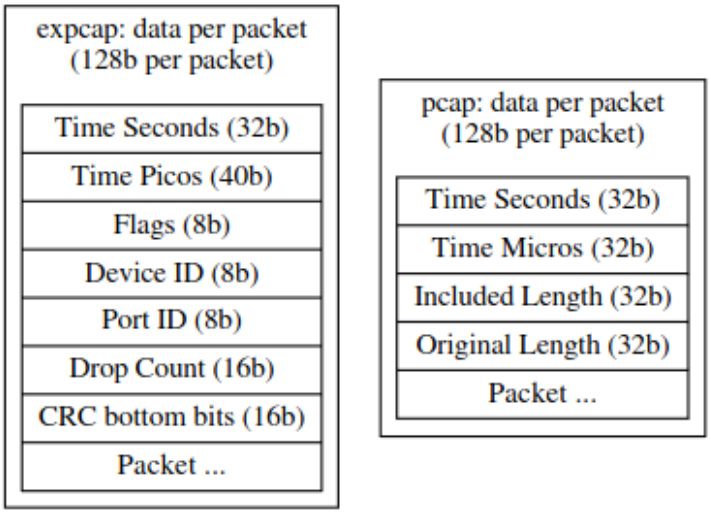

Figure 4.2: Comparison of the *expcap* and *pcap* file formats. Figure taken from [69].

4.2 Workload Selection

In order to validate my analysis as being relevant for data centres, the workloads that are the subject of the experiments were selected accordingly. As presented in Section 3.2, some workloads that keep appearing in papers focusing on high performance networking, specific to DCs, are web servers, caching servers, databases, distributed AI training, MapReduce, etc.

DCPerf [71] is a benchmark suite outsourced by Meta designed to represent real-world hyper-scale cloud-applications in DCs. Meta uses this suite to test hardware or to conduct performance projections. The suite contains 6 benchmarks:

- **TaoBench**: a workload stressing an in-memory key-value store. Probably relevant to Meta's TAO (The Associations and Objects) system used in their social graph infrastructure [72].
- **MediaWiki**: a web serving workload, using HHVM, which is representative for their deployment of Facebook.
- **FeedSim**: an object aggregation and page ranking application.
- **SparkBench**: a data analytics workload using Apache Spark.
- **DjangoBench**: a web serving workload using Cassandra and Django, which is representative for their deployment of Instagram.
- **VideoTranscodeBenchmark**: a video processing workload using *ffmpeg*.

Out of these benchmarks, VideoTranscodeBenchmark was not suitable for my project as it has no network component. SparkBench was also unsuitable as it required a targeted hardware setup using multiple SSDs on multiple nodes.

With these observations made, I decided to choose DjangoBench, FeedSim, and TaoBench. I chose these workloads as they represent distinct classes of data centre applications. While their heterogeneous nature is expected to result in high classification separability, this will serve as validation for my framework's ability to capture and process fundamentally different traffic patterns. A description of each of these three is given below. Since all benchmarks involve network communication between at least two machines, each machine in the setup plays a different "role". The detailed list of parameters for each of the chosen benchmarks is present in Appendix B.

All three of the chosen benchmarks have been slightly modified to accommodate my setup. This is because the benchmarks target very high-performance machines with hundreds of cores and networks that can handle more than 10Gbit/s. The sections below present the changed benchmarks. Most changes are related to the parameters of the chosen workloads.

For each workload I also provide a number of empirical observations of their behaviours. The most important observation is the "data production" rate, i.e. how much traffic is generated per second. This is done in order to confirm the choice of the capture machine hardware.

4.2.1 TaoBench

TaoBench is a client-server benchmark designed to stress an in-memory key-value cache system. The server is based on *memcached*[2] and the client is based on *memtier_benchmark*[3]. Both are patched in order to closely resemble real-world traffic. In order to mimic real scenarios, the item sizes are pulled from a predefined distribution and the items can also be compressed.

The benchmark measures the number of queries-per-second (QPS) and the cache hit ratio. It has two phases: a long warm up phase (at least 20 minutes) which should fill up the cache, and a test phase which does the measurements. By default, the server uses all available CPUs, and a server instance is spawned for each NUMA node.

TaoBench requires at least two machines: a client and a server. The main parameters for the workload are the runtime and the number of workers.

I have empirically observed that a run of this benchmark with one client produces network traffic at an average rate of 0.3GB/s, which is below the write speed of the disks used on the capture machines (see Section 4.1.2). Other observations I made are that the command running on the client machine spawns the set number of threads as separate processes. Then each thread spawns the set number of clients and each client opens a connection with the server. The connection stays open throughout the whole phase of the benchmark, meaning that each TCP flow lasts for as long as the benchmark does. The source port for the connection is randomly assigned to the clients. The CPU utilisation on the server machine is always near 100% during the whole benchmark run.

For all my benchmark runs with this application I have set the server to use 16GB of memory. Only one server instance is spawned each run. I used one client machine with six client threads and 100 connections per thread.

4.2.2 DjangoBench

DjangoBench is a benchmark that measures the performance of a web application using the Django Python framework[4]. It sets up a multi-tiered environment and uses a load-generation tool to simulate traffic. The application itself emulates a social media platform (based on Instagram).

For the web application it uses the Django framework and uWSGI[5]. The server machine uses memcached for caching, while as a database it uses Apache Cassandra[6], a NoSQL data base (DB). The load generator used is *siege*[7], which is a well-known HTTP load generator that can simulate multiple users accessing one or more URLs.

The database contains a list of users, a list of feeds, and a list of notifications. At the beginning of the workload, the web server populates the database with mock entries that simulate a realistic dataset. The web application exposes several APIs that simulate a common social media design, such as fetching a user's feed or marking some comment as seen.

[2] `https://www.memcached.org/`
[3] `https://github.com/RedisLabs/memtier_benchmark`
[4] `https://www.djangoproject.com/`
[5] `https://uwsgi-docs.readthedocs.io/en/latest/`
[6] `https://cassandra.apache.org/_/index.html`
[7] `https://github.com/JoeDog/siege`

The client is given a predefined weighted list of URLs. Meaning the accessed URLs are uniformly accessed. The clients simulate real world scenarios in terms of the access patterns (e.g. most common action is fetching a feed). *Siege* can also simulate a "client think time". Meaning that after receiving a response it waits a bit before it sends another request.

DjangoBench requires at least three machines that have one of three roles: client, server or database. The important parameters are the runtime of the benchmark and the number of client and server workers, which influences how many queries are be done (each client can do multiple queries).

During all tests I ran the three components (DB, server, client) on separate machines. Only the traffic of the server is captured. This is feasible because I empirically observed that the workload generates traffic data at a rate of about 1.5MB/s. This is low, so the disks on the capture machines are not a bottleneck. My other observation was that *siege* opens separate connections for all requests, each from a different ephemeral port. This means that each TCP flow contains precisely only one HTTP request. Another important thing is that the web server opens several connections to the DB, only some being used for data transfers. Others are used to send heartbeat messages. All the connections are open throughout the whole workload run.

In the experimental runs I varied the benchmark duration. Early runs alternated the inclusion of the client "think-time" parameter, but it turned out to be irrelevant, given that each "client" issues only a single request before closing the connection.

4.2.3 Feedsim

Feedsim is a benchmark designed to simulate aggregation and ranking workloads used in recommendation systems. It is a client-server benchmark that can be extended to more complex architectures. The application is entirely written in C++.

The base of Feedsim is OLDIsim [73], a framework open-sourced by Google that supports benchmarks that emulate Online Data-Intensive workloads. Generally OLDI workloads are "user-facing workloads that mine massive data sets across many servers", and include web searching and social networking, as well as page ranking. OLDIsim simulates real-world behaviours, such as I/O waiting, large memory accesses, and cache trashing (both ICache and DCache).

Feedsim requests and responses are serialised using Thrift [74] and compressed using ZSTD [75]. Feedsim is more synthetic compared to the other benchmarks. Clients send random data in large chunks in order to stress the server during decompression and deserialisation. Responses are structured and contain list of objects that mimic a real-world ranked feed. However, the responses contain pseudo-random data. Upon receiving them, the clients essentially ignore the contents.

The main objective of this benchmark is to find the maximum QPS that the server can support given an upper limit on the latency. It does this by using a binary-search like approach. But the client can also be run using a fixed number of QPS. Similar to TaoBench, it has a warm up phase and a test phase.

Feedsim requires at least two machines: one server and at least one client. The only parameter accepted by the server is the number of instances it spawns. The intended design is to run one instance per NUMA node, so in my case I always ran one instance.

Given the design of this benchmark, it is a highly CPU intensive workload. I observed this by tracking the amount of traffic data generated, which is less than a MB per second. Similar to TaoBench, each client opens a connection, and keeps it open throughout the whole benchmark run. This results in a low number of TCP flows with long durations (as long as the benchmark).

During the experimental runs I altered the requested number of QPS. The maximum QPS *feedsim* can achieve with my setup is 10, which was obtained by running the benchmark in search mode. For comparison, an example run from DCPerf's documentation shows an achieved QPS for a machine with 380 cores and 2TB of memory. Besides this I also altered the duration of the benchmark.

4.3 Orchestration

4.3.1 Design Objectives

One of the main objectives of my project is the repeatable creation of datasets that represent the properties of the chosen data centre applications. This consists a core novelty as no other such workflow has been published.

Extracting the traffic data is straightforward: a capture is started on the capture machine, a benchmark is run on the test-bed machines, and the capture is stopped when the benchmark ends. Labelling this data is the challenge and one of the key contributions that I bring over state-of-the-art. Existing datasets that focus on network traffic classification label the data with their intended goals in mind: a dataset created for a malware classification has malware/no malware labels, a dataset created for mobile application classification is labelled with the application names, and so on. I developed a process that associates network traffic with more than just the application name. My main objective was to be able to associate the capture data with any other data available about the workload: application name, version of libraries, system specifications, performance metrics, parameters used, etc.

Capturing network traffic can be manually done by coordinating the capture commands on all machines. This is tedious and does not scale, especially if a workload has multiple phases that have to be captured separately.

The advantages of having experiment runs fully automated, as well as collecting all the data about the systems being tested, is that the whole process can be repeated easily and results can be replicated (proven in Section 5.1). This is an important aspect of computer systems research [76], as publishing artefacts that can be used to replicate the results is encouraged by institutions such as ACM [77].

Given the above description, I summarised my objectives for this section that describe how the orchestration infrastructure was built. This includes running the workloads and capturing the network traffic, as well as capturing all the other data that may be needed. The objectives are as follows:

1. Build a scalable infrastructure that can reliably run all chosen workloads and collect their network traffic.
2. Ensure experimental runs can be repeated (shown in Section 5.1) and reproduced.
3. Collect multiple sets of metadata about each workload run.

4. Ensure that collected metadata can be associated with the captured traffic.
5. Keep the design extensible: think for the future, allow more workloads that can run on different machines with different parameters, allow for more metadata to be collected if needed.

To ensure that the design is extensible, in the following sections I will not directly refer to the chosen benchmarks, in order to maintain the generality of the system. Examples of the experimental runs and their associated configs are given from my used experimental scenarios.

4.3.2 Multi-Machine Task Automation

Before diving into how each component of the benchmarks is run and how data is collected, I will first present how I automated the runs of these commands on the set of machines. For the purpose of this section, the specific programs that are running on the test-bed machines are irrelevant.

I assume the following static model (this information does not change between runs) for a series of test-bed machines $a_1, a_2, \ldots a_n$, where $a_1 \ldots a_n$ are host names:

1. Each machine a_i can be associated with a capture machine c_i. Machine c_i has an ExaNIC that receives the mirrored traffic going to and from machine a_i.
2. Each capture machine c_i has two capture interfaces (one for Tx traffic, one for Rx traffic), and a capture location (a path on the capture machine where the data is stored).
3. Each machine a_i has an associated IP address in the test-bed network.
4. Each machine a_i and each machine c_j have an associated user that can SSH without a password and can run `sudo` commands passwordless.
5. There is a predefined set of workloads that can be run.
6. Each workload has a set of predefined roles that machines should take.
7. Each role of a workload has a set of predefined parameters that it can accept. Furthermore, it is known if each role can have one or more instances (for example, you can have multiple client machines but only one server).
8. Each workload component provides a "start command" generator. This is an executable that takes a dictionary of arguments as input and outputs a command that starts the component on the test-bed machine.
9. Each workload also has a predefined set of phases, each phase completion is signalled in a log file[8]. Phases can be captured or not.

All of this information is stored in a set of config files in JSON format. Below, two sample entries are shown:

Listing 4.2: Example configuration for a test-bed machine

```
1
2     "caelum-108.cl.cam.ac.uk": {
3         "ip": "10.0.0.2",
```

[8]If a phase does not have an explicit completion signal, the capture for that phase stops when workload stops

```
        "user": "abc123",
        "capture_machine": "nf-server13.nf.cl.cam.ac.uk"
    }
```

Listing 4.3: Example configuration for a capture machine

```
    "nf-server13.nf.cl.cam.ac.uk": {
        "user": "abc123",
        "capture_directory": "/local/scratch2/collection_runs",
        "capture_interface1": "eth2",
        "capture_interface2": "eth3"
    }
```

Listing 4.4: Example configuration for a workload

```
    "django": {
        "roles": [
        {
            "name": "db",
            "arguments": ["bind_ip", "duration", "grace_period"],
            "unique": True,
            "command_generator_location": "./scripts/django/
               generate_db_cmd.sh"
        },
        {
            "name": "server",
            "arguments": ["db_addr", "server_workers", "duration"
               , "grace_period"],
            "unique": True,
            "command_generator_location":
            "./scripts/django/generate_server_cmd.sh"
        },
        {
            "name": "client",
            "arguments": ["server_ip", "client_think", "
               client_workers",
            "duration"],
            "unique": False,
            "command_generator_location":
            "./scripts/django/generate_client_cmd.sh"
        }
        ],
        "phases": [
        {
            "name": "warm-up",
            "completion-signal": "Warmup Done",
            "capture": True
        },
        {
            "name": "benchperiod",
            "completion-signal": None,
            "capture": True
        }
```

```
        ]
    }
```

With this in mind, an experimental run can be defined by the following information:

1. Each run uses a named workload.
2. Each role of the workload has an associated machine that runs it.
3. Each role has the list of arguments that should be passed to the command that starts it.
4. For each role it can be mentioned if its traffic should be captured or not.
5. The different programs corresponding to the different roles are started in the order in which they appear in the config files. Each role can have a defined event that has to happen before the next component starts. This can be either a string appearing in a log file or a defined time period.

An example configuration for an experimental run can be seen below:

Listing 4.5: Example configuration for an experimental run

```
{
    "workload_name": "django",
    "roles": [
        "db": {
            "machine": "caelum-108.cl.cam.ac.uk",
            "arguments": {
                "bind_ip": "10.0.0.9",
                "duration": "420",
                "grace_period": "300"
            },
            "start_after": "10",
            "capture": False
        },
        "server": {
            "machine": "caelum-403.cl.cam.ac.uk",
            "arguments": {
                "db_addr": "10.0.0.9",
                "duration": "420",
                "grace_period": "300"
            },
            "ready_signal": "Server setup done",
            "capture": True
        },
        "client": {
            "machine": "caelum-404.cl.cam.ac.uk",
            "arguments": {
                "server_ip": "10.0.0.5",
                "client_tink": "1"
            }
        }
    ],
}
```

The tooling I built begins by reading the main experiment config, and when required also parses configuration files for the test-bed and capture machines to extract their connection details. Taken together, these files give a clear step-by-step blueprint for everything that needs to happen during an experimental run:

1. Captures start on all machines that have to capture data.
2. The different roles of the workload start in the predefined order, waiting for the declared periods or signals.
3. When the current workload phase finishes, the capture stops, and a new capture is started.

For each role, the tool-chain generates the command that is run on the machine to start the component. It copies that command to the target machine and it executes it. The output is captured in a log file that is queried for the potential "signals" that signify the end of a phase or the end of the setup for the said component. It is the responsibility of the benchmark to leave some buffering times between phases so the new captures can start.

At the end of an experimental run, the tool-chain converts the *expcap* files to *pcap* files, it archives, and compresses them. Then the files are sent to a new directory in a shared location. Besides the *pcap* files, the tool-chain also copies the resulting files from the benchmarking software (from the test-bed machines) and the configuration of the run, in order to associate the resulting capture files with the particular experiment.

From an implementation perspective, I opted for a lean architecture. All machines are accessed via SSH, and Python along with Bash scripts handle configuration parsing and command execution. This approach yields a minimalist, yet robust system without complexity and overhead of heavyweight orchestration platforms such as Kubernetes[9].

What the tool-chain currently does not automate is the generation of the config files. These have to be set manually. The tools do check the "correctness" of the provided configs, meaning that a missing or undefined field gets flagged. The installation of the benchmarks is not automated either, but this can be extended by adding other fields in the workload configuration that define the installation process (commands to run, files to copy, etc.).

4.3.3 Integrated Local Workload Execution and Data Collection

In the previous section I talked about the infrastructure that can run the distributed workloads and capture their network traffic. The way workloads are run and metadata is collected on the test-bed machines has been ignored. This section will briefly present the local tool-chain on the test-bed machines that is able to run the chosen workloads and collect system data about them. The local tool-chain can be generalised and used to run other workloads as well.

As mentioned in Section 4.2, all my chosen workloads come from an open-source benchmark suite, DCPerf. All three workloads I chose are fairly "heterogeneous", each require different applications and different setup. Even within the same workload, different setup may be needed for each role. To facilitate the different benchmarks that can be run or added, DCPerf comes with a wrapper tool, *benchpress*. This tool essentially provides a

[9] `https://kubernetes.io/docs/home/`

command line interface that can install and run benchmarks. The CLI of *benchpress* is detailed in Appendix B.

Before running a workload, *benchpress* collects a comprehensive description of the current system. The collected information and the tools used to collect this information are presented in Table 4.2.

Information Class	**Examples**	**Tools used to extract the information**
CPU Information	CPU model, CPU flags, cache sizes, CPU frequency	‘lscpu‘
OS Information	OS Name, OS Version, Kernel Version, System Type, kernel command line	‘uname -a‘, ‘/etc/os-release‘, ‘/proc/cmdline‘
Hardware Information	Information about all hardware components (manufacturer, model, available slots, what hardware is using the slots etc.)	‘dmidecode‘ and ‘lshw‘
Memory Information	Total Memory, Free Memory, Huge Page info	‘/proc/meminfo‘
System Packages	A list of all installed packages and their versions	‘dpkg-query‘
Kernel Parameters	All kernel parameters including system tuning and configurations	‘sysctl -a‘

Table 4.2: Information collected by *benchpress*.

All the information presented above is then stored in a JSON file that has the following high-level format shown in Listing 4.6. This resulting JSON file is generated for each workload and saved locally. A JSON file summarising the workload run is also saved. I can associate both of these files with the captured traffic using the tool-chain presented in Section 4.3.

Listing 4.6: Structure of the system-specs file generated by *benchpress*

```
{
    "cpu_topology": { ... },
    "os_kernel": { ... },
    "kernel_cmdline": [ ... ],
    "dmidecode": { ... },
    "sys_packages": [ ... ],
    "kernel_params": { ... },
    "memory": { ... },
    "hardware": { ... },
    "os-release": { ... }
}
```

4.4 Data Processing Pipeline

The previous section presented the automation infrastructure that orchestrates the workloads and collects their produced network traffic and associated metadata. Using the above tool-chain, one experimental run results in a directory containing the compressed *pcap* files and the summaries of each component. The next step in the constructed pipeline is to process the *pcap* files in a way that results in a dataset that can ultimately be fed into a machine learning model.

I wanted the resulting datasets to be extensive, in order to support multiple processing methods and use cases. As mentioned, my intention for the resulting datasets is to not have only one label, but rather to have flexibility in the classification tasks. Hence, the tools presented in this section can create different datasets with different sets of labels based on a given configuration.

As mentioned in Sections 2.2.1 and 3.1, existing research in network classification of applications uses bidirectional flows as the units that have to be classified, so I followed that convention with my datasets. Other network classification methods use deep learning to classify raw traffic samples, so to facilitate those tasks I also created PyTorch datasets containing raw traffic.

The objectives of this section can be summarised as follows:

1. Split resulting *pcap* files in individual flows.

2. Process the flows in order to extract a comprehensive set of features and raw data.

3. Allow the flexible creation of datasets depending on the intended classification task.

4.4.1 Demultiplexing TCP Traffic

The scope of this project is application classification, as discussed in Section 2.2.1, the appropriate classification unit is the network flow defined as the 5-tuple (*protocol*, *source_ip*, *source_port*, *destination_ip*, *destination_port*). A correct split of a *pcap* file must result in individual *pcap* files that contain only traffic between the (*src_ip*, *src_port*) and (*dst_ip*, *dst_port*). A straightforward solution is to parse the *pcap* file several times, isolating the packets for a single flow on each pass. This can be implemented easily with a tool like *tshark*. The issue is that this results in N reads of the file if the file contains N flows, which is slow for large *pcap* files.

A better solution is to parse the *pcap* file only once. As packets are encountered, flows identifications are collected. If a flow with that identification is open, the new packet is added to that flow, otherwise a new flow is opened. Flows are be closed when there is a TCP connection tear-down, or after a certain time out period. This solution is faster, but requires more memory to hold all current open flows.

The above approach is taken by the tool *demuxer* developed for a previous project [78]. An overview of the way *demuxer* works is shown in Figure 4.3 I chose to use this tool as part of my data processing pipeline. My main contribution for *demuxer* is adding support for nanosecond *pcap* files. A run of this tool on an existing *pcap* file results in multiple *pcap* files that correspond to separate flows.

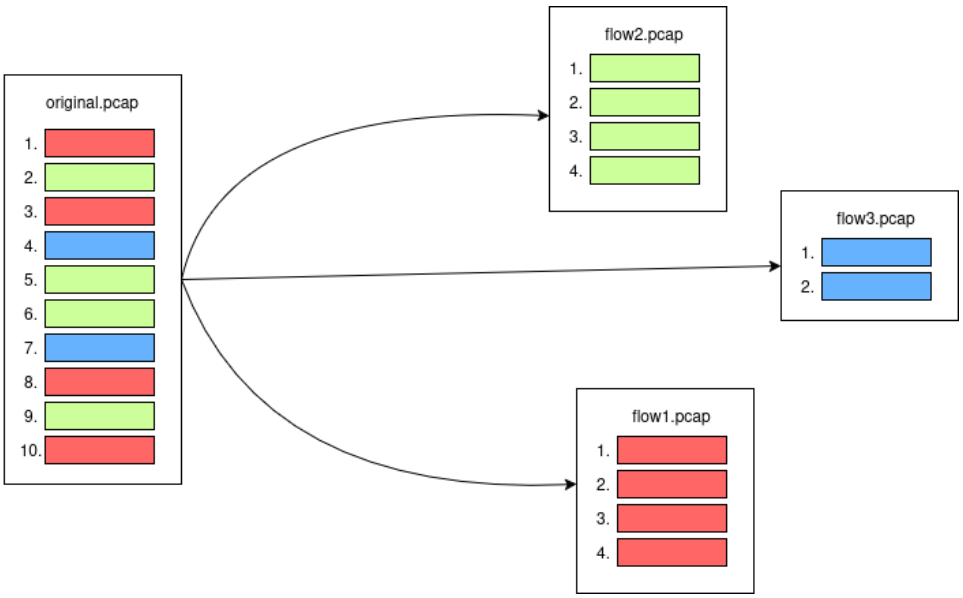

Figure 4.3: The operation performed by *demuxer*. Each small rectangle represents a network packet and each colour represents a separate flow.

4.4.2 A Comprehensive Feature Set For TCP Flows

In order to allow for complex analysis on the captured traffic, it is necessary to extract as much information as possible. Extracting a set of data features makes the information comprehensible for both humans and ML classifiers. I decided to create a set of tools that parses the *pcap* files, computes a set of features, and store them in CSV format. My implementation uses the Python library `dpkt`[10].

I computed a set of approximately 200 features. Many of them were based on the feature set published by Moore *et al.* [34]. Before diving into the computed features I present the following definitions that are used in the feature definitions:

Idle Period : A period of at least 2 seconds with no traffic in either direction.

Interactive Period : Non-idle periods.

Bulk Transfer : A period within interactive periods with at least 3 consecutive packets containing data in one direction and no packets containing data from the other direction.

Microburst Period : A period of at least 4 packets in one direction with an inter-packet-gap of at most 1241ns (Based on the definition of Woodruff *et al.* [69, 65]).

Large Transfer : A data transfer that spans multiple consecutive packets. Identified by at least 3 consecutive packets carrying data in one direction, with all except the last one carrying the maximum IP payload size.

Most features are computed in three variations:

1. All packets regardless of direction are considered.
2. Only packets sent in client→server direction are considered.
3. Only packets sent in server→client direction are considered.

[10] `https://dpkt.readthedocs.io/en/latest/`

The full list of features and their definition can be found in Appendix C. Below is the list of feature categories:

1. **Basic Packet Statistics**: Statistics related to the packet sizes (wire size, IP size).
2. **Transfer Mode Statistics**: Statistics related to the transfer modes present in the flow (idle, bulk, interactive).
3. **Timing Statistics**: Statistics related to inter arrival time of packets.
4. **Segment Statistics**: Statistics related to segment sizes.
5. **TCP Flags Statistics**: Counts of various TCP flags.
6. **TCP Handshake Statistics**: Whether the TCP handshake was captured or not and its direction.
7. **Ordering Statistics**: Statistics related to the ordering of packets (out-of-order packets) and retransmissions.
8. **TCP Window Statistics**: Statistics related to the TCP window sizes advertised during the connection.
9. **Timestamp Statistics**: Statistics related to the measured RTT between the two machines.
10. **Microburst Statistics**: Statistics related to the observed microburst periods.
11. **Large Transfer Statistics**: Statistics related to captured large transfers between the two machines.

As mentioned in the previous section, besides this feature set, my tool-chain also extracts raw traffic from the parsed flows. Based on existing research in deep learning for network traffic classification [42, 40, 43], I decided to extract the first 800 bytes of a flow in the following variations:

1. First 800 bytes of traffic including all traffic layers.
2. First 800 bytes of traffic including only application layer (TCP payload).
3. First 800 bytes of all layers traffic, excluding the TCP handshake.
4. First 800 bytes of application layer, excluding TCP handshake.

The features are stored in CSV format, while the raw traffic samples are stored in `npy` format, a binary format used by the NumPy[11] library.

4.4.3 Creating Datasets

Running the tool described in the previous section on a set of *pcap* flows representing individual flows, results in a CSV file containing the list of flows and corresponding features. To create a dataset that can be used for classification, multiple of these CSV files corresponding to different classes have to be concatenated.

To facilitate the dynamic creation of datasets where different flows can have different labels depending of the application, I built a program that takes a configuration specifying a list of CSV files and the labels that should be applied to the flows in each CSV file. The tool creates a new column for all data points called "label" and applies the corresponding

[11] `https://numpy.org/doc/stable/`

label. The result is a larger CSV file that can then be used in a ML classifier. The `npy` files containing raw traffic samples are combined into a PyTorch[12] dataset that can be used for deep learning applications.

I also decided to convert the dataset to Parquet format used by Apache Spark[13]. This is a format often used in large scale data processing. The whole data processing pipeline starting from the original *pcap* file and ending with the resulting dataset can be seen in Figure 4.4.

[12] `https://docs.pytorch.org/docs/stable/index.html`
[13] `https://spark.apache.org/`

Figure 4.4: Overview of the data processing pipeline.

Chapter 5

Evaluation

This chapter starts by assessing the repeatability capabilities of the developed framework. Then it will present the evaluation of existing traffic classification methods on the datasets created using the methodology presented in Chapter 4. Two datasets are used in the evaluation: an application-identification dataset that labels flows with the application that produced them, and a performance-estimation dataset for one of the workloads that labels flows with the achieved QPS during the run.

Developing new classification methods is beyond the scope of the projects. Hence why this chapter focuses on the insights that the existing classification methods can provide for the collected data. The presented results should be treated as a "proof-of-concept" for what can be done with network data. The chosen methods for classification are taken from existing work as presented in Section 3.1.

5.1 Assessing Repeatability

As mentioned in Section 4.3.1, a main objective when developing the orchestration framework was to ensure repeatability and reproducibility of all experiments and measurements. The ACM [77] defines repeatability as the capability to obtain the same measurement using the same equipment, by the same team. Essentially, the measurements are constant between different runs. Reproducibility is achieved if a different team can obtain the same measurements using the same equipment. Reproducibility cannot be assessed, as a second team would need to be involved. In this section I prove, using a simple example, that my framework produces repeatable measurements.

To isolate the effects of my framework, I chose a simple test to run. I decided to do 5 runs of a group of 10 measurements of round trip time (RTT) using *ping*[1]. I wanted to show that the 5 runs are equivalent, and there is no unexpected statistical difference between them. For each measurement I report the minimum, maximum, average and standard deviation for the RTT.

The distributions of each metric for each run is shown in Figure 5.1.

From the figure it can already be seen that for all statistics the distributions overlap. There is no visible difference between them. To formally assess this though, I am using

[1]This also implied "extending" my framework to run a different workload, which in a way proves the extensibility.

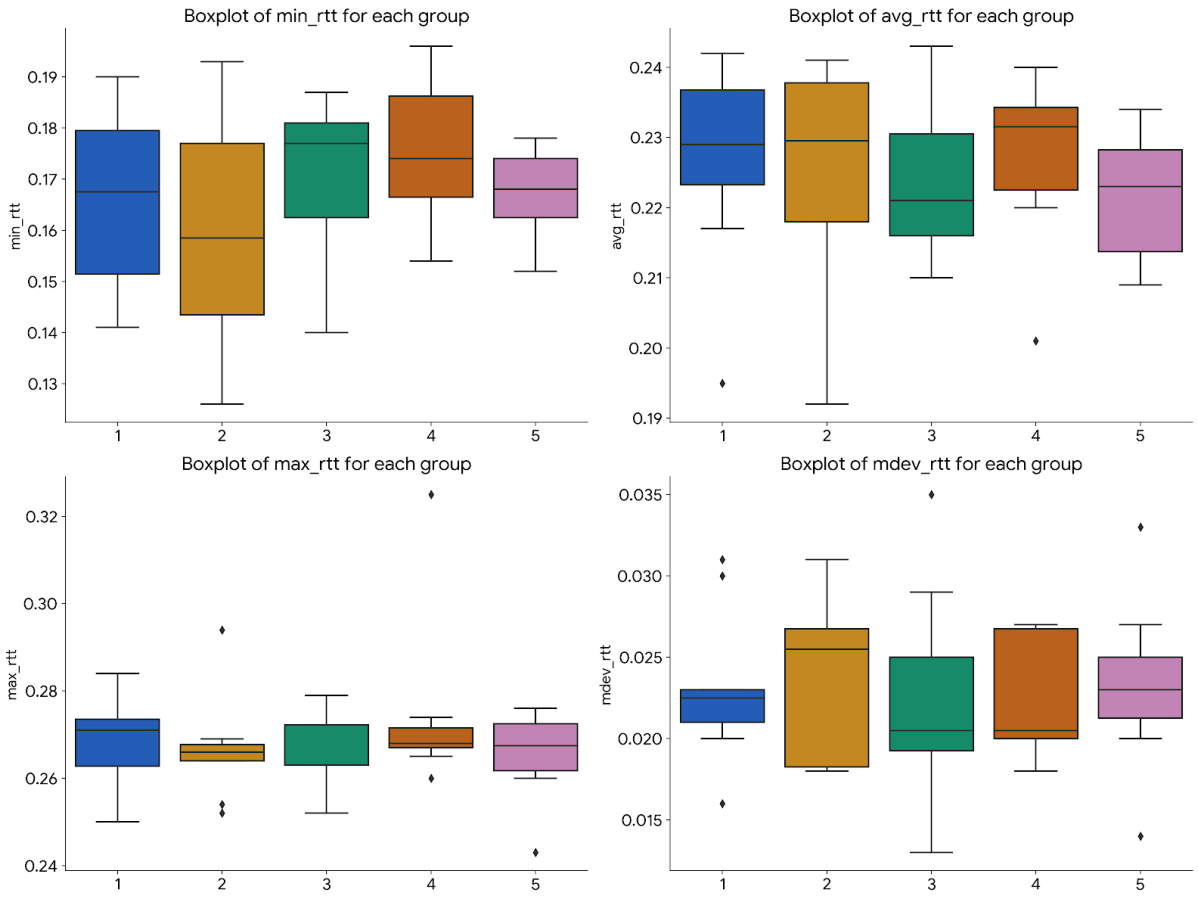

Figure 5.1: Boxplots showing the distributions of each RTT statistic between the 5 runs. Each run has 10 measurements, each box represents the distribution of that statistic over the 10 measurements.

the ANOVA F-test[2]. This test answers the question: "Are the observed differences in average RTT (or the other statistics) between the 5 runs meaningful or are they just noise?". The null hypothesis is that the means of each statistic for all 5 groups are equal. Using the F-test, I calculated the p-values. If the p-value is greater than 0.05, the null hypothesis cannot be rejected. The p-values for each statistic are found in Table 5.1. Given that the null hypothesis cannot be rejected for any of the measured statistics, I conclude that there is no statistical difference between the 5 runs.

It is important to understand that all measurements have variability [20]. The graph shows some variation between different measurements but this is due to *ping*'s intrinsic behaviour; it is not introduced by my orchestration framework. *ping* is well-studied [20, 79], and in this case it does not exhibit any unexpected variance. I can conclude that my framework does not produce any statistically significant differences between multiple runs.

5.2 Summary of Datasets

The datasets used were produced by running each of the selected workloads a number of times with different parameters. The selected workload for the performance-estimation

[2] `https://en.wikipedia.org/wiki/F-test`

Statistic	p-value
Minimum RTT	0.25
Maximum RTT	0.59
Average RTT	0.78
Standard Deviation	0.96

Table 5.1: p-values computed using ANOVA F-test for each measured statistic. A p-value greater than 0.05 means that there is no significant statistical difference between the 5 runs.

dataset is *feedsim*. The application-identification dataset is presented in Table 5.2 and the performance-estimation dataset is presented in Table 5.3. For each of them the target classes are presented with the number of samples and the number of times the workload was run to produce the samples.

The *feedsim* workload can achieve a maximum of 10 QPS running on the test-bed system, as mentioned in Section 4.2.3, hence the chosen targets for the performance-estimation dataset. The three QPS values were fixed before the run. In the application-identification dataset, *django_client* contains the flows that represent the client-server communication, while *django_db* represents the DB-server communication.

Class Name	Number of Samples	Number of Runs
django_client	9822388	16
django_db	1217	16
feedsim	2187	57
tao	7201	12

Table 5.2: Overview of the application-identification dataset.

Class Name	Number of Samples	Number of Runs
feedsim_5qps	675	14
feedim_8qps	669	14
feedsim_10qps	845	29

Table 5.3: Overview of the performance-estimation dataset.

A noticeable property of the first dataset is how unbalanced it is. The *django* workload generates a considerably larger number of flows compared to the other two. This is expected given its behaviour described in Section 4.2.2. Each HTTP request constitutes a separate TCP flow. Besides, the observed pattern is similar to what the literature describes in terms of proportion of different workload types (see Section 2.1). In data centres more than 90% of the flows are short, generated by web service applications, while the rest of the flows are long flows generated by batch workloads (in this case *feedsim*) and key-value store workloads (in this case *django_db* and *tao*).

5.3 Feature-Based Classification

The first classification method I focused on is feature-based classification. This relies on the dataset containing approximately 200 features for each flow. I used a set of supervised learning algorithms: logistic regression, support vector machines (SVM) and random forests. I am using the `scikit-learn`[3] framework for this task.

As explained above, the application-identification dataset is heavily unbalanced. To counter that, I have randomly sampled a subset of *django_client* flows (83628, which represent 90% of the total dataset). Due to a low number of samples, I ignored the *django_db* class. Upon closer examination only about 2-3 flows per run are relevant, the others are either heartbeats or TCP handshakes followed immediately by a TCP tear down.

For the classification tasks I used a train-test split of 80-20. Each algorithm was run 5 times with different random seeds.

5.3.1 Feature Importance Analysis

Given the relatively high number of features, one of the questions I aimed to answer was: "Are all features needed?". Showing that only a subset of features can be used for accurate classification is a valuable insight that can influence the design of future systems. To check what features are "important" I used proven statistical methods. My choice was to use correlation-based feature selection [80]. The idea behind this method is that a good feature subset contains features highly correlated with the target classes, but uncorrelated with each other.

To find the correlation between pairs of features I used the Pearson correlation coefficient[4]. To select discriminating features I used the mutual information (MI) measure[5].

The correlation matrices for each of the two datasets can be seen in Figures 5.2 and 5.3. The list of all features and their explanations can be found in Appendix C.

[3] `https://scikit-learn.org/stable/index.html`

[4] `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`

[5] `https://en.wikipedia.org/wiki/Mutual_information`

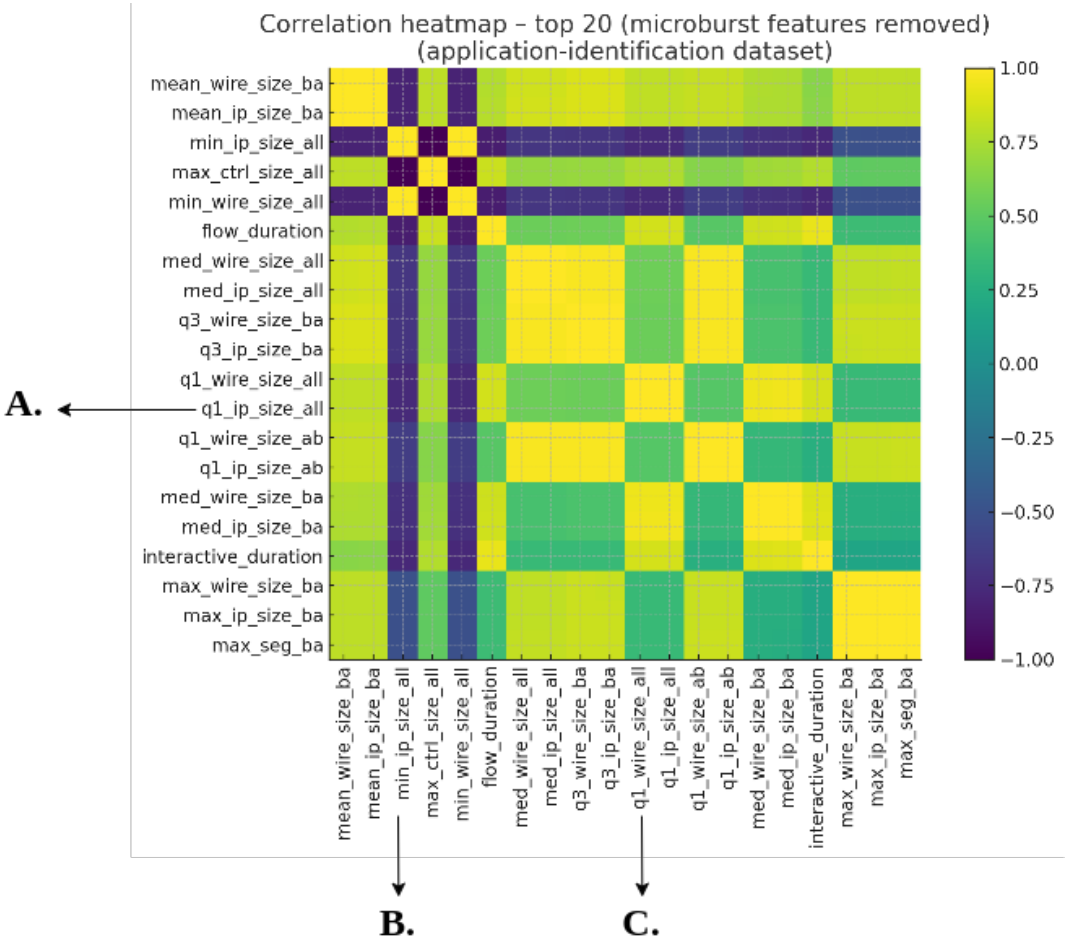

Figure 5.2: Correlation matrix showing the top 20 most inter-correlated features for the application-identification dataset. Feature A is the first quartile value for the observed IP packet sizes, feature B is minimum observed IP packet size and feature C is the q1 value for the observed packet sizes (including all layers). We see that A has a strong negative correlation with B which means that when A grows, B goes down. In contrast, A has a strong positive correlation with C. These make sense when we think of the actual meaning of features: it is normal that the q1 values for IP packet sizes and total wire sizes are correlated, given that the difference between the IP packet size and total packet size is constant (except a few rare packets). In this case, we don't need all of these three features for the classification, as essentially A, B and C measure the same thing.

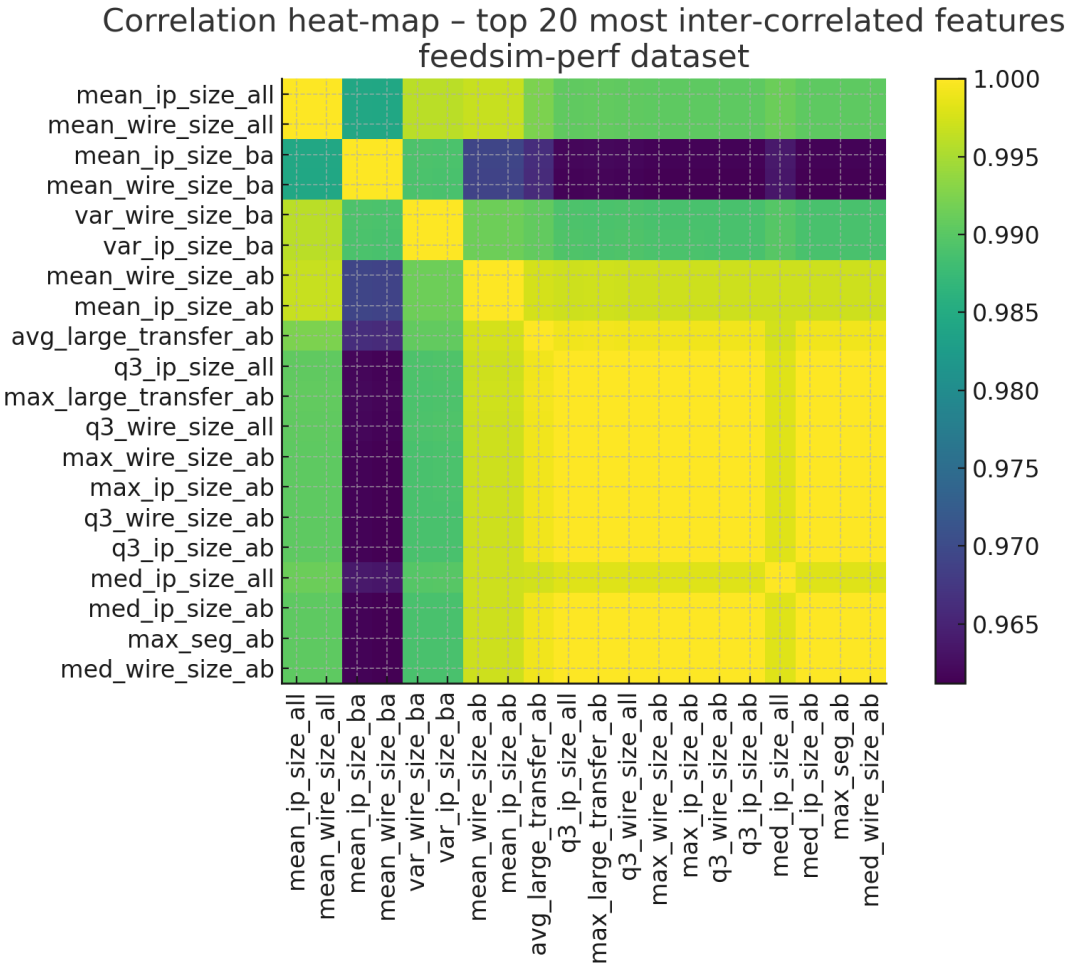

Figure 5.3: Correlation matrix showing the top 20 most inter-correlated features for the performance-estimation dataset. It should be noted the different scale compared to previous figure. Before the scale was -1 to 1, now it is 0.96-1, meaning that features of this dataset are more correlated that the previous one. This is expected, as all features in this figure are related to packet sizes in some way (IP size, full packet size), and *feedsim* is expected to send the same queries and responses regardless of the target QPS, the only difference should be how often they are sent.

In Figures 5.2 and 5.3 we can see that for both datasets that most of the top inter-correlated features are size related. This is expected since if packet size increases, all other size related features will follow. Other strongly correlated groups are features related to throughput, features related to timing, and features related to flow duration.

For all features I computed the mutual information (MI) score to see how "discriminating" they are. In Figures 5.4 and 5.5 I plotted the top 10 and bottom 10 features by mutual information scores for the two datasets. A high mutual information score signifies that knowing that feature reduces the uncertainty about the target class.

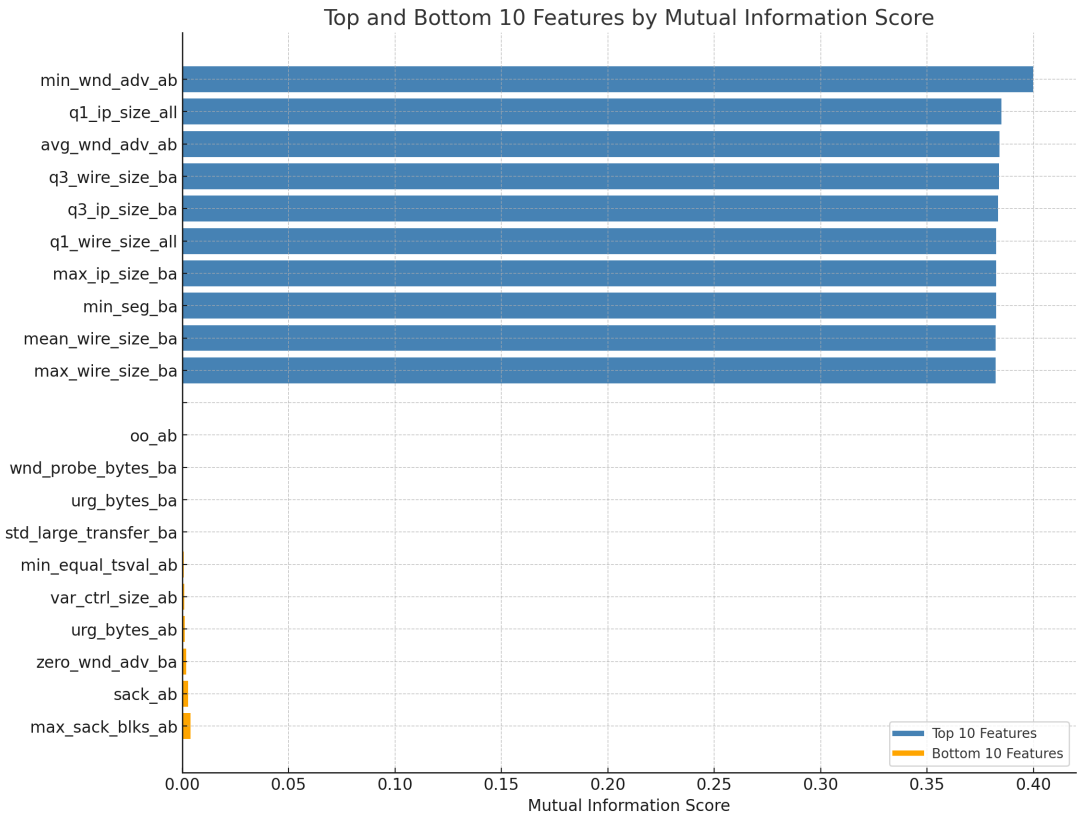

Figure 5.4: Top 10 and bottom 10 features by mutual information score for the application-identification dataset. The reason why the top 10 features are those is unclear at this point. The bottom ones are more obvious: most of them have constant or near constant values between all classes (e.g. out of order packets which is constantly 0), so knowing their value does not reveal anything about the target of the classification.

Looking at the top 10 features by mutual information score for the application-identification dataset we see that most of them are also strongly inter-correlated (see Figure 5.2). For example q1 value for IP packet size (`q1_ip_size_all`) is among the top with q1 value for total packet size (`q1_wire_size_all`). These two are perfectly correlated, so using the correlation-based feature selection methodology, only one of them should be kept.

The correlation matrix of the final set of features can be seen in Figures 5.6 and 5.7. The features in these two matrices are the ones used for classification.

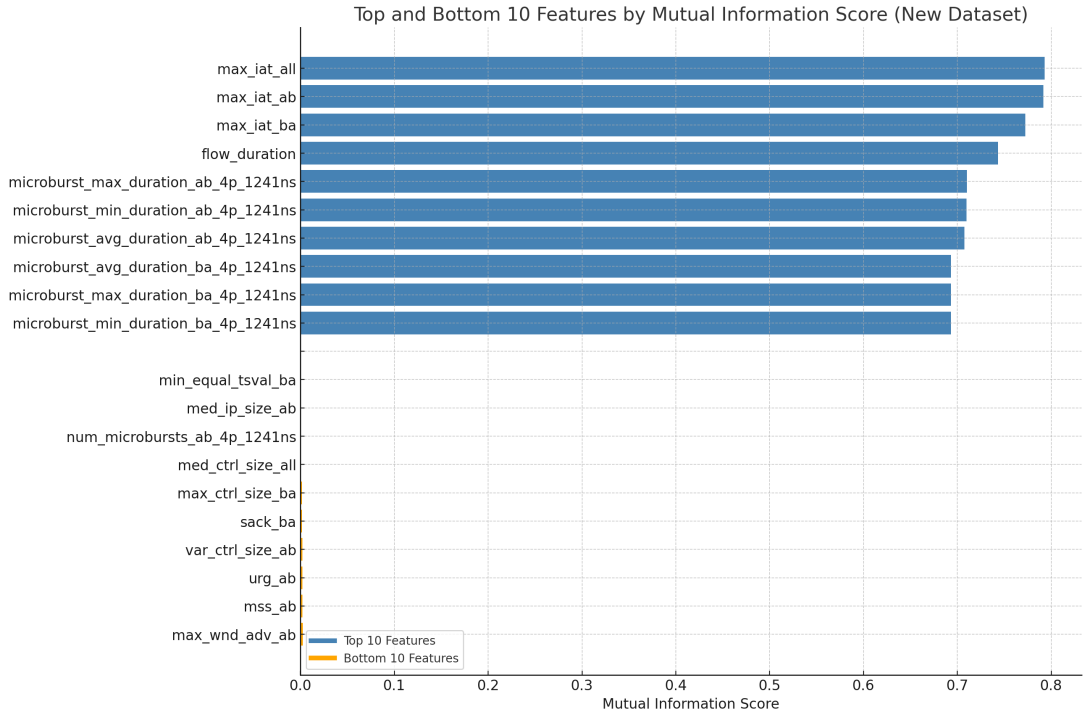

Figure 5.5: Top 10 and bottom 10 features by mutual information score for the performance-identification dataset. Compared to previous figure, now it is more clear why the top 10 features are timing related and microburst related. As mentioned in Figure 5.3, between the different the runs of *feedsim* are timing related. Essentially, higher QPS means burstier traffic.

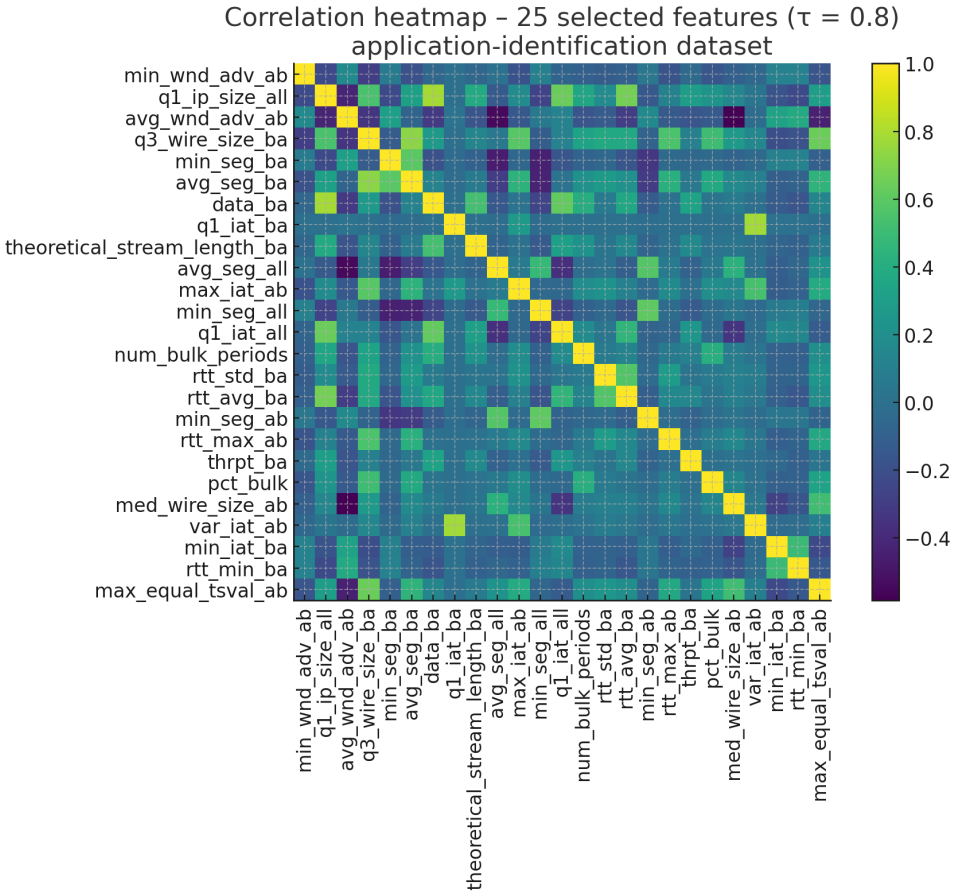

Figure 5.6: Correlation matrix for the chosen features. This set is less correlated and all features in the set have high mutual information scores. We see for example that `q1_ip_size_all` is present, `q1_wire_size_all` is not, which was also predicted by the previous informal analysis. The chosen features are more "diverse": there are features related to packet sizes (`q1_ip_size_all`), timing (`rtt_max_ab` which is the maximum RTT measured on the server), transfer rates (`thrpt_ba`, which is the overall throughput measured for data transferred by the client) and more.

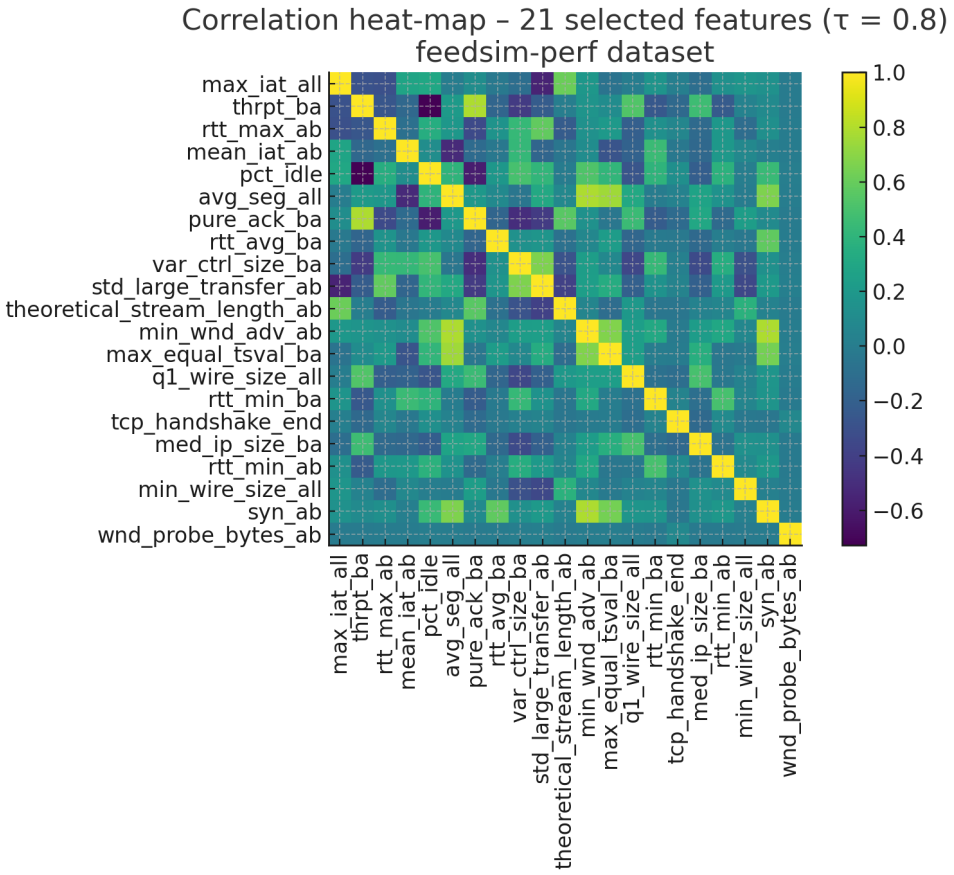

Figure 5.7: Correlation matrix for a subset of chosen features, for the performance-estimation dataset. As with the previous dataset, they are more diverse compared to the ones that showed up as the top ones by mutual information score.

5.3.2 Supervised Classification

Using the selected features I trained the supervised learning algorithms (random forests, SVM and logistic regression). The results can be seen in Table 5.4.

Classification Algorithm	**Application-Identification Average Accuracy**	**Feedsim Performance-Estimation Average Accuracy**
Random Forests	100%	83%
SVM	100%	58%
Logistic Regression	100%	52%

Table 5.4: Classification accuracy using the supervised learning.

For the application-identification datasets, all three models achieve 100% accuracy on all runs. For the performance-estimation dataset, I plotted the average accuracy over the 5 runs in Figure 5.8.

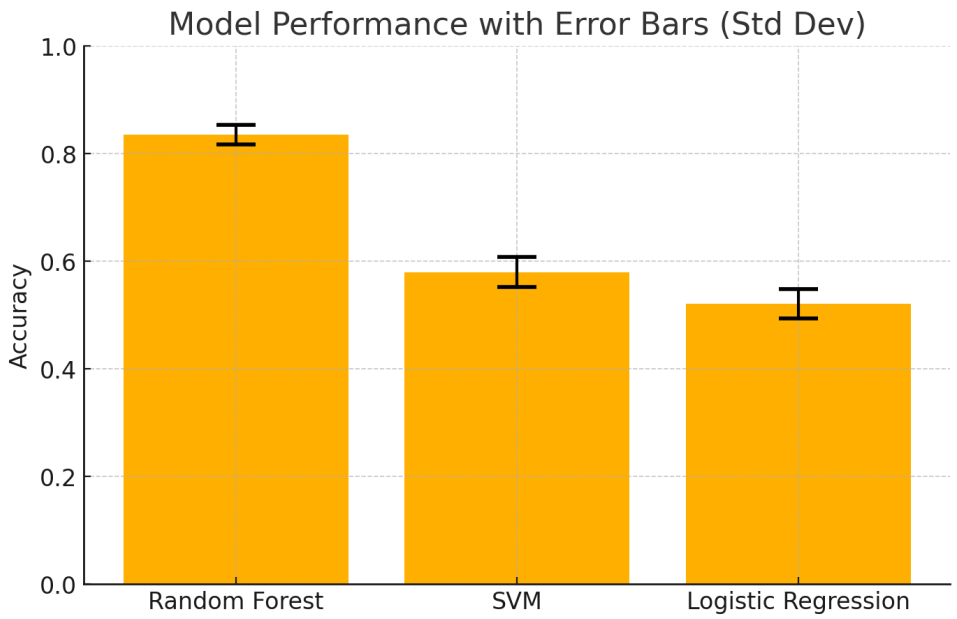

Figure 5.8: Average accuracy for the three tested ML classifiers on the performance-estimation dataset over 5 runs. Error bars are plotted as well.

We see that for the application-identification dataset, all algorithms are able to perfectly learn the patterns of each application. For the performance-estimation data however, only random forests has a good accuracy. It is strange that even logistic regression which is a simplistic algorithm is able to identify all flows. To understand, I also looked at the distributions of values within each class for the selected features.

The distributions show that some features are nearly perfect discriminants. Meaning that a particular range of values indicates one class with a high certitude. I plotted the distributions of two of these features below in figures 5.9 and 5.10. For contrast, I also plotted the distribution of a "bad" feature, that has a very low mutual information score in Figure 5.11.

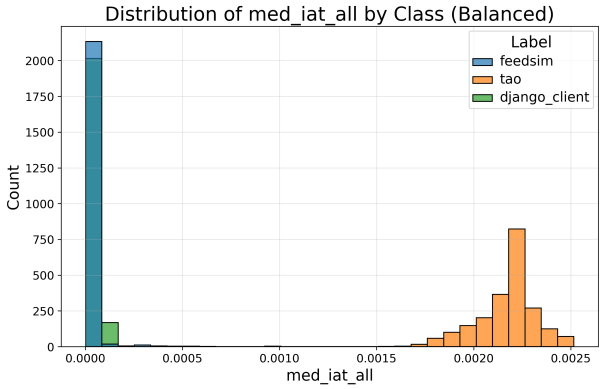

Figure 5.9: Distribution of the median inter-arrival time feature. The distributions for *tao* and *django_client* overlap almost perfectly while the distribution for *feedsim* is completely different. This means that this feature can perfectly separate *feedsim* from the other two workloads.

Figure 5.10: Distribution of the q1 (first-quartile) IP packet size. *django_client* and *tao* overlap briefly, but *feedsim* is clearly different and separated.

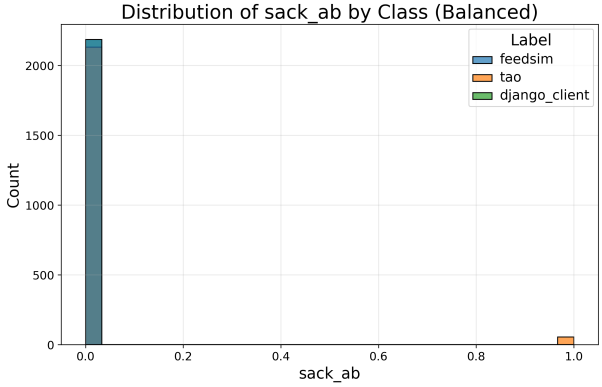

Figure 5.11: Distribution of number of packets with SACK blocks attached. This is an unhelpful feature, as we see all three workloads do not send any SACK blocks. Only *tao* sends SACK blocks, but that happens rarely, and it might be due to other factors unrelated to the application itself, as SACK is used to recover lost packets.

Looking at the distributions of the two "good" features, we can see that they are completely different for the three classes. Even if we only consider these two features, the classification can be done with high accuracy. So the explanation for the achieved accuracy is that the workloads are very heterogeneous and generate very specific traffic

patterns.

5.4 Deep Learning Classification

One of the issues with feature-based classification, as presented in the previous section, is that it can only classify completed flows. To have the full picture needed for classification, flows have to have been completed to compute all features. This is not practical for some scenarios where the classification has to be done in real-time. An alternative is to use deep learning (DL) classification that only needs the raw packet contents as inputs. With this method, flows can be classified as soon as the target number of bytes is captured.

To serve as the input, I collected the first 800 bytes of each flow (based on results from Wei *et al.* [41]) with the following constraints:

- All TCP layers.
- All TCP layers without the TCP handshake.
- Only application layer (TCP payload).
- Only application layer without the TCP handshake.

For the neural network architecture I used a one-dimensional convolutional neural network (1D-CNN). The exact architecture was chosen empirically. This can be seen below in Figure 5.12.

The CNN was trained for 30 epochs without fine tuning hyper-parameters. All training was done using my laptop's GPU. The results of the classification task for each dataset can be seen in Table 5.5. I ran the model 5 times with different random seeds.

Captured Traffic	**Application-Identification**	**Feedsim Performance-Estimation**
All layers	99.92%	33%
All layers ignoring TCP handshake	100%	95%
Application Layer	99.9%	81%
Application Layer ignoring TCP handshake	100%	85%

Table 5.5: Classification accuracy using the 1D-CNN.

As before, for the application-identification dataset, there is no variance between the different runs. For the performance-estimation dataset I plotted the average accuracy and standard deviation in Figure 5.13.

We observe that the classifier perfectly identifies the individual applications regardless of the criteria used. I conjecture that because each application uses a different transport level protocol, the neural network is able to learn the pattern specific to each protocol. For the performance-estimation dataset, we observe that to get high accuracy, the TCP handshake has to be ignored. My hypothesis is that this happens because the TCP handshake is the same regardless of the number of QPS for the runs. Taking all layers into account the model is able to identify the performance metric with 95% accuracy.

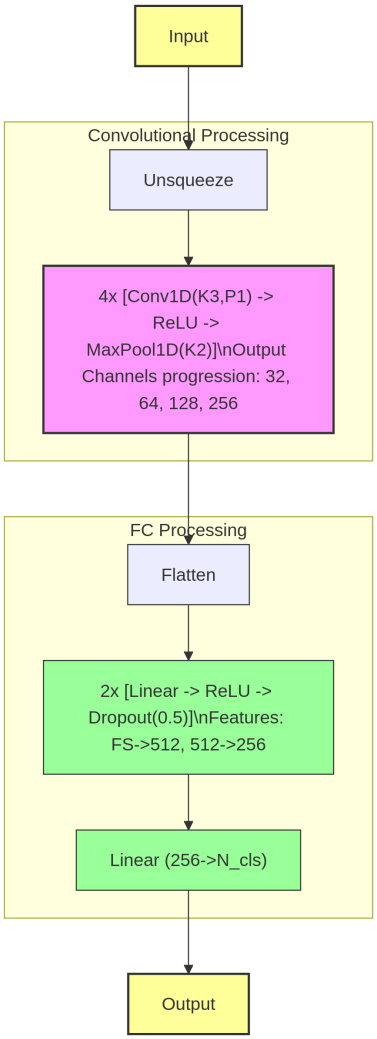

Figure 5.12: Overview of used CNN architecture. There are 4 1-dimensional convolutional layers, each followed by a ReLU activation and max-pooling operation. The result is flattened and passed through 3 fully connected layers that use ReLU and drop out for regularisation.

A drawback of deep learning classification is that it is harder to understand how the classification works. In previous section we've seen that with feature based classification we can precisely understand which features are discriminating between our classes. With the deep learning approach we can at most identify which subset of bytes in the input contribute to the decision.

5.5 Discussion

This section evaluated existing machine learning traffic classification methods on the created datasets using the methodology from Chapter 4. The evaluated algorithms, both supervised ML and deep learning, can achieve 100% accuracy when it comes to identifying the applications. This may seem surprising but at a closer inspection, we see that some features are essentially "fingerprints" of the workloads, meaning that the distributions of the feature values do not overlap between the target classes. This was an expected result as mentioned in Section 4.2.

Indeed, looking at the meaning of features, the differences are clear. For example, one of

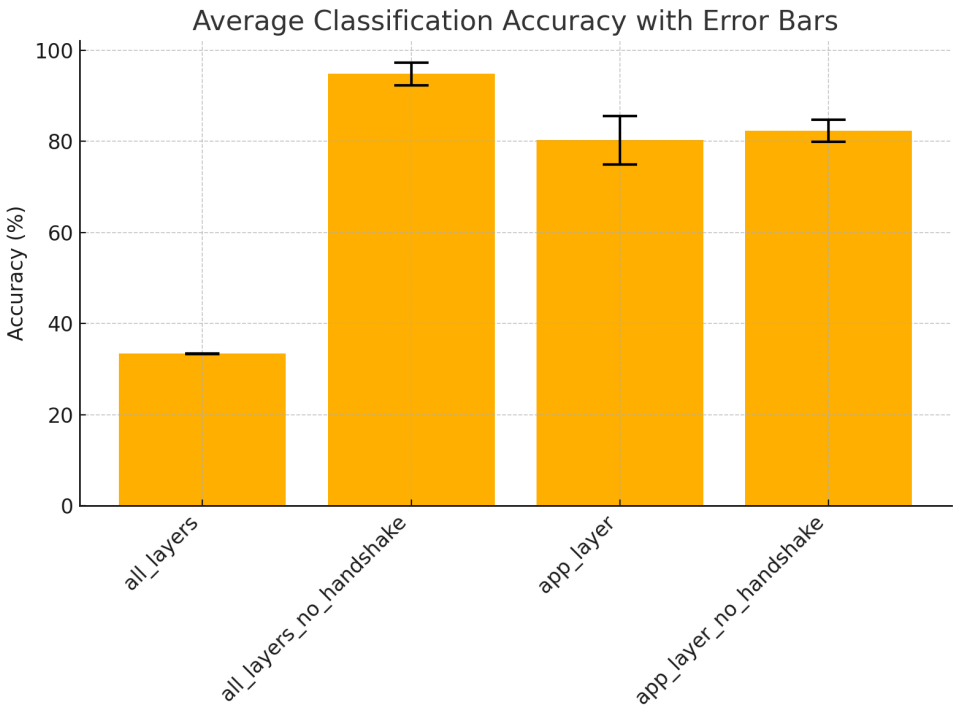

Figure 5.13: Average accuracy for each scenario on the performance-estimation dataset. Error bars are plotted.

the most discriminative features is the number of bytes sent from the client to the server. From the description of the workloads, it was evident that there might be limited overlap between the three. Same goes for flow duration, and all features that are correlated with flow duration.

For the performance-estimation dataset, the situation is slightly different. The classical machine learning algorithms failed to achieve good accuracies. The issue here is that probably the chosen set of features are targeted more towards identification of different workloads, not towards discriminating between different runs of the same workload. The discriminating features between different performance regimes are mostly related to timings (e.g. inter packet arrival times). However, the deep learning method proved efficient in this case achieving an accuracy of 95%. As mentioned, given the nature of DL methods, it is difficult to understand what is the underlying reason for the good classification accuracy.

The shortcoming of this analysis is potentially the lack of "interesting" data. Given that the workloads are so different, was it even necessary to train an ML model to classify them? I believe this issue comes from the chosen workloads, as initially their differences were not obvious. The data is reflective of their observed behaviours, so the methodology was not wrong in neither the data collection nor the data analysis stages. The root cause was the chosen setup in terms of workloads and their parameters. However, it should be mentioned that the analysis in this chapter clearly showed how different the workloads are, so even though the results may not seem exciting, they are correct and the evaluation achieves its goal: it correctly identifies running applications based on their network traffic characteristics.

5.6 Project Summary

This project's objective was the development of a network traffic capture and analysis framework that allows for repeatable dataset creation. I believe this was achieved and the developed framework can reliably run selected workloads, extract information from their network traffic, and perform an analysis of their network activity. The framework is readily extensible to new workloads, opening numerous avenues for continuations of the project. The ability to repeat the same tests multiple times is demonstrated in Section 5.1.

The collected traffic is processed to obtain comprehensive datasets. A novel approach to the dataset creation is that the labelling is not static. During every workload run, the tool-chain captures more than just network traffic: it records comprehensive hardware and system level metrics as well. Consequently, traffic captures can be linked to information that goes well beyond a simple application label. We can see what the kernel parameters were, or what hardware was used. Thus, all experiments can be reproduced, and furthermore this allows for deeper analysis on the captured traffic. All this collected metadata can ultimately be used both for labelling, and as features for classification or analysis.

One of the identified research gaps was the lack of a data centre targeted network traffic dataset. This project addressed this gap by creating the dataset using three data centre specific workloads. A major benefit of the presented methodology is that all datasets can be released openly: since the experiments are conducted in a DC-style test-bed, no confidential information is at stake. Besides being the first DC centric dataset, it is also one of the only public datasets including all capture traces.

The created dataset is used to train ML classifiers that can accurately identify the different applications, as seen in Sections 5.3 and 5.4. While identifying the application itself was straightforward, the breakthrough was the novel approach of classifying an application's performance solely through its network traffic. This is an unexplored avenue for network traffic classification which could be used for critical applications in data centres. Knowing how an application performs can tell us about how the hardware is doing (a hot CPU throttles, reducing performance), or how loaded a certain cluster is. This information can help scheduling in data centres, which in turn can help reduce the energy consumption.

Chapter 6

Conclusion

Modern data centre operators struggle to tune, secure, and ultimately understand their networks because they lack clear visibility into the traffic their applications generate. Public packet-capture datasets are rare, due to confidentiality and privacy concerns, and the few that exist neither reflect DC-specific workloads nor provide enough context to reproduce the original experiments. This gap leaves researchers without a reliable foundation on which to build or test classification tools based on network traffic.

I believe the presented work addresses this gap, firstly because it offers a rigorous and repeatable methodology for creating network traffic datasets. Secondly, it produces the first reproducible open dataset that includes full packet captures of data centre workloads. Thirdly, it explores a novel capability of network traffic classification: performance-estimation of the running workloads. These outcomes also fulfil the success criteria of the project as defined in Chapter 1.

The results reveal that separating the broad classes of DC applications, introduced in Section 3.2, is remarkably straightforward: their traffic signatures are so distinct that even simple classifiers can tell them apart. The magnitude of these differences was unforeseen when the workloads were selected. A welcome surprise was the classifiers' ability to separate runs of the same workloads with measured performance as the sole difference. This result underlines that network traffic analysis is still an open topic: future efforts should broaden the scope beyond application identification.

Traffic classification in data centres should reach beyond application identification. My proof-of-concept shows that, using nothing more than raw network traces, we can infer key performance metrics. Because operators already possess rich, fine-grained telemetry, this broader view is feasible. Leveraging the full spectrum of available data opens the door to foundation models trained to understand, optimise, and ultimately run the data centres.

6.1 Future Work

The main extension to the work would be the additional analysis of more homogeneous workloads (e.g. can we distinguish two web servers using different frameworks?). This is feasible given the extensibility of the network capture and data processing framework.

I think it is important to extend this research beyond the use of network traffic. All available data and metadata should be accounted for. The presented framework can be

extended to collect more data about workloads. This is an exciting direction which can lead to more optimised data centres. For example, considering resource usages can lead to better scheduling from the utilisation point-of-view (POV). This in turn leads to lower energy usages, which results in lower pollution levels.

A deep analysis of DC workloads may also lead to better DCN topologies. Currently DCNs use mostly Clos, but certain applications may not need that kind of design. So a good understanding of workloads and the ability to predict their behaviour can result in specialised topologies in specific clusters.

Besides performance optimisations, another direction can be intrusion detection in DCs. With so many tenants in a DC that have dynamic behaviours, it is hard for operators to detect if a user is misbehaving. Network traffic classification has already been used to detect malware running on end-user applications. But this should be extended to DCs where we may have bot farms, cryptocurrency miners, or even malicious actors targeting the DC itself.

All of the above look at the problem from the DC operator POV. However, it is important to also understand what "threats" might appear. If the operator can identify the DCN topology, what stops a tenant with access to multiple servers from detecting the topology based on the networking behaviour? When creating new DC models, we must anticipate and guard against the ways malicious actors might misuse the available information.

Putting all of these suggested use-cases together, I believe that in the future we can have foundation models that operate and design data centres. These foundation models can then perform all the maintenance and accounting for a DC. They can decide how to schedule workloads, what energy sources to use, they can detect intrusions in the system, and they could even suggest hardware improvements. Foundation models are built on large amounts of data. Given that data centres are isolated environments, all the data can be collected and traced to its origins. The potential use cases of such large amounts of data are unlimited.

Bibliography

[1] I. Akbari, M. A. Salahuddin, L. Aniva, N. Limam, R. Boutaba, B. Mathieu, S. Moteau, and S. Tuffin, "Traffic classification in an increasingly encrypted web," *Communications of the ACM*, vol. 65, no. 10, pp. 75–83, 2022.

[2] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 139–152, 2015.

[3] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 440–453, 2016.

[4] D. A. Popescu, "Measurement-based resource allocation and control in data centers: A survey," *arXiv preprint arXiv:2408.09497*, 2024.

[5] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 435–446, 2013.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the seventh conference on emerging networking experiments and technologies*, pp. 1–12, 2011.

[7] W. Liu, Y. Yan, Y. Sun, H. Mao, M. Cheng, P. Wang, and Z. Ding, "Online job scheduling scheme for low-carbon data center operation: An information and energy nexus perspective," *Applied Energy*, vol. 338, p. 120918, 2023.

[8] J. C. Mogul and M. Arlitt, "Sc2d: an alternative to trace anonymization," in *Proceedings of the 2006 SIGCOMM workshop on Mining network data*, pp. 323–328, 2006.

[9] J. Mirkovic, "Privacy-safe network trace sharing via secure queries," in *Proceedings of the 1st ACM workshop on Network data anonymization*, pp. 3–10, 2008.

[10] M. Mohammady, L. Wang, Y. Hong, H. Louafi, M. Pourzandi, and M. Debbabi, "Preserving both privacy and utility in network trace anonymization," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 459–474, 2018.

[11] J. L. Guerra, C. Catania, and E. Veas, "Datasets are not enough: Challenges in labeling network traffic," *Computers & Security*, vol. 120, p. 102810, 2022.

[12] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[13] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," *ACM SIGCOMM computer communication review*, vol. 45, no. 4, pp. 183–197, 2015.

[14] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 123–137, 2015.

[15] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[16] A. Andreyev, "Introducing data center fabric, the next-generation Facebook data center network." `https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/`, November 2014.

[17] A. Vahdat, H. Liu, J. Snow, P. Ranganathan, and B. Koley, "Speed, scale, reliability: 25 years of data center networking." Google Cloud Blog `https://cloud.google.com/blog/products/networking/speed-scale-reliability-25-years-of-data-center-networking`, February 2024.

[18] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[19] M. Noormohammadpour and C. S. Raghavendra, "Datacenter traffic control: Understanding techniques and tradeoffs," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1492–1525, 2017.

[20] N. Zilberman, M. Grosvenor, D. A. Popescu, N. Manihatty-Bojan, G. Antichi, M. Wójcik, and A. W. Moore, "Where has my time gone?," in *Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings 18*, pp. 201–214, Springer, 2017.

[21] R. L. Sites, *Understanding software dynamics*. Addison-Wesley Professional, 2021.

[22] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[23] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling.," in *OSDI*, vol. 4, pp. 18–18, 2004.

[24] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[25] T. P. Eszterhas, "Measuring queuing effects in data center networks," MPhil Advanced Computer Science Thesis, University of Cambridge, June 2021.

[26] Endace, "Endace DAG 10X2-S," tech. rep., BEST Systeme GmbH / Endace, 2016. Accessed on May 30, 2025.

[27] Exablaze, "ExaNIC X10 Datasheet," tech. rep., Exablaze. Accessed on May 30, 2025.

[28] "Exalink fusion - ultra low latency 10gbe switch." `https://xenon.com.au/product/exalink-fusion/`, 2024. Accessed on June 1, 2025.

[29] G. Harris, M. Richardson, and Sandelman, "PCAP Capture File Format," Tech. Rep. draft-gharris-opsawg-pcap-00, Internet Engineering Task Force, Dec. 2020.

[30] A. W. Moore and K. Papagiannaki, "Toward the accurate identification of network applications," in *International workshop on passive and active network measurement*, pp. 41–54, Springer, 2005.

[31] A. Madhukar and C. Williamson, "A longitudinal study of p2p traffic classification," in *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pp. 179–188, 2006.

[32] S. Fernandes, R. Antonello, T. Lacerda, A. Santos, D. Sadok, and T. Westholm, "Slimming down deep packet inspection systems," in *IEEE INFOCOM Workshops 2009*, pp. 1–6, IEEE, 2009.

[33] N. Hubballi and M. Swarnkar, "*bitcoding*: Network traffic classification through encoded bit level signatures," *IEEE/ACM Transactions On Networking*, vol. 26, no. 5, pp. 2334–2346, 2018.

[34] A. W. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, p. 50–60, June 2005.

[35] N. Williams and S. Zander, "Evaluating machine learning algorithms for automated network application identification," 2006.

[36] K. L. Dias, M. A. Pongelupe, W. M. Caminhas, and L. De Errico, "An innovative approach for real-time network traffic classification," *Computer networks*, vol. 158, pp. 143–157, 2019.

[37] R. Alshammari and A. N. Zincir-Heywood, "Machine learning based encrypted traffic classification: Identifying SSH and Skype," in *2009 IEEE symposium on computational intelligence for security and defense applications*, pp. 1–8, IEEE, 2009.

[38] S. Dong, "Multi class SVM algorithm with active learning for network traffic classification," *Expert Systems with Applications*, vol. 176, p. 114885, 2021.

[39] N.-F. Huang, G.-Y. Jai, H.-C. Chao, Y.-J. Tzang, and H.-Y. Chang, "Application traffic classification at the early stage by characterizing application rounds," *Information Sciences*, vol. 232, pp. 130–142, 2013.

[40] Z. Wang, "The applications of deep learning on traffic identification," *BlackHat USA*, vol. 24, no. 11, pp. 1–10, 2015.

[41] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *2017 International conference on information networking (ICOIN)*, pp. 712–717, IEEE, 2017.

[42] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang, "End-to-end encrypted traffic classification with one-dimensional convolution neural networks," in *2017 IEEE international conference on intelligence and security informatics (ISI)*, pp. 43–48, IEEE, 2017.

[43] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas, and J. Lloret, "Network traffic classifier with convolutional and recurrent neural networks for internet of things," *IEEE access*, vol. 5, pp. 18042–18050, 2017.

[44] G. D. Gil, A. H. Lashkari, M. Mamun, and A. A. Ghorbani, "Characterization of encrypted and VPN traffic using time-related features," in *Proceedings of the 2nd*

international conference on information systems security and privacy (ICISSP 2016), pp. 407–414, SciTePress Setúbal, Portugal, 2016.

[45] MAWI Working Group, "MAWI Working Group Traffic Archive." `http://mawi.wide.ad.jp/mawi/`, 1999–Present. Accessed: May 31, 2025.

[46] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying IoT devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, pp. 1745–1759, 2018.

[47] G. Aceto, D. Ciuonzo, A. Montieri, V. Persico, and A. Pescapé, "Mirage: Mobile-app traffic capture and ground-truth creation," in *2019 4th International conference on computing, communications and security (ICCCS)*, pp. 1–8, IEEE, 2019.

[48] R. Wang, Z. Liu, Y. Cai, D. Tang, J. Yang, and Z. Yang, "Benchmark data for mobile app traffic research," in *Proceedings of the 15th EAI international conference on mobile and ubiquitous systems: Computing, networking and services*, pp. 402–411, 2018.

[49] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K. Claffy, "Gt: Picking up the truth from the ground for internet traffic," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 5, pp. 12–18, 2009.

[50] S. Garcia, M. Grill, J. Stiborek, and A. Zunino, "An empirical comparison of botnet detection methods," *computers & security*, vol. 45, pp. 100–123, 2014.

[51] R. Alshammari and A. N. Zincir-Heywood, "An investigation on the identification of VoIP traffic: Case study on Gtalk and Skype," in *2010 International Conference on Network and Service Management*, pp. 310–313, IEEE, 2010.

[52] P. Branch and J. But, "Rapid and generalized identification of packetized voice traffic flows," in *37th Annual IEEE Conference on Local Computer Networks*, pp. 85–92, IEEE, 2012.

[53] D. Adami, C. Callegari, S. Giordano, M. Pagano, and T. Pepe, "Skype-Hunter: A real-time system for the detection and classification of Skype traffic," *International Journal of Communication Systems*, vol. 25, no. 3, pp. 386–403, 2012.

[54] A. A. Afuwape, Y. Xu, J. H. Anajemba, and G. Srivastava, "Performance evaluation of secured network traffic classification using a machine learning approach," *Computer Standards & Interfaces*, vol. 78, p. 103545, 2021.

[55] P. Wang, F. Ye, X. Chen, and Y. Qian, "Datanet: Deep learning based encrypted network traffic classification in SDN home gateway," *IEEE Access*, vol. 6, pp. 55380–55391, 2018.

[56] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74, 2010.

[57] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 66–76, IEEE, 2013.

[58] D. Ersoz, M. S. Yousif, and C. R. Das, "Characterizing network traffic in a cluster-based, multi-tier data center," in *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pp. 59–59, IEEE, 2007.

[59] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pp. 202–208, 2009.

[60] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, 2010.

[61] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[62] MLCommons Association, "Mlperf™ inference: Datacenter." MLCommons Website `https://mlcommons.org/benchmarks/inference-datacenter/`, June 2024. Accessed on June 1, 2025.

[63] J. Zerwas, K. Aykurt, S. Schmid, and A. Blenk, "Network traffic characteristics of machine learning frameworks under the microscope," in *2021 17th International Conference on Network and Service Management (CNSM)*, pp. 207–215, 2021.

[64] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter, "Bullet trains: a study of NIC burst behavior at microsecond timescales," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, (New York, NY, USA), p. 133–138, Association for Computing Machinery, 2013.

[65] J. Woodruff, A. W. Moore, and N. Zilberman, "Measuring burstiness in data center applications," in *Proceedings of the 2019 Workshop on Buffer Sizing*, pp. 1–6, 2019.

[66] Y. A. Harchol, C. Reiss, and J. Wilkes, "Yet more Google compute cluster trace data." Google Research Blog `https://research.google/blog/yet-more-google-compute-cluster-trace-data/`, February 2024.

[67] Alibaba Group, "Alibaba cluster trace program. clusterdata." `https://github.com/alibaba/clusterdata/tree/master`, March 2019. Accessed on June 1, 2025.

[68] The SWIM Project at UCB, "SWIM (Scalable Partially Ordered Workload Generator) wiki." `https://github.com/SWIMProjectUCB/SWIM/wiki`, November 2016. Accessed on June 1, 2025.

[69] J. C. Woodruff, "Analyzing data center applications using high-precision packet traces," MPhil Advanced Computer Science Thesis, University of Cambridge, June 2019.

[70] Cisco, "Exanic drivers, utilities and development libraries." `https://github.com/cisco/exanic-software`, 2023. Accessed on May 26, 2025.

[71] R. Kannan, A. Raman, D. Wong, V. Rao, K. Hsieh, R. L. Fontes, N. Stufflet, W. T. B. Man Io, K. Hazelwood, and R. Bianchini, "DCPerf: An open-source benchmark suite for hyperscale compute applications." `https://engineering.fb.com/2024/08/05/data-center-engineering/dcperf-open-source-benchmark-suite-for-hyperscale-compute-applications/`, August 2024. Accessed: 2025-06-01.

[72] Facebook Engineering, "TAO: The power of the graph." https://engineering.fb.com/2013/06/25/core-infra/tao-the-power-of-the-graph/, June 2013. Accessed: 2025-06-01.

[73] GoogleCloudPlatform, "Oldisim: A simulator for OLDI (online data-intensive) applications." https://github.com/GoogleCloudPlatform/oldisim, Accessed on June 2025.

[74] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook white paper*, vol. 5, no. 8, p. 127, 2007.

[75] M. Thomson and M. Nottingham, "Using Early Data in HTTP," RFC 8478, RFC Editor, October 2018.

[76] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Commun. ACM*, vol. 59, p. 62–69, Feb. 2016.

[77] Association for Computing Machinery, "Artifact review and badging - current." https://www.acm.org/publications/policies/artifact-review-and-badging-current, August 2020. Accessed on May 22, 2025.

[78] A. Moore, D. Zuev, and M. Crogan, "Discriminators for use in flow-based classification," tech. rep., University of Cambridge, 2005.

[79] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush, "From Paris to Tokyo: On the suitability of ping to measure latency," in *Proceedings of the 2013 conference on Internet measurement conference*, pp. 427–432, 2013.

[80] M. A. Hall, *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

Appendix A

Experimental Setup Used

This appendix contains the details of the hardware setup used in the experiments.

Component	**Specification**
Operating System	Ubuntu 24.04.1 LTS
Linux Kernel Version	6.8.0-52-generic
CPU	Intel(R) Xeon(R) CPU E5-2430L v2 @ 2.40GHz 12 Cores
RAM	64 GB DDR3 1600MHz
Storage	Micron P400m-MTF 100GB SSD Seagate ST1000NM0033-9ZM 1TB HDD
Network Cards	Broadcom NetXtreme BCM5720 Gigabit Ethernet PCIe 1Gbit/s Intel Corporation Ethernet 10G 2P X520 Adapter 10Gbit/s

Table A.1: Test-bed Machines Specifications

Model	Arista Networks DCS-7050Q-16 switch
Latency	800-850ns
Ports	16 ports of 40GbE
Maximum Frame Size	9236 bytes (allows Jumbo Frames)

Table A.2: Leaf Switch Specifications

Component	**Specification**
Operating System	Ubuntu 16.04.5 LTS
Linux Kernel Version	4.4.0-31-generic
CPU	Intel(R) Xeon(R) CPU E5-2658 v4 @ 2.30GHz 27 Cores
RAM	256 GB DDR3 1600MHz
Storage	Kingston SSDNow UV400 Write speeds of 1Gbit/s Kingston SA400S3 SSD Write speeds of 3Gbit/s

Table A.3: Capture Machines Specifications

Appendix B

Configuration Parameters for Workloads

The detailed parameters of the chosen workloads are detailed in this section. Further down the CLI of *benchpress* is detailed.

Parameter	Explanation	Default Value
–memsize	Specifies the cache memory size in GB.	Required
–port-number	The port number the server will listen on.	11211
–num-servers	Number of servers to spawn.	Number of cores / 72
–pin-threads	Pin server to specific CPUs.	No pinning
–interface-name	Name of the network interface to configure, the one the server will bind to.	eth0
–smart-nanosleep	Enable a randomised nanosleep with exponential back-off for thread yielding.	False
–warmup-time	Duration of warm up phase in seconds.	1200
–test-time	Duration of test phase in seconds.	360
–timeout-buffer	Extra time between the two phases.	120

Table B.1: Server Parameters for TaoBench Workload

Parameter	Explanation	Default Value
–server-hostname	Hostname or IP address of the server.	Required
–server-port-number	Port number to which the client should connect.	Required
–server-memsize	The memory size of the server.	Required
–num-threads	Number of client threads.	Number of cores - 6
–clients-per-thread	Number of client connections per thread.	380
–warmup-time	Duration of warm up phase in seconds.	1200
–test-time	Duration of test phase in seconds.	360
–wait-after-warmup	Extra time in between the two phases.	120

Table B.2: Client Parameters for TaoBench Workload

Parameter	Explanation	Default Value
–db-addr	Address of the database.	Required
–server-workers	The number of uWGSI workers.	Number of cores
–duration	Duration of the benchmark.	5 minutes

Table B.3: Server Parameters for DjangoBench workload

Parameter	Explanation	Default Value
–server-ip	IP address of the server.	Required
–duration	Duration of the benchmark.	5 minutes
–client-workers	Number of Client Workers	Number of Cores
–repetitions	How many times the benchmark should be repeated	1
–client-think	Simulate client think time or not.	False

Table B.4: Client Parameters for DjangoBench Workload

Parameter	Explanation	Default Value
–server-ip	IP address of the server.	Required
–server-port	Port used by the server.	19212
–client-workers	Number of Client Workers	4
–client-threads	Number of Threads per Client Worker	4
–queries	Number of QPS used.	0 (search mode)
–warmup-duration	Warm up duration in seconds.	60
–test-duration	Test phase duration in seconds.	120

Table B.5: Client Parameters for Feedsim Workload

Command	Explanation	Additional Info
list	Lists all installed benchmarks.	
report	Reports all benchmark results.	The tool was intended to run all benchmarks at once to get a feel of the systems performance, hence why it reports results for everything.
run <benchmark>	Run a certain benchmark	Accepts parameters that cna be passed to the actual run scripts.
install <benchmark>	Will install a benchmark with all its dependencies.	All installations are done in a local directory.
clean <benchmark>	Removes all dependencies for a given benchmark.	
info <benchmark>	Provides a quick reference for the given benchmark.	

Table B.6: Summary of the functionality provided by *benchpress*

Appendix C

List of Computed Features for Network Flows

Table C.1: Complete List of Features and Descriptions

Index	Feature Name	Description
Packet-Size Statistics (all directions)		
1	min_wire_size_all	Minimum packet size
2	q1_wire_size_all	First quartile packet size
3	med_wire_size_all	Median packet size
4	mean_wire_size_all	Mean packet size
5	q3_wire_size_all	Third quartile packet size
6	max_wire_size_all	Maximum packet size
7	var_wire_size_all	Variance of packet sizes
8	min_ip_size_all	Minimum IP packet size
9	q1_ip_size_all	First quartile IP packet size
10	med_ip_size_all	Median IP packet size
11	mean_ip_size_all	Mean IP packet size
12	q3_ip_size_all	Third quartile IP packet size
13	max_ip_size_all	Maximum IP packet size
14	var_ip_size_all	Variance of IP packet sizes
15	min_ctrl_size_all	Minimum control overhead size
16	q1_ctrl_size_all	First quartile control overhead size
17	med_ctrl_size_all	Median control overhead size
18	mean_ctrl_size_all	Mean control overhead size
19	q3_ctrl_size_all	Third quartile control overhead size
20	max_ctrl_size_all	Maximum control overhead size
21	var_ctrl_size_all	Variance of control overhead sizes
Packet-Size Statistics (server→client)		
22	min_wire_size_ab	Minimum packet size server→client
23	q1_wire_size_ab	First quartile packet size server→client
24	med_wire_size_ab	Median packet size server→client
25	mean_wire_size_ab	Mean packet size server→client
26	q3_wire_size_ab	Third quartile packet size server→client

Continued on next page

Table C.1 – continued from previous page

Index	Feature Name	Description
27	max_wire_size_ab	Maximum packet size server→client
28	var_wire_size_ab	Variance of packet sizes server→client
29	min_ip_size_ab	Minimum IP packet size server→client
30	q1_ip_size_ab	First quartile IP packet size server→client
31	med_ip_size_ab	Median IP packet size server→client
32	mean_ip_size_ab	Mean IP packet size server→client
33	q3_ip_size_ab	Third quartile IP packet size server→client
34	max_ip_size_ab	Maximum IP packet size server→client
35	var_ip_size_ab	Variance of IP packet sizes server→client
36	min_ctrl_size_ab	Minimum control overhead size server→client
37	q1_ctrl_size_ab	First quartile control overhead size server→client
38	med_ctrl_size_ab	Median control overhead size server→client
39	mean_ctrl_size_ab	Mean control overhead size server→client
40	q3_ctrl_size_ab	Third quartile control overhead size server→client
41	max_ctrl_size_ab	Maximum control overhead size server→client
42	var_ctrl_size_ab	Variance of control overhead sizes server→client
Packet-Size Statistics (client→server)		
43	min_wire_size_ba	Minimum packet size client→server
44	q1_wire_size_ba	First quartile packet size client→server
45	med_wire_size_ba	Median packet size client→server
46	mean_wire_size_ba	Mean packet size client→server
47	q3_wire_size_ba	Third quartile packet size client→server
48	max_wire_size_ba	Maximum packet size client→server
49	var_wire_size_ba	Variance of packet sizes client→server
50	min_ip_size_ba	Minimum IP packet size client→server
51	q1_ip_size_ba	First quartile IP packet size client→server
52	med_ip_size_ba	Median IP packet size client→server
53	mean_ip_size_ba	Mean IP packet size client→server
54	q3_ip_size_ba	Third quartile IP packet size client→server
55	max_ip_size_ba	Maximum IP packet size client→server
56	var_ip_size_ba	Variance of IP packet sizes client→server
57	min_ctrl_size_ba	Minimum control overhead size client→server
58	q1_ctrl_size_ba	First quartile control overhead size client→server
59	med_ctrl_size_ba	Median control overhead size client→server
60	mean_ctrl_size_ba	Mean control overhead size client→server
61	q3_ctrl_size_ba	Third quartile control overhead size client→server
62	max_ctrl_size_ba	Maximum control overhead size client→server
63	var_ctrl_size_ba	Variance of control overhead sizes client→server
Transfer Mode Statistics		
64	num_bulk_periods	Number of bulk transfer periods
65	bulk_duration	Total duration of bulk transfers
66	flow_duration	Total flow duration
67	pct_bulk	Percentage of time in bulk transfer
68	idle_duration	Total idle time (gaps ¿ 2 seconds)
69	pct_idle	Percentage of time idle

Continued on next page

Table C.1 – continued from previous page

Index	Feature Name	Description
70	interactive_duration	Total interactive time (gaps ¡ 2 seconds)
71	pct_interactive	Percentage of time in interactive mode
Inter-Packet Timing Statistics		
72	min_iat_all	Minimum inter-packet time
73	q1_iat_all	First quartile inter-packet time
74	med_iat_all	Median inter-packet time
75	mean_iat_all	Mean inter-packet time
76	q3_iat_all	Third quartile inter-packet time
77	max_iat_all	Maximum inter-packet time
78	var_iat_all	Variance of inter-packet times
79	min_iat_ab	Minimum inter-packet time server→client
80	q1_iat_ab	First quartile inter-packet time server→client
81	med_iat_ab	Median inter-packet time server→client
82	mean_iat_ab	Mean inter-packet time server→client
83	q3_iat_ab	Third quartile inter-packet time server→client
84	max_iat_ab	Maximum inter-packet time server→client
85	var_iat_ab	Variance of inter-packet times server→client
86	min_iat_ba	Minimum inter-packet time client→server
87	q1_iat_ba	First quartile inter-packet time client→server
88	med_iat_ba	Median inter-packet time client→server
89	mean_iat_ba	Mean inter-packet time client→server
90	q3_iat_ba	Third quartile inter-packet time client→server
91	max_iat_ba	Maximum inter-packet time client→server
92	var_iat_ba	Variance of inter-packet times client→server
Segment Size Statistics		
93	mss_ab	MSS value requested by A in SYN packet server→client
94	mss_ba	MSS value requested by B in SYN packet client→server
95	max_seg_ab	Maximum segment size observed server→client
96	max_seg_ba	Maximum segment size observed client→server
97	min_seg_ab	Minimum segment size observed server→client
98	min_seg_ba	Minimum segment size observed client→server
99	avg_seg_ab	Average segment size server→client
100	avg_seg_ba	Average segment size client→server
101	max_seg_all	Maximum segment size observed in either direction
102	min_seg_all	Minimum segment size observed in either direction
103	avg_seg_all	Average segment size across both directions
Packet and Byte Counts		
104	pkts_ab	Number of packets server→client
105	bytes_ab	Number of bytes server→client
106	thrpt_ab	Throughput server→client (bytes/sec)
107	pkts_ba	Number of packets client→server
108	bytes_ba	Number of bytes client→server

Continued on next page

Table C.1 – continued from previous page

Index	Feature Name	Description
109	thrpt_ba	Throughput client→server (bytes/sec)
TCP Flags Statistics		
110	syn_ab	Number of SYN packets server→client
111	syn_ba	Number of SYN packets client→server
112	fin_ab	Number of FIN packets server→client
113	fin_ba	Number of FIN packets client→server
114	ack_ab	Number of ACK packets server→client
115	ack_ba	Number of ACK packets client→server
116	pure_ack_ab	Number of pure ACK packets server→client (ACK only, no data)
117	pure_ack_ba	Number of pure ACK packets client→server (ACK only, no data)
118	urg_ab	Number of URG packets server→client
119	urg_ba	Number of URG packets client→server
120	urg_bytes_ab	Total bytes of urgent data server→client
121	urg_bytes_ba	Total bytes of urgent data client→server
122	push_ab	Number of PUSH packets server→client
123	push_ba	Number of PUSH packets client→server
124	sack_ab	Number of packets with SACK blocks server→client
125	sack_ba	Number of packets with SACK blocks client→server
126	dsack_ab	Number of packets with D-SACK blocks server→client
127	dsack_ba	Number of packets with D-SACK blocks client→server
128	max_sack_blks_ab	Maximum number of SACK blocks in any packet server→client
129	max_sack_blks_ba	Maximum number of SACK blocks in any packet client→server
130	data_ab	Number of data packets server→client
131	data_ba	Number of data packets client→server
132	data_bytes_ab	Total bytes of data packets server→client
133	data_bytes_ba	Total bytes of data packets client→server
TCP Handshake Statistics		
134	tcp_handshake	Whether TCP handshake is present
135	tcp_handshake_direction_ab	Whether handshake is from server→client
136	tcp_handshake_direction_ba	Whether handshake is from client→server
137	tcp_handshake_end	Index of first packet after handshake
Ordering Statistics		
138	oo_ab	Number of out-of-order packets server→client
139	oo_ba	Number of out-of-order packets client→server
140	oo_bytes_ab	Total bytes of out-of-order packets server→client
141	oo_bytes_ba	Total bytes of out-of-order packets client→server
142	retrans_ab	Number of retransmitted packets server→client
143	retrans_ba	Number of retransmitted packets client→server

Continued on next page

Table C.1 – continued from previous page

Index	Feature Name	Description
144	retrans_bytes_ab	Total bytes of retransmitted packets server→client
145	retrans_bytes_ba	Total bytes of retransmitted packets client→server
146	unique_bytes_ab	Total bytes of unique packets server→client
147	unique_bytes_ba	Total bytes of unique packets client→server
Theoretical Stream Length Statistics		
162	theoretical_stream_length_ab	TSL for server→client direction
163	theoretical_stream_length_ba	TSL for client→server direction
164	missed_data_ab	Bytes missed in server→client direction
165	missed_data_ba	Bytes missed in client→server direction
Timestamp Statistics		
166	rtt_min_ab	Minimum RTT for server→client→server
167	rtt_max_ab	Maximum RTT for server→client→server
168	rtt_avg_ab	Average RTT for server→client→server
169	rtt_std_ab	Standard deviation of RTT for server→client→server
170	rtt_min_ba	Minimum RTT for client→server→client
171	rtt_max_ba	Maximum RTT for client→server→client
172	rtt_avg_ba	Average RTT for client→server→client
173	rtt_std_ba	Standard deviation of RTT for client→server→client
174	min_equal_tsval_ab	Minimum length of consecutive packets with same TSval server→client
175	max_equal_tsval_ab	Maximum length of consecutive packets with same TSval server→client
176	min_equal_tsval_ba	Minimum length of consecutive packets with same TSval client→server
177	max_equal_tsval_ba	Maximum length of consecutive packets with same TSval client→server
Large-Transfer Statistics		
178	num_large_transfers_ab	Number of large transfers in server→client direction
179	num_large_transfers_ba	Number of large transfers in client→server direction
180	max_large_transfer_ab	Maximum number of packets in a large transfer in server→client direction
181	max_large_transfer_ba	Maximum number of packets in a large transfer in client→server direction
182	avg_large_transfer_ab	Average number of packets in a large transfer in server→client direction
183	avg_large_transfer_ba	Average number of packets in a large transfer in client→server direction
184	std_large_transfer_ab	Standard deviation of the number of packets in a large transfer in server→client direction
185	std_large_transfer_ba	Standard deviation of the number of packets in a large transfer in client→server direction
Microburst Statistics		
186	num_microburst_ab	Number of microburst periods server→client direction

Continued on next page

Table C.1 – continued from previous page

Index	Feature Name	Description
187	microburst_max_ab_length	Maximum length of a microburst (in packets) in server→client direction
188	microburst_min_length_ab	Minimum length of a microburst server→client direction
189	microburst_avg_length_ab	Average length of all microbursts server→client direction
190	microburst_std_length_ab	Standard deviation of microbursts lengths server→client direction
191	max_duration_microburst_ab	Maximum duration of a microburst in seconds server→client direction
192	min_duration_microburst_ab	Minimum duration of a microburst in seconds server→client direction
193	avg_duration_microburst_ab	Average duration of a microburst in seconds server→client direction
194	std_duration_microburst_ab	Standard deviation of microburst durations server→client direction
195	num_microburst_ba	Number of microburst periods in client→server direction
196	microburst_max_ba_length	Maximum length of a microburst (in packets) in client→server direction
197	microburst_min_length_ba	Minimum length of a microburst client→server direction
198	microburst_avg_length_ba	Average length of all microbursts client→server direction
199	microburst_std_length_ba	Standard deviation of microbursts lengths client→server direction
200	max_duration_microburst_ba	Maximum duration of a microburst in seconds client→server direction
201	min_duration_microburst_ba	Minimum duration of a microburst in seconds client→server direction
202	avg_duration_microburst_ba	Average duration of a microburst in seconds client→server direction
203	std_duration_microburst_ba	Standard deviation of microburst durations client→server direction