Network-based Application Monitoring as a Service in Cloud Data Centers

Mona ElSaadawy

Doctor of Philosophy

School of Computer Science

McGill University

Montréal, Québec, Canada

June, 2022

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Abstract

Nowadays, many cloud applications can be considered large complex distributed services. The increasing sophistication and complexity have made performance monitoring a major issue and a critical process for both cloud providers and cloud customers. Many different monitoring techniques are used for such applications to diagnose and resolve performance issues, from simply measuring resource consumption, to application-specific measures such throughput and request service time as well as observing how the different application components invoke each other in form of a call graph during run-time. To fulfill such monitoring needs, existing cloud monitoring systems require sophisticated application and/or platform instrumentation which is cumbersome, requires expertise knowledge about the application and/or platform, and cannot be deployed on-demand.

Interestingly, many of the application-specific performance metrics and dependencies can be inferred from the network messages exchanged between the application components and a variety of monitoring tools are thus based on such message exchange. Moreover, using new trends such as Software Defined Networking (SDN) and Network Function Virtualization (NFV) shows promise to instantiate monitoring functionalities into the network on-demand. However, many of the existing network-based monitoring tools provide so far only partial functionality and/or come with a significant overhead.

As such this thesis explores in a comprehensive manner the potential for the cloud provider to offer holistic transparent *Monitoring-as-a-Service (MaaS)* functionality that is purely based on monitoring network traffic avoiding software instrumentation and allows a flexible placement of monitoring functionality in the network, and at the same time runs with low overhead. The MaaS functionality includes providing some component-level performance metrics, such as response time and throughput, information about the dependency of components during run-time, and identifying the service type each component provides e.g., MySQL database or a web-service.

We have worked on three contributions in this direction. Firstly, we explore how to collect the measurements in the network and where to place the analysis functionality. Towards this end, we explore existing network-based monitoring approaches and adjust them for application monitoring, as well as propose a new network-based application monitoring approach. In particular, we combine port mirroring with tunneling to enable message filtering and reformatting, and we propose a novel *sniffing* approach. We provide their implementations based on a software switch, analyze their advantages and disadvantages, and provide a comprehensive performance evaluation to highlight the trade-offs and show that moving application monitoring to the network is an attractive option.

Secondly, we have designed and implemented a *MaaS prototype*, using the proposed network-based monitoring approach that we have developed in the first step as core building block. The MaaS prototype is itself a distributed system that follows a client-server architecture that enables a flexible deployment of the monitoring and analysis functionalities into the network. The MaaS prototype uses monitoring agents that are co-located with software switches in order to extract performance metrics from the message flows between application components in a non-intrusive manner, and that send the calculated metrics to the administrators for visualization in near real-time and with acceptable overhead. The agents support several service types and a wide range of performance metrics to be monitored on-demand. In addition, the MaaS prototype is designed in a modular way that allows for extensibility where new types of services and new performance measures can be added to the MaaS in an incremental manner.

Thirdly, we have developed the *DyMonD* system that creates and visualizes application call graph formations and offer service identification. It uses our network-based monitoring approach to provide the cloud users a global view of how their application is actually executing across services. DyMonD analyzes the packet traffic among components by observing specific network flows at the switch-level, again in a lightweight manner. With the extracted information it can determine call dependencies and service types, and track performance metrics. In addition, DyMonD optionally performs a fine-grained service type identification for microservice-based architectures. Among many approaches that perform service type identification using network messages, including rule-based, statistical correlation-based and deep learning-based approaches, DyMonD employs a novel deep learning model that is based on Bidirectional LSTM layers to dynamically identify the service type provided by each component to build a proper call graph. The advantage of the deep learning approaches is that they limit the need of expert intervention. The evaluation results show that DyMonD provides the call graph with high accuracy and with a reasonable overhead compared to other network-based and application instrumentation-based tools.

In summary, this thesis presents a holistic non-intrusive MaaS platform for monitoring distributed applications that can be applied in a wide range of settings and provides both the cloud and application administrators with the information they need to diagnose and resolve performance issues.

Abrégé

De nombreuses applications cloud peuvent aujourd'hui être considérées comme de grands services distribués complexes. La sophistication et la complexité croissantes ont fait de la surveillance des performances un problème majeur et un processus critique pour les fournisseurs et les clients du cloud. De nombreuses techniques de surveillance sont utilisées pour ces applications afin de diagnostiquer et de résoudre les problèmes de performance, allant de la simple mesure de la consommation des ressources à des mesures spécifiques à l'application telles que le débit et le temps de traitement des demandes, en passant par l'observation de la manière dont les différents composants de l'application s'invoquent mutuellement sous la forme d'un graphe d'appel pendant l'exécution. Pour répondre à ces besoins de surveillance, les systèmes existants de surveillance du cloud requièrent une instrumentation sophistiquée de l'application et/ou de la plate-forme, ce qui est difficile, nécessite une connaissance approfondie de l'application et/ou de la plate-forme et ne peut être déployé à la demande.

Il est intéressant de noter que de nombreuses mesures de performance et dépendances spécifiques à l'application peuvent être déduites des messages réseau échangés entre les composants de l'application et qu'une variété d'outils de surveillance sont donc basés sur ces échanges de messages. En outre, l'utilisation de nouvelles tendances telles que le SDN (Software Defined Networking) et la NFV (Network Function Virtualization) est prometteuse pour instancier des fonctionnalités de surveillance dans le réseau à la demande. Cependant, bon nombre des outils de surveillance existants basés sur le réseau ne fournissent jusqu'à présent qu'une fonctionnalité partielle et/ou sont assortis d'une surcharge importante.

Cette thèse explore de manière exhaustive le potentiel pour le fournisseur de cloud d'offrir une fonctionnalité MaaS (Monitoring-as-a-Service) holistique et transparente qui est purement basée sur la surveillance du trafic réseau sans instrumentation logicielle et permet un placement flexible de la fonctionnalité de surveillance dans le réseau, tout en fonctionnant avec une faible surcharge. La fonctionnalité MaaS comprend la fourniture de certaines mesures de performance au niveau des composants, comme le temps de réponse et le débit, des informations sur la dépendance des composants pendant l'exécution, et l'identification du type de service fourni par chaque composant, par exemple une base de données MySQL ou un service Web.

Nous avons travaillé sur trois contributions dans cette direction. Premièrement, nous explorons comment collecter les mesures dans le réseau et où placer la fonctionnalité d'analyse. À cette fin, nous explorons les approches de surveillance existantes basées sur le réseau et les adaptons à la surveillance des applications, et nous proposons une nouvelle approche de surveillance des applications basée sur le réseau. En particulier, nous combinons la mise en miroir des ports avec le tunneling pour permettre le filtrage et le reformatage des messages, et nous proposons une nouvelle approche de reniflement. Nous proposons leurs implémentations basées sur un commutateur logiciel, analysons leurs avantages et inconvénients, et fournissons une évaluation complète des performances pour mettre en évidence les compromis et montrer que le transfert de la surveillance des applications vers le réseau est une option intéressante.

Deuxièmement, nous avons conçu et mis en œuvre un prototype MaaS, en utilisant comme élément de base l'approche de surveillance basée sur le réseau que nous avons développé dans la première étape. Le prototype MaaS est lui-même un système distribué qui suit une architecture client-serveur permettant un déploiement flexible des fonctionnalités de surveillance et d'analyse dans le réseau. Le prototype MaaS utilise des agents de surveillance qui sont co-localisés avec les commutateurs logiciels afin d'extraire les mesures de performance des flux de messages entre les composants d'application de manière non intrusive, et qui envoient les mesures calculées aux administrateurs pour visualisation en temps quasi réel et avec un surcoût acceptable. Les agents prennent en charge plusieurs types de services et un large éventail de mesures de performance à surveiller à la demande. En outre, le prototype MaaS est conçu de façon modulaire, ce qui permet une extensibilité grâce à laquelle de nouveaux types de services et de nouvelles mesures de performance peuvent être ajoutés au MaaS de manière progressive.

Troisièmement, nous avons développé le système DyMonD qui crée et visualise les formations de graphes d'appel des applications et l'identification des services d'offre. Il utilise notre approche de surveillance basée sur le réseau pour fournir aux utilisateurs du cloud une vision globale de la façon dont leur application s'exécute réellement à travers les services. DyMonD analyse le trafic de paquets entre les composants en observant des flux réseau spécifiques au niveau du commutateur, là encore de manière légère. Grâce aux informations extraites, il peut déterminer les dépendances d'appel et les types de service, et suivre les mesures de performance. En outre, DyMonD effectue en option une identification du type de service détaillée pour les architectures basées sur les micro-services. Parmi les nombreuses approches qui effectuent l'identification du type de service à l'aide de messages réseau, y compris les approches basées sur des règles, des corrélations statistiques et l'apprentissage profond, DyMonD utilise un nouveau modèle d'apprentissage profond basé sur des couches LSTM bidirectionnelles pour identifier dynamiquement le type de service fourni par chaque composant afin de construire un graphe d'appel adéquat. L'avantage des approches d'apprentissage profond est qu'elles limitent le besoin d'intervention d'un expert. Les résultats de l'évaluation montrent que DyMonD fournit le graphe d'appel avec une

grande précision et avec un surcoût raisonnable par rapport à d'autres outils basés sur le réseau et l'instrumentation des applications.

En résumé, cette thèse présente une plate-forme MaaS holistique et non intrusive pour la surveillance des applications distribuées qui peut être appliquée dans un grand éventail de paramètres et fournit à la fois aux administrateurs du cloud et des applications les informations dont ils ont besoin pour diagnostiquer et résoudre les problèmes de performance.

Acknowledgements

This PhD thesis would not have been possible without the support of many people. Foremost, I am extremely grateful to my supervisor, Prof. Bettina Kemme, for her noble guidance, support, full encouragement, enthusiasm, patience and continuous faith in me over the years of my PhD journey. Her immense knowledge and plentiful experience have given me the power and spirit to excel in my academic research and daily life. She is the best mentor and advisor for my doctorate study beyond the imagination. Apart from my supervisor, I want to express my gratitude to Prof. Mohamed Younis for his precious contribution and guidance for my publications. His endless help and support are hard to forget throughout my life.

I extend my gratitude to my PhD progress committee, Profs. Muthucumaru Maheswaran and Jörg Kienzle for their invaluable advice, feedback and sharing insightful suggestions.

I would also like to thank my labmates, Omar Asad, Joseph D'Silva and Maximilian Schiedermeier, for their help, useful discussions and and exchange of interesting ideas. I also would like to thank the students that I co-supervised and who collaborated on some of my research projects: Winnie He, Laetitia Fesselier, Taha Salman, Trung Thien, Jifeng Wang, Petar Basta, Yunjia Zheng, Helen Kulka, Aaron Lohner, Ruoyu Wang and Xinchen Hou. I am thankful for the quality work which they have done for validating and implementing some of the ideas and concepts presented in this thesis.

A special thank to all of our system support staff, especially Andrew Bogecho. Andrew is an amazing and very helpful guy. He facilitated access to all computing resources I needed for my research projects and timely addressed any system issues, which was extremely helpful throughout my work.

I want to express my gratitude to the organizations which provided me with funding to pursue my doctoral studies: the Egyptian Cultural Office in Canada and Natural Sciences and Engineering Research Council of Canada (NSERC), as well as the funding that my advisor provided me with. Without these sources of funding, realizing this thesis would not have been possible.

I also thank my friends for their unfailing support and continuous encouragement. I should not forget to thank my family, especially my mother, Nagah, and my sisters for their constant moral support, help, availability, affection, patience and hope they had given to me. Without that hope, this thesis would not have been possible.

Last but not least, I want to express my deep appreciation and gratitude to my husband, Nooreldin Salama, who enormously supported and encouraged me throughout this long academic road and in my life in general. I am grateful for his patience, understanding and coping with sometimes (often) long working hours towards the completion of this thesis. I just could not be able to finish this thesis without his priceless help and support. Finally, I'm very grateful to my two kids, Farida and Yusuf, who provide unending inspiration and were patient enough of being even grumpier than normal whilst doing my PhD study.

Contents

List of Figures

List of Tables

1 Introduction

Many application domains have started to move their services into the cloud, such as e-commerce, health management, education, entertainment, and many more. The performance of these applications has a direct impact on revenue and customer satisfaction. For example, Google loses 20% traffic for each additional delay of 0.5 second to their page-load time and Amazon loses 1% of revenue for every 100 ms in latency increase [JKW17]. Overall, effective use of logging and monitoring is an important part of administrating the cloud applications, keeping track of their performance, and to detect, diagnose and resolve performance-related problems.

1.1 Motivation

Cloud applications often follow a distributed service-oriented architecture for modularity, scalability and reliability purposes.

Figure 1.1 illustrates a basic example of a multi-tier architecture with a front-end

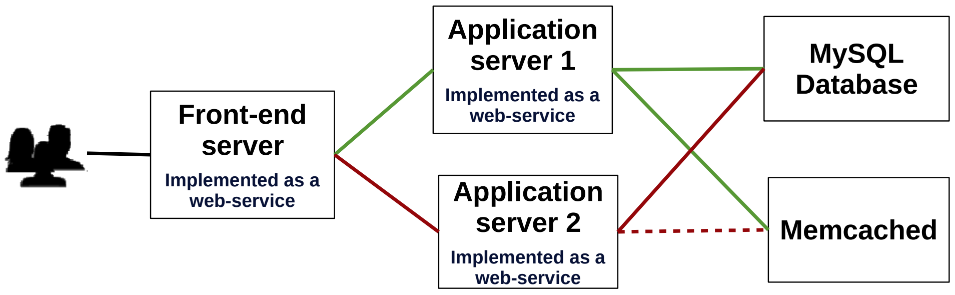

Front-end server
Implemented as a web-service
Application server 1
Implemented as a web-service
Application server 2
Implemented as a web-service
MySQL Database
Memcached

Figure 1.1: Example of a multi-tier application architecture.

server, application servers, Memcached[1]-based caching service and a MySQL[2] database for persistence. Each component normally runs on a dedicated virtual machine (VM) or container, or may be replicated on a cluster of machines to support higher demand. The front-end web server receives client requests and distributes them to the application server replicas, where the processing logic is stored. As such, front-end servers also often serve as a load balancer. Both types of components are often implemented as HTTP-based web-services. Processing client requests can involve data retrieval from different data sources such as databases and/or caches, in the above example a MySQL database and a Memcached caching service. Cache systems are used to store the most frequently accessed data in main-memory in order to reduce latency. However, more and more applications are further divided into smaller components with tens of microservices where each of them is typically a HTTP-based web-service and internally implements a particular functionality such as an authentication or a recommendation service. These architectures have further expanded to include replicated and/or distributed services for scalability and reliability, e.g., a multi-node Cassandra key-value store [Cas] or distributed computational services such as Spark [Spa].

Performance analysis of such distributed applications is complex as there are many possible causes for performance anomalies. Over time, a set of measures have been shown to be useful and are provided by many monitoring tools. A common first step to get insight into the performance and potential bottlenecks is to measure hardware resource consumption in terms of CPU, memory, I/O and network. Many cloud monitoring tools use the unit of a virtual machine on a hypervisor to observe the behavior of individual components [dCRCG+16].

[1]`https://memcached.org/`.
[2]`https://www.mysql.com/`.

On top of this, application-layer measures such as throughput, request service time, message size, or characteristics of service call distributions are also important to understand application performance issues, facilitating resource management and help troubleshoot faulty applications. For instance, considering the distributed application illustrated in Figure 1.1, if application replica 2 is not using the Memcached service at all or less than expected (e.g., due to misconfiguration), this will cause significantly higher response times for a subset of the requests since they need to be processed by the database instead of the much faster cache. This kind of performance anomaly will not be detected by looking at resource consumption unless the database is overloaded [LW15]. A similar load on CPU, memory, and the network would be reported for each application replica as they all do the same task – receiving an input request and then making a network call to either MySQL database or Memcached server. However, this kind of performance anomaly could be detected when looking at component throughput and response time between each component as it would reveal higher message throughput between the application replica 2 and the database server compared to the one to the cache server, and a longer request service time between the front-end and that application replica compared to the application replica 1. Thus, end-to-end application-layer performance metrics as well as their breakdown within and across components/tiers can help an administrator diagnose application performance issues.

Furthermore, having a global view of the distributed application structure and dependencies during run-time is also crucial [WZXG20]. With global view we mean a visualization of how the different components invoke each other in form of a call graph [J. 17]. Ideally, a monitoring tool should be able to automatically generate during run-time a graph that looks similar to Figure 1.1. It should not only include the structure and dependencies among the components but also show the service type of each component, e.g., that the component is a MySQL database or a HTTP-based web-service. Although the overall architecture of an application is commonly described in some sort of documentation, as applications evolve, new dependencies might be created that are not reflected in the documentation. In addition, the actual inter-component dependency often varies over time according to the workload. For example, replicas to the individual application components might be added or removed or certain components might not be active in a given setup. For instance, a run-time visualization of the example application in Figure 1.1 shows there are two instances of the application server, while a documentation might indicate only one application server. Furthermore, a system can be configurable with various database systems, while the run-time visualization of the application call graph will show the one actually used. Thus, real-time call graphs are needed [ERR18].

Overall, combining the run-time application call graph with the component-level performance metrics such as throughput and response time provides a holistic application monitoring platform that can help the application administrator determine when particular components become the bottleneck, or when misconfiguration causes outages such as the caching service misconfiguration described above, as well as manage application components far more efficiently.

Having such a global view of the application at run-time is not only beneficial to the application administrators but also assists a cloud provider in placing applications properly; for example ensuring that two database replicas are not put on the same server for more reliability, or that a web-server is close to the caching service it calls frequently. Hence, clouds must become "distribution-aware" so that they can deduce the overall structure and dependencies within a client's distributed application and use that knowledge to better guide monitoring and management services.

For all the aforementioned monitoring features, many monitoring solutions exist [ama, Goo22, SBB+, Zip, WEG, HvH20, LTRW, ZWG+18, WGH+15, sys, MCSL17, LKK+19, HLZ+]. Most existing monitoring systems [ama, Goo22, SBB+, Zip, WEG, HvH20] provide such features through sophisticated application and/or platform instrumentation which are tightly integrated and therefore must be developed for each application resp. platform. Furthermore, they are mostly static, meaning that in order to enable monitoring, the system must be properly pre-configured or the application must be interrupted and restarted.

Interestingly, many of the application-specific performance metrics, dependencies and service type information can be inferred from the messages exchanged between the application components, and a variety of monitoring tools are thus based on such message exchange. By sniffing the message flows that a component exchanges with other components, they can deduce application performance metrics such as response time and throughput and also infer inter-dependency information. Some of them [Tsh, sys, HLZ+] exploit the programmability feature of new networking paradigms such as Software Defined Networking (SDN) [JMD14] and Network Function Virtualization (NFV) [MSG+16] to instantiate monitoring of application traffic flows on demand [LTRW] or integrate application performance analysis function into the network component that routes the application messages [ZWG+18, WGH+15]. Additionally, some work has shown that service and application types can be automatically inferred from the network flows through deep packet inspection (DPI) [LKC+16, SCP14], traditional feature-based machine learning techniques [ZXW+13], and more recently, deep learning techniques [MCSL17, LKK+19]. A strength of these network-based monitoring systems is that they do not require platform or application instrumentation. This is a strong advantage because they can support a wide range of service types without necessarily requiring a deep expert knowledge in all the different applications and platforms that exist or will exist in the future. This is crucial given the rapid changing market for cloud platforms. Furthermore, they can be instantiated at run-time and do not require the application to be interrupted and restarted. However, as far as we are aware of, none of these tools provides all of the application monitoring capabilities we are aiming at but rather focuses on specific functionality. That is, they do not represent full-fledged, integrated real-time application monitoring systems. Furthermore, the systems we explored all induced a significant overhead on the application and/or network components.

Given this context, the broad objective of this thesis is to explore how to provide a holistic and comprehensive *Monitoring-as-a-Service (MaaS)* solution that has the advantages that come with deriving all information from the network flows between components but overcomes the shortcomings of existing network-based work. Towards this end, this thesis proposes a holistic dynamic network-based application performance monitoring system that provide four main features: 1) It provides the communication structure of a running distributed application in form of a call graph, 2) it automatically derives the service type of each component, 3) it derives important performance metrics for each component as a whole and in regard to the other components it interacts with, and 4) it does all that with very low overhead and on-demand, that is, without interrupting the running application. Being a network-based approach, our *MaaS* solution does not rely on application or platform instrumentation or deep knowledge about specific services. The proposed solution allows for dynamic and flexible integration of monitoring and analysis functionality into the cloud infrastructure in a transparent way, serving both cloud and application administrators. We envision it to be offered by the cloud provider as a service, to check the health of the running applications and/or manage the cloud resources.

1.2 Thesis Contributions

The main contribution of this thesis is a dynamic holistic network-based application monitoring framework that decouples the monitoring functionality from the application platform and the applications themselves, and can be deployed on demand at run-time with low overhead. We do that by developing an efficient network monitoring function that runs at the software layer of the network. In contrast to many network-based solutions, it follows a loosely coupled approach that offers a flexible and adjustable integration of application monitoring and analysis functionality into the network with minimal overhead. More concretely, this thesis provides the following contributions:

Application monitoring in the network The first step of this thesis was to design a solution that collects the data at the communication layer and links it to the analysis with reasonable overhead. While there are network-based solutions that are located at the end-hosts where the components reside, intercepting the message flows right when they leave or arrive at the component, our focus was at looking how we can integrate the solution deeper in the network layer, by exploiting the characteristics and capabilities of software-defined networking. To this end, we have explored existing SDN-based monitoring approaches, some already used for application monitoring, others mainly designed for monitoring the network itself, and have evaluated their performance in terms of the impact on the network element and communication overhead. We have found two different existing approaches. The first is to instrument SDN components such as switches and routers such that they duplicate messages to forward a copy to a second destination that then performs the actual analysis [LW15,LTRW]. While it has the advantage of simplicity, the message traffic overhead can be significant. The second approach is to extend the source code of the software switch to have an inclusive analysis function [ZWG+18, MHM+14, CLKdR16, WGH+15, DFC+16]. However, the extra computation might have a negative impact on latency and throughput. Therefore, we advocate a novel network-based monitoring approach that strikes a balance between responsiveness, compute and message overhead. We have been inspired by the NFV concepts where the network functionality is executed within virtualized functions that are executed in the software but are still an integrative part of the network. We have adapted a conceptual similar strategy to allow for a flexible and adjustable amount of application monitoring and analysis functionality within the network. The main idea is to decouple the network monitoring function from the switch forwarding path by developing a virtual network monitoring function, that we refer to as a *sniffer*, that is only loosely coupled with the switch functionality, and runs on the host of the software switch. This sniffer inspects messages and is able to calculate performance metrics locally as needed. It can then send performance results and/or message summaries to an analysis server for aggregation. Such an approach works for software switches that run on general purpose hardware that also allows other processes to execute. In our comparative study, we have evaluated and compared our solution with the other approaches for calculating one specific application performance metric, namely response time. Our results show that our solution performs favorable in terms of impact on the monitored application, general resource consumption and communication overhead.

Monitoring as a Service (MaaS) prototype The second contribution of this thesis is the design and implementation of a possible architecture for a holistic Monitoring-as-a-Service (MaaS). The service needs to allow the administrators to specify the components and the performance metrics they are interested in. As such we have designed a multi-tier distributed architecture where the sniffers at the software switches serve as agents to collect performance metrics and communicate with an analysis component for aggregation and coordination. A front-end component interacts with the MaaS client to receive the monitoring requests and visualize the results in near real-time. We have developed solutions for a wide range of common application-level performance metrics, for instance throughput, response time, error rates and request types. Several of these performance metrics are independent of the service types but some require some deep message inspection- which we support so far for a range of services, for instance HTTP-based web-services, MySQL database and Memcached. We have designed our MaaS system so that new types of services and new performance metrics can be added in an incremental manner.

DyMonD In our third contribution, we have extended our application monitoring functionality to provide a further service, namely dynamic identification of application component call graphs. We have proposed *DyMonD*, a full-fledged network-based framework for dynamic holistic application level monitoring that: 1) infers the overall structure and dependencies between distributed components of the cloud applications, 2) identifies the service type provided by each application component, 3) and tracks component performance metrics such as throughput and response time. It can serve an arbitrary number of different services and platforms with low overhead using the sniffer approach. So far, our solution supports request/reply-based services as a proof of concept given their ubiquity usage by the cloud applications. *DyMonD* analyzes the packet traffic among components by observing relevant network flows. With the extracted information it can determine call dependencies and service types, and track component performance metrics. DyMonD employs a novel deep learning model to classify the service type of each component through the packet data. Therefore, it does not need to rely on static information such as service ports. DyMonD classifies particular software systems that are widely used by cloud applications, such as MySQL, Memcached, or web-service (based on HTTP). In contrast to many other deep-learning based service identification solutions, which rely on receiving the first few (handshake) packets of a connection, DyMonD provides excellent prediction results even if monitoring starts at a random time after connection setup. Furthermore, DyMonD yields high prediction accuracy for secured traffic.

In this prospective, this thesis also contributes with generating a large flow-based dataset for a wide range of commonly used service types in multi-component cloud applications, conducting a thorough analysis of using different deep learning architectures for recognizing them, examining various design parameters and demonstrating their trade-offs.

Furthermore, for micro-service architectures, which wrap each component as a web-service based on the HTTP protocol, DyMonD can optionally determine application-specific service types (e.g, "authentication" or "recommender") by performing a deep-packet inspection and applying natural language processing techniques.

1.3 Publications

This section lists the publications that are part of this thesis along with a brief of their contributions.

Published

[Els19] Monitoring as a service for SDN based cloud data centers.
Mona ElSaadawy.
Proceedings of the 20th International Middleware Conference (Middleware'19) (Doctoral Symposium), 2019.

In this paper, we outline the research problem and motivation of this Ph.D. thesis, as well as the foundation and initial design of our three contributions: Sniffer, MaaS framework, and DyMonD. This work is done by me under the guidance of my advisor.

[SKY] Enabling Efficient Application Monitoring in Cloud Data Centers using SDN.
Mona ElSaadawy, Bettina Kemme, and Mohamed Younis.
IEEE International Conference on Communication, ICC'20, 2020.

In this paper, we explore mechanisms to integrate application monitoring into SDN. In particular, we analyze whether switch-based message filtering is feasible and see whether the amount of messages that need to be sent to an external analysis tool can be reduced by exploiting the filtering capacity of SDN switches without impacting the switches' performance. Furthermore, we advocate a novel approach that strikes a balance between responsiveness, compute and message overhead. The main idea is to decouple the monitoring function from the switch forwarding path by developing a separate process that is only loosely coupled with the switch functionality, and runs on the host of the software switch. Such an approach works for software switches that run on general purpose hardware that also allows other processes to execute. This separate process allows for a flexible and adjustable amount of application monitoring and analysis at the software switch itself. We conducted an extensive performance evaluation and comparison of the approaches presented in this paper, as well as highlighted the corresponding trade-offs. As the main contributor of this paper, I designed and implemented the proposed monitoring approach and performed the experiments under the guidance of my advisor. Profs. Bettina Kemme and Mohamed Younis joined the discussion of the paper, provided feedback, and helped with the paper writing. The contributions made by this paper are covered in Chapter 3

[EFK21]	Application Monitoring as a Network Service. Mona ElSaadawy, Laetitia Fesselier, and Bettina Kemme. *41st IEEE International Conference on Distributed Computing Systems, ICDCS'21* (Demo), 2021.

In this demonstration, we demonstrate the design and implementation of a complete Monitoring as a Service (MaaS) framework that is built with the network monitoring function proposed in [SKY] as core building block. The MaaS framework follows a client-server model that provides a client interface, where a MaaS user can dynamically choose the components to be monitored and what performance measures they are interested in, as well as the monitoring duration/window. The current MaaS supports several service types and a wide range of performance metrics. In addition, we have designed the MaaS software in a modular way that allows for extensibility for new service types and performance metrics. As the main contributor of this paper, I worked on the basic design and concepts of the MaaS framework, and wrote the paper under the guidance of my advisor. Laetitia performed a considerable part of the MaaS implementation through a summer undergraduate project and a follow-up research course[3]. The contributions made by this paper are covered in Chapter 4.

[SBZ$^+$]	Flow-based Service Type Identification using Deep Learning. Mona ElSaadawy, Petar Basta, Yunjia Zheng, Bettina Kemme, and Mohamed Younis. *IEEE International Conference on Network Softwarization, Netsoft'21*, 2021.

[3]COMP400 report:`https://www.cs.mcgill.ca/~lfesse/doc/Application%20Monitoring%20As%20A%20Network%20Service-v2.pdf`

1.3 Publications

In this paper, we have provided a comprehensive study of the use of deep learning models for service type identification of network flows considering various service types that are commonly used in multi-component cloud applications. We have compared the performance of various deep learning models, while using header-based and payload-based data for training. We have highlighted the trade-offs and the impact of various parameters on the classification performance and required model training time. As the main contributor of this paper, I worked on generating the dataset from pre-collected network traces, implemented the deep learning models along with their optimization algorithms, run the experiments, and wrote the paper under the guidance of my advisor. Petar and Yunjia collected the network traces for the considered service types. This includes the development of three Java applications to use those service types. Prof. Mohamed Younis helped in the paper editing and reviewing process. Part of the content of Chapter 5 of this thesis is derived from this paper.

[ELW+21] DyMonD: Dynamic Application Monitoring and Service Detection Framework.
Mona ElSaadawy, Aaron Lohner, Ruoyu Wang, Jifeng Wang, and Bettina Kemme.
the 22nd International Middleware Conference (Middleware'21) (Demo), 2021.

In this demonstration, we demonstrate the design and implementation of DyMonD, a holistic framework that dynamically monitors the software layer of the cloud network to track dependencies between application components and derive performance metrics. It adapts a novel deep learning model to identify the service type of each component, and visualizes all information in form of a call graph. Our evaluation results confirm that DyMonD can infer the proper call graph and identify the services at run-time with acceptable overhead and good accuracy. As the main contributor, I worked on the design and implementation of DyMonD, and wrote the paper under the guidance of my advisor. Aaron, Ruoyu and Jifeng worked on the integration and communication between DyMonD's individual components as well as the visualization frontend through undergraduate research projects[4]. The contribution made by this paper are covered in Chapter 6.

[4]DyMonD source code: `https://github.com/a-a-lohn/DyMonD/`

Under revision

[SYWK22] Dynamic Application Call Graph Formation and Service Identification in Cloud Data Centers.
Mona ElSaadawy, Mohamed Younis, Jifing Wang, and Bettina Kemme.
IEEE Transactions on Network and Service Management (TNSM), 2022.

This journal paper extends the previous Middleware demo paper about DyMonD by presenting the details of DyMonD's design, its individual components as well as the performance evaluation. This work was done by me including writing the paper under the guidance of my advisor. Prof. Mohamed Younis helped in the paper editing and reviewing process. Jifeng helped in editing the visualization tool used by DyMonD through an undergraduate research project. The contributions made by this paper are covered in Chapter 6.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 presents some relevant background, as well as relevant related work.

Chapter 3 provides first details about various approaches that can be used to dynamically collect application information at the network layer with corresponding limitations. We will then present our network-based application data collection approach and its contributions towards limiting the monitoring overhead while providing flexibility in integrating the monitoring and analysis functionality into the network.

Chapter 4 presents the design and implementation of a complete multi-tier Monitoring as a Service (MaaS) architecture that is built with the network monitoring function proposed in Chapter 3 as core building block. We will illustrate the application monitoring capabilities of our MaaS prototype and discuss its extendability feature that allows for monitoring new services and performance metrics by exploiting its modular implementations.

Chapter 5 provides a thorough analysis of the different deep learning approaches and architectures that can be employed to classify the various service types that are commonly used in multi-component cloud applications. We demonstrate the trade-off of various deep learning approaches and analyze their sensitivity to crucial design parameters in terms of service identification performance and training time overhead. Additionally, we introduce two novel deep learning architectures which adapt Bidirectional deep learning models that outperform the other models in terms of service identification accuracy with reasonable overhead.

Chapter 6 presents *DyMonD*, our holistic application monitoring framework that infers the application dependency information, labels the application components with the service type it provides, tracks component-level performance metrics, and then visualizes all this information in the form of a call graph. It employs our network-based monitoring approach as well as the proposed Bidirectional deep learning model. We also show the additional fine-grained service identification feature that DyMonD offers for microservice-based architectures.

All chapters describe main contributions, present our concrete solution, describe its implementation and present a detailed performance evaluation that shows the feasibility and relevance of the solution.

Chapter 7 provides the conclusions of the thesis and outlines some of the future research directions in the context of Monitoring-as-a-Service.

2 Background and Related Work

This chapter provides background in cloud networking, cloud management and monitoring approaches as well as related work pertinent to this thesis. In Section 2.1, we take a brief look at software-driven networking paradigms such as SDN and NFV, and their characteristics that enable the dynamic management and provision of today's cloud network infrastructure. We then show in Section 2.2 the basic architecture of cloud data centers and how distributed applications are deployed inside them. We then discuss the basics of performance monitoring in Section 2.3. We present the set of performance measures that are usually tracked in today's cloud data centers to ensure the health of the cloud infrastructure and cloud applications. This includes system, network and application layer measures. We then introduce in Section 2.4 various approaches used for network monitoring, with emphasis on the ones that adapt SDN and/or NFV to provide agile monitoring mechanisms. Section 2.5 focuses on application-layer performance monitoring which can be either done by instrumenting the application/platform at their running host, or by analyzing the network messages that are exchanged by the components of the distributed application. We discuss the advantages of network-based performance monitoring techniques that motivate us to employ it as a best candidate for the Monitoring-as-a-Service solution we aim at. Finally, in Section 2.6 we have a more detailed look at the area of network traffic classification as it is highly relevant for service identification in our application monitoring context.

2.1 Networking Paradigms

2.1.1 SDN

Software Defined Networking (SDN) [JMD14] is a relatively new computer networking paradigm providing a fundamental shift in the way network configuration and real-time traffic management is performed. To motivate SDN, we first describe the functionality of networking devices and the traditional way to configure them. From there, we discuss the limitations of this traditional configuration approach that call for a new network configuration paradigm such as SDN.

Traditional Networks

Networking devices such as switches and routers are the building blocks that enable communication between entities on a network. Enabling communication means anything that helps data get from source to destination. For example, a network switch is a multi-port device that receives messages on its input ports and forwards them to its output ports according to given forwarding rules. Those forwarding rules use the received packet information such as the destination IP address and map it to an action such as forwarding the packet to an output switch port or even dropping it. For example, a forwarding rule could be that any packet received from the switch port number X with destination IP of $A.A.A.A$, should be forwarded to the switch port number Y. These forwarding rules represent the control plane of the networking device, while the actual process of relaying the messages represents the forwarding plane.

Traditionally, networking devices have been developed by manufacturers. Each vendor designs their own firmware and other software to operate their own hardware in a proprietary way [KRV+15]. Thus, the forwarding rules are configured and deployed into the hardware of the networking devices through the proprietary software as illustrated on the left hand side of Figure 2.1, leading to a tight coupling of control and forwarding planes. This traditional approach of manual configuration of networking devices is cumbersome and error-prone for large networks. In addition, it significantly increases the complexity and cost of network reconfiguration required whenever new services, technologies or hardware are to be deployed within existing networks, or to implement network optimization algorithms such as traffic prioritizing, access control, and bandwidth management that are used to achieve the required Quality of Service (QoS) defined for the running applications.

Being aware of these limitations, the networking research community has worked on

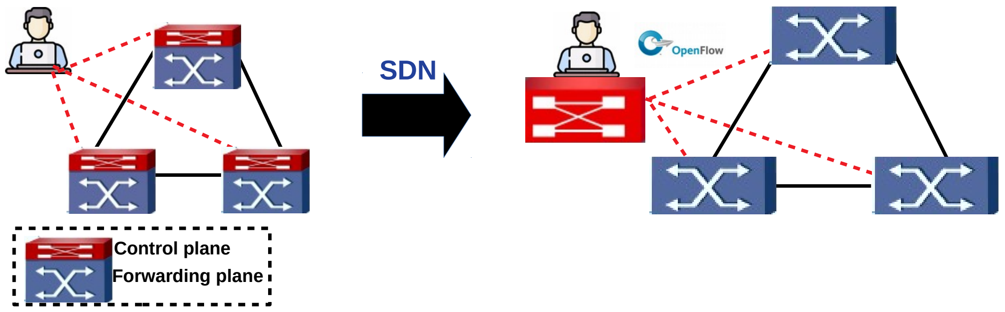


Figure 2.1: Traditional and SDN based networking.

analyzing the design of traditional networks and proposed abstractions that allow for easier and better network configuration. Thus, proposals for a new networking paradigm, namely programmable networks have emerged.

SDN-based networks The SDN framework is one of the programmable networks proposals. It centralizes the control plane in a piece of software decoupled from the networking device hardware (i.e. the forwarding plane) as illustrated on the right hand side of Figure 2.1. This allows network control to become directly programmable via an open interface (e.g., OpenFlow [MAB+08]) and the underlying infrastructure to become simple packet forwarding devices that can be programmed. This programmability can be used to automate network configuration. Thus, SDN is a key element that is deployed in large data center networks to provide a quick response to the dynamic change in the networking requirements, while eliminating the manually intensive regime of fine tuning individual hardware components.

The SDN framework consists of two main components: *SDN controller* and *SDN switch* [JMD14,KRV+15]. The SDN controller is a software-based entity that represents the control plane in the SDN framework. The SDN controller provides a set of rules to the SDN switches, indicating how to forward the different flows through the network. A network flow is defined as the packet stream between a source and a destination that is typically identified by a set of packet header fields, including layer 2-4 (i.e. data link, network and transport layers) packet information such as IP address, port number, protocol type, and other information. As a result, the SDN controller can customize how to route individual flows using its own application logic. OpenFlow – the de facto standard of SDN – is an API used for exchanging control messages between the controller and the switches.

A SDN switch could be hardware- or software-based, and has one or more flow tables – configured by the SDN controller through the OpenFlow API – which contain matching fields to match incoming flows' packets with certain actions such as prioritization, queuing, packet forwarding and dropping. When a packet reaches a port on the SDN switch, the switch performs a lookup in the flow tables for a flow entry that matches the packet header characteristics, such as destination IP address, and executes the set of actions defined in the matched flow such as forward the packet to the switch port that is connected to the target destination, duplicate the packet to another destination such as an analysis tool, or even drop the packet. In the case of no match, the SDN switch forwards the packet (or just its header) to the controller to request a new flow rule for the un-matched packet. In addition, each flow rule has some counters, such as the flow's number of packets and bytes, which are recorded and updated based on the matched packets.

The following is an example of a flow table rule:

TCP protocol, Source IP= A.A.A.A, Source Port= X, Destination IP= B.B.B.B, Destination port=Y, TCP Flags=ACK, Actions=output to out1, out2.

A.A.A.A and B.B.B.B. are the IP addresses of the source and destination communication entities, respectively, and out1 and out2 are the switch ports connected to the targeted destinations. In addition, the counters for the number of received packets and bytes are updated accordingly, whenever a packet is handled that sets the flow rule to true.

2.1.2 NFV

Network functions such as packet switching, intrusion detection, load balancers and firewalls are traditionally implemented as custom hardware appliances, where software is tightly coupled with specific proprietary hardware. Network function virtualization [MSG+16] has been recently proposed to provide more agile networks, with significant savings for operation and capital expenses, by leveraging the virtualization technology to design, deploy and manage network functions and services. This means that network functions – including packet switching – can be implemented as an instance of plain software that is decoupled from the underlying hardware, and running on standardized compute nodes. OpenVswitch (OVS) [KAB+14] is one of the widely used virtualized software switches in the networks of today's cloud data centers. Various cloud computing platforms and virtualization management systems have integrated software switches such as OVS, including OpenStack [Lam14], openQRM [Ope], OpenNebula [Llo] and oVirt [Ovi].

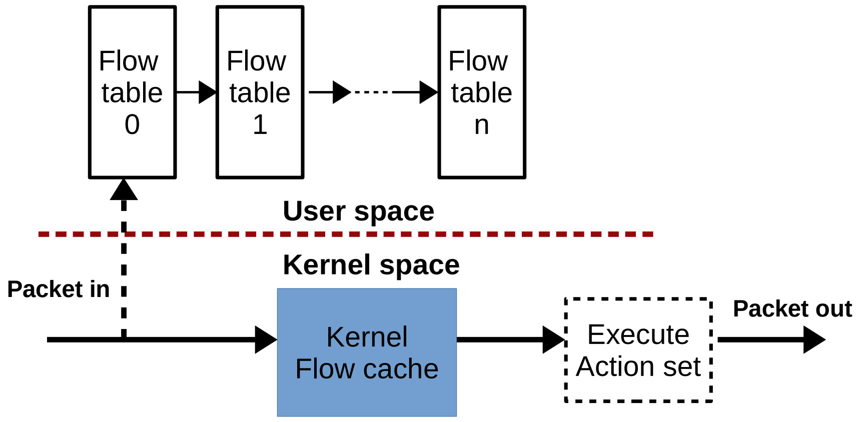


Figure 2.2: The OVS architecture.

Software switch design and performance A software switch such as OVS typically consists of a kernel space and a user space as depicted in Figure 2.2. They work together to forward packets, with the user space being a full (but slow) set of forwarding rules while the kernel space serves as a cache consisting of the subset of recently matched flow rules so that active flows can be directly processed in the kernel. In particular, incoming packets are first matched against the flow rules in the kernel space. If no match is found to a packet, it is copied to the user space, and the flow matching process is executed in the user space. The relevant rules are then cached to the kernel space. Due to the locality of the network traffic, most packets should be processed in the fast kernel path. However, the kernel space has a limited number of flow entries due to the memory size limitation.

Software switches have some performance issues due to memory copy and context switching between the kernel and user spaces. To optimize the performance of software switches, a set of libraries and drivers for fast packet processing is used such as the Data Plane Development Kit (DPDK) [DPD]. With DPDK, a software switch can copy packets to its user space with no kernel intervention which accelerates packet processing workloads running on a wide variety of CPU architectures.

NFV deployment In a typical NFV deployment, a network operator sets up an NFV infrastructure (NFVI). NFVI consists of computing nodes and network resources that host the virtualized network functions (VNFs). In addition, the network operator obtains VNFs from vendors that build the network function software, and installs the VNFs on the NFVI using the NFVI infrastructure manager (VIM), which controls the allocation of resources for the VNFs [MSG$^+$16]. OpenStack [Lam14] is an example of an open source VIM, controlling the physical and virtual resources for VNFs. Finally, the network is configured to correctly forward the packets along the sequence of VNF components through which a request should flow (i.e. service chain). An example of a network policy could be to require "any web traffic targeted to web-server WS to first go through the firewall and then the load balancer VNFs".

NFV vs. SDN NFV and SDN have a lot in common since they both leverage automation and virtualization to achieve agility, cost reduction, dynamism, and automation. However, SDN and NFV are different concepts, aimed at addressing different aspects of a software-driven networking solution. NFV aims at abstracting the *network functions* by decoupling network functions such as the network switch from specialized hardware elements, while SDN abstracts the *network* by separating the control logic of the networking devices from the networking device itself.

In fact, NFV and SDN are highly complementary, and hence combining them in one networking solution may lead to greater value. SDN can accelerate NFV deployment by offering a flexible and automated way of chaining network functions, while NFV is able to support SDN by providing the infrastructure upon which the SDN software can be run. For instance, the SDN controller, as a network function, is commonly implemented as pure software which runs on commodity servers, while the SDN switch can be implemented either in software running on commodity servers or as specialized hardware.

In our work, we heavily depend on VNF as a common infrastructure as we exploit the software switch to extract relevant message information for application monitoring. Furthermore, our overall MaaS architecture was inspired by NFV and SDN. While the final implementation is quite different, our application monitoring functions have a conceptual similarity with NFVs as they dynamically capture and analyze certain message flows as we will show later in Chapters 3, 4, and 6.

2.2 Cloud Architecture

A typical cloud data center architecture today consists of a 2-3 layer tree of switches and/or routers, that connect the physical machines that are distributed in racks and that host the application components, such as shown in Figure 2.3. Many of today's cloud data centers already deploy SDN and NFV technology for their network. According to Cisco Global Cloud Index report[1], SDN and NFV traffic volume is already making up 50% of the data traffic within today's data centers. Hardware packet switches are being used in the core

[1]`https://virtualization.network/Resources/Whitepapers/0b75cf2e-0c53-4891-918e-b542a5d364c5_white-paper-c11-738085.pdf`

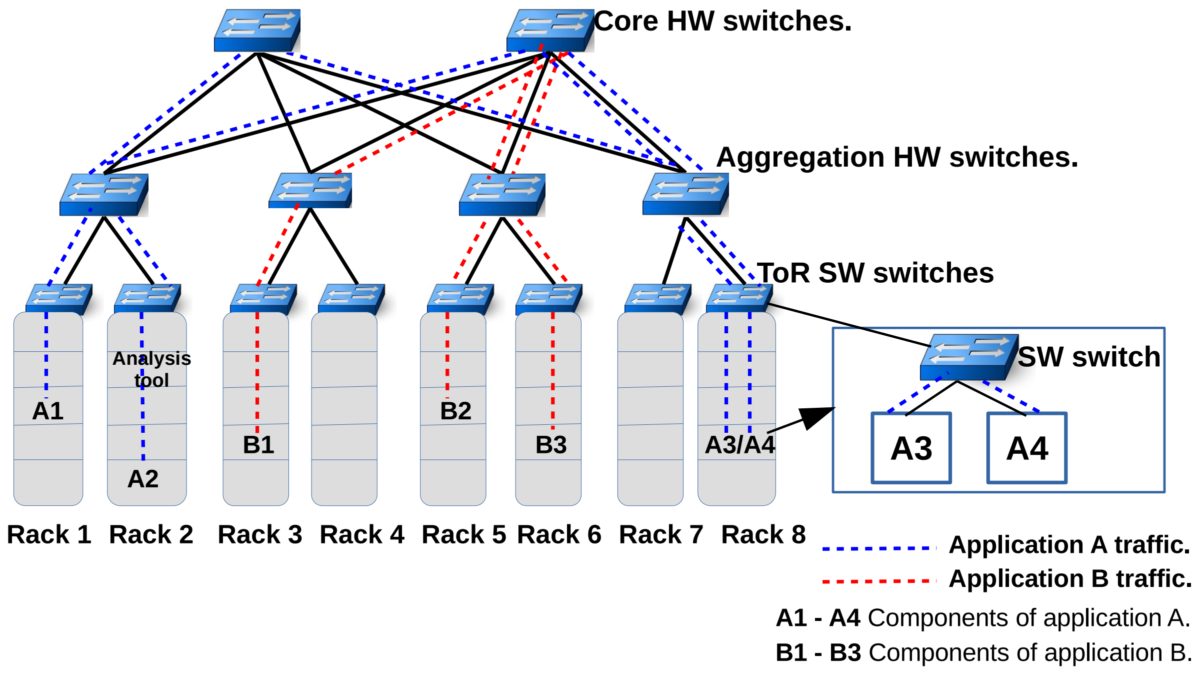


Figure 2.3: Example of cloud architecture

network where low latency is a must. The lower levels often deploy software switches. For instance, optimized software switches (e.g., OVS integrated with DPDK) are largely deployed as top-of-rack (TOR) switches [HRW15, CAR, Riz12, MAR$^+$14].

The leaf nodes in the different racks might host one or more application components. Virtual environments such as virtual machines or containers allow several components to run on the same physical host. Software switches are commonly used in such high-end machines that host many application components to facilitate the communication between the co-located application components as well as with the outside network.

The application components are distributed over the leaf nodes in the different racks. Figure 2.3 shows two example applications A and B, with components A1 - A4 and B1 - B3, respectively. Some or all of the application components could also be co-located on a single physical machine. Nevertheless, the communication between the application components passes through one or more switches. Figure 2.3 shows an example with components A3 and A4 on the same machine connected through a software switch that routes messages exchanged between them, which in turn is connected to the ToR switch to enable communication with the outside network as well.

2.3 Performance Monitoring in the Cloud

In this section, we first outline the basic monitoring aspects required for ensuring the health of the cloud infrastructure as well as the running applications. Then, we will discuss each monitoring aspect and the corresponding literature in separate sections, with highlighting the potential role of SDN & NFV in providing dynamic cloud application performance monitoring solutions.

Effective monitoring of the overall health of the cloud infrastructure and of individual applications requires data collection from the infrastructure and application components to get useful performance indicators. This includes system, network and application layer measures.

System layer measures track the health of the physical nodes as well as the individual containers and the virtual machines by collecting the state of hardware resources such as processor, memory, hard drive and ingoing and outgoing network traffic. Having information about the utilization of these hardware modules helps the cloud administrator to ensure that no physical host becomes a bottleneck or is unnecessarily under-utilized, and that all application components are assigned the physical resources that were agreed on. For the application administrators, knowing the resource utilization by each of the components helps in detecting bottlenecks within the application and guides decisions such as whether individual components need to be replicated or require more resources [dCRCG+16]. The performance and utilization of system resources can be measured by deploying monitoring agents at the operating system level or the hypervisor level of the individual physical machines. There exist plenty of monitoring tools that provide this in an efficient and transparent manner [ama, azu, Goo22]. Thus, our research does not focus on such system resources monitoring.

Monitoring the network is also crucial for cloud management. Network layer measures such as utilization of the individual links of the network hierarchy, message throughput and message delay on individual links and within parts of the network, and end-to-end packet delay and packet loss rates allow the detection of bottlenecks within the network infrastructure and help guide the cloud administrator in distributing applications across their infrastructure such as to avoid overload of some parts of the network and to guarantee service level agreements (SLAs). For instance, in Figure 2.3 assume that the ToR switch of the rack to which web-service A3 is connected is overloaded and thus starts to drop packets that are sent to A3. In return, the components communicating with A3 will resend these dropped messages which will cause a delay in processing and potentially also a further overload of the switch. There are many approaches to monitor the cloud network and diagnose network performance anomalies. We have been inspired by these approaches to see whether they can also be used for application monitoring. Thus, we will discuss them further in Section 2.4.

We have already outlined the importance of application layer performance for both cloud administrators as well as application administrators in the introduction (Chapter 1). We have identified that application specific performance metrics such as throughput and response time, knowing the call graph of distributed applications at run-time, and knowing the specific service type of the individual components are all important to monitor the health of an application. For instance, they allow application administrators to determine bottlenecks when they see that response times of individual components are higher than anticipated or to detect failures in the execution workflow or misconfigurations when the call graph does not follow the expected structure. And they allow cloud administrators to ensure that the assignments of components to physical hosts does not cause unnecessary message delay or congestion in the network, or lead to loss of reliability if component replicas were to be collocated on the same physical host. This thesis focuses on such application layer performance monitoring. We will discuss existing solutions to application monitoring in detail in Section 2.5. Furthermore, in Section 2.6 we focus on solutions for network traffic classification that can potentially be used for the service identification that is needed for application monitoring.

Whether it be system, network or application monitoring, it typically consists of two tasks. The first task is to capture and collect the relevant data, be it from the hosts, the application components or the network. This task is concerned about what, where and how to capture and collect performance related data. The second task is to aggregate and analyze the collected performance data to create useful performance metrics and present them to the administrators. This data aggregation and analysis can be performed by dedicated analysis nodes which requires the collecting components to send all collected data to such analysis nodes. Alternatively, the collecting components can potentially perform the analysis locally, although some aggregation might be necessary at the end to consolidate information if there are several components that collect data. We will see in the following sections how existing solutions perform data collection and analysis.

2.4 Network Monitoring

Cloud providers perform extensive network monitoring to ensure the health of their network infrastructure. Network layer measures such as network latency, i.e. the time the network takes to deliver the network packets to their destination, throughput, i.e. the amount of successfully delivered network messages per time unit, and packet loss, i.e. the amount of the network packets that failed to reach their intended destination, are commonly collected to assess the network performance. There exists extensive research on how to monitor the cloud network to detect and diagnose network performance issues. We discuss a variety of these network monitoring proposals in this section.

Data collection at the host Monitoring network performance relies on capturing information from the network traffic and analyzing it. Commercial network sniffers such as Wireshark/TShark[2] or tcpdump[3] can be deployed at end-hosts to collect network messages exchanged over specific network interface(s) in real-time. For example, in Figure 2.3, TShark can be deployed at the end host running A1 to sniff the network interface connected to A1 and collect the network messages that are exchanged over it. Wireshark/TShark and tcpdump perform some local analysis of the collected network packets and extract relevant data, mainly the header information, and transform them into log entries. Most typically, these log entries are then written to files that can be further processed by an external analysis component. Additionally Wireshark/TShark provides itself quite sophisticated analysis functions that provide network metrics such as packet loss and number of transmitted packets/bytes over the monitored connections.

Message mirroring Alternatively, the network components such as switches can be instrumented to collect network messages for network monitoring purposes. For instance, *port mirroring* is a popular network data collection approach used for network monitoring purpose. *Port mirroring* is a switch configuration that instructs the switch to "mirror" all traffic that passes through specific switch port(s) to another switch port that is connected to an analysis component. For example, port mirroring can be configured for the rack 1 ToR switch in Figure 2.3 to mirror all network packets that are transmitted over the switch port that is connected to A1 to the analysis tool deployed in rack 1. The concept of port mirroring is quite simple and easy to configure, yet it imposes significant communication overhead as it mirrors all packets independently of content.

The *"match"* capability in commodity switches, i.e. in both traditional and SDN switches, can help in reducing such communication overhead by filtering out certain flows/packets to be collected from the network and mirrored to a remote analysis node. Pre-defined rules, similar to the forwarding rules, are used to match the network packet header information and then certain actions are executed such as mirror to a remote analysis tool. This approach avoids sending possibly large and mostly irrelevant packets to the analysis tool. For instance, EverFlow [ZKC+15] utilizes such "match and mirror" capability in commodity switches to

[2]TShark `www.wireshark.org/docs/man-pages/tshark.html`
[3]Tcpdump `http://www.tcpdump.org/`.

capture certain TCP control packets to debug Data Center Network (DCN) faults such as link latency and packet loss.

SDN has shown to have promising features to provide support for dynamic network performance monitoring. In particular, SDN switches and controller are instrumented to collect and analyze network parameters. For instance, Detection-as-a-Service (DaaS) [MKK17] combines an intrusion detection system (IDS) with SDN programmability features to passively detect anomalies in the traffic that passes through the SDN network and to prevent malicious traffic from flowing into the SDN network. In the collection phase, DaaS instruments SDN switches to mirror any packet that arrives to the first flow path switch and forward it for analysis to a DaaS node where an IDS instance is running. IDS is programmed to identify patterns in the flows it receives that may indicate a network attack. During the analysis phase, the DaaS node in turn will decide whether the flow traffic is malicious or normal. A network reconfiguration will be needed if the DaaS node tags the traffic flow as malicious in order to block it. The malicious flow will be blocked by inserting a flow blocking entry into the switch flow table with the help of the SDN controller. Other related approaches can be found in [GAM15, CMLX15].

All mirroring-based approaches share three main limitations. First, the mirroring rules are coupled with the forwarding rules and are executed at the same time the messages are processed by the switch. This may lead to processing overhead and thus, delay in routing the message. Second, flow mirroring rules only apply to the packet header; a deep inspection is not possible. Thus, in some cases one might have to send more packets than actually necessary as filtering at the header-level might be quite coarse-grained. Furthermore, the switch performs a set of actions that are limited to packet forwarding and/or dropping. Thus, the collecting components, i.e. the switches, are only capable to collect and forward the network data to a remote analysis node without having the ability to perform any kind of analysis locally, which also leads to message overhead in the network as the switch has to send all the packet data to an external analysis node, while a local analysis capability may enable extracting and forwarding only the needed information from the packet to the analysis node.

Switch enhancements To address limitations of mirroring, some research work extends the switch functionality to decouple the monitoring rules from the switch forwarding path, support inspecting the packets based on higher layer information such as the packet payload and/or executing customized actions. Most of this research work utilizes the software-driven networking technologies such as SDN and NFV to provide such extended switch functionality. Both [WGH+15] and [ZWG+18] embed network monitoring function inside the OVS software switch by modifying the OVS source code, while attempting to decouple the monitoring functions from the forwarding path of OVS in different ways. UMON [WGH+15] decouples monitoring from forwarding by defining a monitoring flow table in the user space of the OVS software switch, while Zha el al. [ZWG+18] extend the kernel space of the OVS software switch to buffer the monitored packets into a ring buffer where they can be picked up by the monitoring process. Both approaches are meant for collecting network flow statistics such as the byte counts for specific flows, that can afterwards be pulled by the analysis node for performance monitoring purposes. However, these proposals still negatively affect the forwarding latency of the OVS as the matched packets must be copied to the monitoring flow table in [WGH+15] or to the ring buffer in [ZWG+18].

Several projects have focused on extending switch software to provide monitoring functionality. In particular, they often provide deep packet inspection for the network packets [CLKdR16] or define new actions for the matched packets [FDN14, Ham14, MHM+14]. For instance, the authors in [CLKdR16] extend the OVS software switch architecture to inspect not only the packet header but also the payload information in the packets to detect network security attacks. They achieve that by extending the software switch code to have a deep packet inspection module that inspects the packet payload against a set of predefined string patterns. Those string pattern rules are inserted into the switch at the time of the switch initialization. Logs of matched packets are generated and sent to a log server for further analysis. In contrast, [Ham14] extends the software switch code to enable the software switch to perform user-defined actions and analyze received network packets to detect network security attacks such as port scanner detector, which looks for repeated attempts to connect to a closed port on a system (i.e. the victim) from another system (i.e. the attacker). Similarly, [MHM+14] augmented software switches with application processing logic defined in a table called application table. This table is similar in spirit to the OpenFlow flow table. However, the application table actions are customized to be either ordinary OpenFlow API functions or specific application functions. Therefore, more sophisticated and application specific functions such as firewall and load balancer can be generated locally within the switch by executing application table actions for captured packets. While none of these proposals considered application performance functionality, conceptually some of the performance analysis logic we are aiming at might also be realized by user-defined or application actions in such architectures. However, depending on the extra tasks to be performed, the switch performance might be negatively affected and it might now handle significantly less packets per time unit than without application-specific functionality.

The common disadvantage of all the switch enhancement-based proposals is that modifying the OVS base code is not a trivial task, specially for the kernel space modifications. Packet processing languages such as P4[4] have been developed to facilitate enhancements to the switch functionality [KCBH21]. They allow for a more convenient way to specify rules that determine when a specific packet should trigger actions and to also program more complex actions that include maintenance of data structures on the switch [KSK21, MFP+22]. However, as far as we are aware of, P4 has not yet been thoroughly evaluated. In addition, P4 is somewhat limited in its processing capabilities as it uses fixed length data structures, which make it more challenging to perform a sophisticated deep packet inspection.

Without the need of message mirroring, the SDN switch and controller characteristics can also be exploited for measuring network performance parameters. For example, OpenNetMon [vADK14] measures the network throughput and packet loss by pulling the flow's sent bytes counters from the SDN switches. Furthermore, it uses the programmability feature of the SDN controller to embed flow rules directly into the SDN switches to inject probe packets that travel the same flow paths to calculate the path latency. In this scheme, the SDN switches are the collectors for the performance data, while the controller performs the aggregation and analysis tasks of the collected data to measure the network performance. In [SMX+15], in-network monitors are embedded along the forwarding path of the network packets to embed a tag into the packet headers for network monitoring purposes such as determining path latency and packet loss. These in-network monitors can be separate nodes or co-reside with the forwarding plane of the SDN switches along the packet traffic path. In other words, these in-network monitors are deployed as VNFs and then configured by the SDN controller to be part of the service chain of the network packets. The SDN controller also configures the marking actions for the network packet at these in-network monitors according to the monitoring requirements. When the network packets arrive the last monitoring agent before their targeted destination, the packet tags are forwarded to the SDN controller for further analysis.

In a nutshell, a considerable amount of network monitoring and analysis is already taking place in the network. This raises the question of whether the network components could also be used for application monitoring and analysis. We will discuss in the next section how some of network traffic analysis approaches discussed in this section can be adapted for application performance monitoring.

[4]P4 https://p4.org/

Tool	Performance metrics	Application call graph	Dynamic service identification	Microservice identification	On-demand
Software instrumentation-based approaches					
Twitter [LLL$^+$12], Google Cloud Monitor [Goo19, Goo22], and [SAR14]	✓	✗	✓	✓	✗
Facebook [CMF$^+$14]	✓	✓	✓	✓	✗
OpenZipkin [Zip] and Jaeger [Jae]	✓	✓	✓	✓	✗
Dapper [SBB$^+$]	✓	✓	✓	✓	✗
Kieker [HvH20]	✓	✓	✓	✗	✗
NewRelic [New]	✓	✓	✓	✓	✗
AppDynamics [App]	✓	✓	✓	✓	✗
SolarWinds [Sol]	✓	✓	✓	✗	✗
Datadog [Dat]	✓	✓	✓	✓	✗
Dyntrace [Dyn]	✓	✓	✓	✗	✗
Weavescope [Weaa]	✓	✓	✗	✗	✓
Network-based approaches					
NetAlytics [LTRW], [SMX$^+$15]	✓	✗	✗	✗	✓
PreciseTracer [SZL$^+$12]	✓	✓	✗	✗	✓
SmartRelationship [ZZZ$^+$]	✗	✓	✗	✗	✓
SysDig [sys]	✓	✓	✗	✗	✓
CAT [ENOP21]	✓	✓	✗	✗	✓
Net-Cohort [HSG$^+$]	✗	✓	✗	✗	✓
TopClass [HLZ$^+$]	✗	✓	✓	✗	✓

Table 2.1: Application monitoring tools for distributed applications

2.5 Application Monitoring

There exists a wide range of solutions that provide application-layer performance monitoring for distributed applications. In this section we provide an overview of existing solutions that provide one or more of the application monitoring functionalities we aim at, namely collecting high-level application measures such as throughput and response time, inferring the call graph of distributed applications and knowing the specific service type of the individual components at run-time. We have a close look at how they perform the monitoring, how and where they collect the relevant information, and whether they are able to start and stop application monitoring on demand and with zero interruption to the running application. We distinguish between software instrumentation and network based approaches as they are fundamentally different. Table 2.1 provides an overview of the different features provided by existing systems.

2.5.1 Software Instrumentation-based Application Monitoring

The most common way to extract application relevant metrics is through *software instrumentation*, which relies on an application and/or platform dependent data collection process and requires deep knowledge of the system. It typically creates explicit application and/or platform specific log entries that then allow to extract the high level measures such as request service time [LLL+12, Goo19, SAR14] or build call graphs [CMF+14, Zip, SBB+, HvH20]. A log message is a text string with an abundance of contextual information about events that occur during run-time. For instance, Twitter [LLL+12] instruments its code to generate structured *client event* log messages, that can be later processed by an analysis node to infer some application performance measures such as request service time. Alternatively, a software platform can incorporate a logging mechanism that can deliver the necessary metrics for the applications deployed on the platform. For example, Apache Tomcat uses its proprietary "Access Log Valve"[5] for collecting performance data and creating log files. Any request arriving at a Tomcat web application is passed to the access log valve process as well, where information about both the request and its response is collected and analyzed to calculate application-layer performance metrics such as request service time, which are then saved into access log files. Another example is Google Cloud's operations suite [Goo22] which uses both user-defined and platform logs to collect performance data that is analyzed by an analysis component.

Logging and tracing are also widely used as a data collection mechanism for analysis systems that infer call graphs of distributed applications. Many approaches trace individual client requests to build call graphs and determine end-to-end latency. For instance, Facebook [CMF+14] logs messages with request IDs for their individual services to construct the request execution graph. OpenZipkin [Zip] and Jaeger [Jae] are open-source tools that instrument various platforms to generate log traces and build dependency diagrams for the traced requests that show how they are processed by the components of the application. OpenZipkin is used by popular troubleshooting tools such as Edgar [Net], which tracks the request flows across Netflix distributed micro-services. Jaeger is combined with a Linux kernel tracer LTTng [DD08] to analyze the execution path of requests in [GEJD21]. Weavescope [Weaa] uses established container APIs (for example, the Docker API) to gather information about the containers that run on a specific host to build a topology of those containers. Google's Dapper [SBB+] instruments its RPC middleware to generate log traces and builds the execution tree of individual requests, while Kieker [HvH20] relies on monitoring probes within the components to provide these traces in order to construct a dependency graph for the instrumented components that shows how they invoke each other.

Furthermore, there is a set of commercial tools that perform *automatic instrumentation* to known frameworks and libraries in order to trace the application execution path. Examples include New Relic [New], SolarWinds [Sol], Datadog [Dat], AppDynamics [App] and Dyntrace [Dyn], which employ a sort of automatic instrumentation for each service deployed on the monitored host and show the relationships between services, processes and

[5]`https://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/catalina/valves/AccessLogValve.html`.

hosts, in addition to some performance metrics. Automatic instrumentation works by modifying the code at run-time or at compile-time to add tracing capability to the libraries and frameworks the application depends on [CFAI17, ASRC14]. The automatic instrumentation, installation and usage differ from language to language, depending on the capabilities of the language run-time. For example, for Java one can leverage the capability of the execution environment, i.e. the Java Virtual Machine (JVM), to instrument the byte code of Java classes, while languages such as Go require wrapping the existing libraries with instrumentation code. Automatic instrumentation injects small pieces of code before and after certain events, like HTTP requests and database queries, to measure their duration and collect metadata (e.g., the database statement as well as HTTP related information such as the URL, parameters, and headers) [ASRC14]. For example, auto instrumentation code could be injected just before and after the servlet invocation of any class that extends "javax.servlet.HttpServlet". Similarly, Fournier et al. [FEAD] add tracing points to PHP to get the start and end time of web requests in order to monitor the request service time.

Obviously, platform- and application-based instrumentation approaches can easily derive the type of service that each component provides, at various levels of granularity as they have deep access to the internal functionality of each component. That is, service identification is not really a challenge for this kind of application monitoring systems. The commercial tools that rely on automatic instrumentation also often know exactly with which service type they are working, as shown by the servlet example above. Some of them also access configuration files to determine the fine-grained information, e.g., that a particular HTTP service offers “authentication” [New, App, Dat].

The major disadvantage of all these approaches is that they involve sophisticated framework/application or language dependent instrumentation, and thus, require considerable efforts and engineering knowledge for each new framework that needs to be integrated. For example, Tomcat valves can not be used in a different servlet/JSP container and automatic instrumentation used for JVM can not be applied to Python-based applications. Also, most of these approaches need a restart of the application to activate monitoring, and thus, cannot perform monitoring on demand.

2.5.2 Network-based Application Monitoring

Interestingly, a considerable number of high-level application specific metrics can be obtained in an application/platform agnostic way by only looking at the application messages exchanged between components of the application. One example is request service time that can be calculated by capturing and matching outgoing response messages with their corresponding incoming request messages and taking the difference between the two capture times for each request-response pair as the request service time. Network monitoring tools such as Wireshark/TShark or tcpdump that we have already discussed in Section 2.4 can be used to observe and monitor the message exchange. They are deployed at the host and capture relevant messages in real-time by using message filtering at the network interface to the application layer. Thus, these tools cannot only be used to provide network-related statistics but also application-relevant data. In fact, Wireshark/TShark already provides quite sophisticated analysis tools for these application messages in order to calculate some application relevant performance metrics such as the request service time.

Network-based approaches are also used within large end-hosts that host many components that communicate with each other [J. 17, ENOP21, Weaa, SZL+12, HSG+, HLZ+]. Sieve [J. 17] uses the kernel module SysDig [sys] to collect the communicating paths of the different components as an event stream of system calls with some information about the monitored processes, in order to map processes to components and infer the links between them. CAT [ENOP21] employs system tracer tools such as ptrace [ptr] to trace network and storage-related system events (e.g., recvfrom, pwrite64), while extracting some content information from those events such as the number of sent/received or read/written bytes, to better correlate the events and detect errors in their data flows across the monitored application's components. Weavescope [Weaa] deduces dependency information between containers running on a specific host by monitoring the network of the container platform such as Docker[6] and Kubernetes[7]. In PreciseTracer [SZL+12], a TCP Tracer is deployed inside each VM to track communication and produce request execution path and performance statistics. Net-Cohort [HSG+] uses both packet sniffing and metrics correlation to discover links between VMs. TopClass [HLZ+] captures all packets transferred between VMs through the Linux netfilter table to derive the application call graph.

All the aforementioned approaches execute at the host. In most cases they sniff all the network flows which impacts the performance of the application, adds significant computational overhead and increases the analysis time. In addition, the monitoring process often interferes with the execution path of system calls [sys, ENOP21] or switch functionality [ZZZ+], which might negatively impact performance at high traffic rates.

There also has been some work that exploits the SDN and NFV functionality for application monitoring. For instance, NetAlytics [LTRW] deploys analysis nodes, called monitoring agents, across the cloud network and connects them directly to the TOR SDN switches. The SDN controller is then used to instrument the SDN switches to mirror their

[6]`https://www.docker.com/`
[7]`https://kubernetes.io/`

flows to the monitoring agents. The main focus is on analyzing traffic that arrives at the monitoring agent in real-time by developing specific parsers and a query language that allows to specify which ports need to be monitored and how.

Capturing the information about the network flows that go through the software switches can also be used to infer the application dependency information and deduce its call graph. For instance, SmartRelationship [ZZZ$^+$] uses tcpdump and/or flow information exporters such as *sFlow*[8] and *NetFlow*[9] to collect and forward the exchanged flow information, such as source and destination IPs and port numbers from the virtual switches to an analysis node to build the application call graph. However such flow exporters are limited to extract aggregated information about the exchanged flows and do not provide the packet level details required for complete application performance analysis (like the response time analysis etc). In addition, as the flow exportation process is tightly coupled with the switch, it might negatively impact the switch performance at the higher traffic rate as more flows are collected and forwarded to the analysis tool.

We have been inspired by these network-based application performance monitoring approaches because they do not require a deep knowledge about the running application/platform and can be applied in a wide range of settings. In our solution, we focus on approaches that involve the switches and are decoupled from the VMs, containers, and processes that run the components, as this allows for a more flexible assignment of monitoring functionality. At the same time, we want to avoid the forwarding delay and reduce the communication overhead that is currently caused by the switch-based solutions.

Finally, in regard to service identification, while this is a rather straightforward process for the application monitoring tools that use software instrumentation, it is more challenging when we only have access to the message flows between the application components. The next section thus discusses network-based service identification in more detail.

2.6 Network-based Service Identification

In the context of a MaaS for cloud-based distributed applications we believe that the appropriate granularity for a service label of a component is in most cases the software system that is used by the component, e.g., a MySQL, PostgreSQL or DB2 database

[8]https://sflow.org/

[9]`https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html`

system, a Memcached or Redis cache[10], or a HTTP-based web-service[11]. Each of them has their own well-defined communication protocol that defines how message are exchanged between clients and service. Thus, in the context of this thesis, we are looking for a Service Type Classifier (STC) that can identify the service type of each component based on the communication flows it has with its clients.

In fact, there is a large body of research approaches that can be used for this purpose, referred to as “Network Traffic Classification” (NTC): given a flow of messages, the task is to classify that flow. In [ZJYP21], the authors lay out different granularities for the classification result. Quite common is the service type level that we also envision, that is based on the communication protocol used. But the aim could also be to identify large scale well-known applications (such as Facebook, Google, Youtube, etc.), a more high-level definition of service (mail, database system), or the activity type such as chatting or streaming. NTC has been intensively used for various cloud management tasks such as provisioning Quality of Service (QoS), identifying faults, resource allocations, and security monitoring [PSkJ17, LKK+19, RKL20, RYCW21, GKK+19, WZZ+17].

There exists different definitions of what exactly is a network flow that is given as input to the classifier. In the most common case, it is defined as all the messages from a sender identified by a specific IP address and port to a receiver, again identified by an IP address and port number. But it could also be the bidirectional flow of messages that are exchanged between these two endpoints.

Given that NTC seems a promising approach that we can use in the context of our MaaS, we have a closer look in this section at the different approaches proposed for NTC in the research literature.

2.6.1 Rule-based NTCs

Rule-based NTC techniques rely on packet headers or perform a deep-packet inspection, and require knowledge of the communication patterns.

[10]`https://redis.io`.

[11]As mentioned in Chapter 1, the individual components in microservice-based architectures are all wrapped within their own web-service. Thus, a STC based on communication protocols would identify all of these microservice-based components as HTTP-based services, which might not be very informative. In this case, a more fine-grained level service identification about the particular purpose of the individual components, such as persistence, authentication, etc., would be valuable and provide more insights into the application call graph. Therefore, we propose a Natural Language Processing (NLP)-based approach for a fine-grained analysis of web-services in Chapter 6.

- *Header-based* approaches rely solely on the header information of the network packet, such as the source/destination IP addresses and port numbers to identify the application/service type [J. 17, SZL+12, ZZZ+, LKC+16, SCP14]. This works well for application/services with predefined IP/port numbers but not when these services are configured to use dynamic IP/port numbers. We consider this not adequate for our purposes.
- *Deep packet inspection (DPI)* techniques overcome IP/port number dependency by analyzing message formats and match them with predefined protocol-specific characteristics. For instance, HTTP request packets contain the URL string, Memcached packets contain information about whether a call is a put or a get, etc. Despite high identification rates, the usage of format rules requires maintaining an up-to-date rule database for a wide range of services. [FRR+14] provides a survey of DPI-based classification approaches.

Others use more advanced techniques. For instance, TopClass [HLZ+] matches the application call graph to predefined service architecture templates through graph similarity algorithms in order to infer the most likely running service in each application component. However, the computational overhead of graph similarity algorithms is high and proportionally increases with the number of service templates. Also, these service templates need first to be created, requiring expert knowledge.

2.6.2 Traditional Machine Learning based NTCs

Some NTCs use traditional machine learning classifiers such as logistic regression, decision trees and support vector machines. In particular, their input for learning is a set of records, each record consisting of a set of features (including a class label) describing a particular network flow. These features need to be extracted from the messages that make up the network flow, thus requiring a complex feature engineering process [ZXW+13]. The learned model can then be used to classify unlabeled network flows. Those flows need again be transformed into a feature vector before they can be fed to the model for classification.

The stochastic profile of network flows such as bytes transmitted, packet inter-arrival times, and flow duration can be used as features.

Both supervised and unsupervised methods have been proposed in the literature. For instance, Singh et al. [Sin] use unsupervised K-means to form groups of different applications based on the similarity of their network traffic. As examples of supervised learning, the approach of [WYKH15] uses random forests to classify several mobile applications, while

Amaral et al. [ADP+] additionally use stochastic and extreme gradient boosting algorithms to classify applications such as Skype and Facebook. In addition, Parsaei et al. [PSkJ17] employ different kinds of neural network estimators, while Shafiq et al. [SYW16] use multi-layer perceptron, C4.5 decision tree, and support vector machine to classify network traffic.

The disadvantage of these traditional machine learning models is clearly the tedious feature engineering process. It is clearly not trivial to find the appropriate parameters to be fed to the machine learning algorithm.

2.6.3 Deep Learning-based NTCs

Recently, deep learning has been exploited as NTC for service/application type classification purpose. The interesting aspect about deep learning models is that the features are extracted automatically, which alleviates the need for manual feature engineering needed for traditional machine learning approaches. In general, deep learning models learn both the low-level features and the high-level presentation of input data across model layers and have been shown to outperform traditional machine learning algorithms as the amount of learning data increases [LKK+19]. This makes deep learning very attractive for traffic classification, as there is no need to have detailed knowledge of the service specifics and message formats to learn a model.

Existing deep-learning based NTCs use either the packet-header or packet-payload as training data. For instance, Wang et al. [WZZ+17] have introduced a Convolutional Neural Network (CNN) model to differentiate between malware traffic and normal traffic using the packet-header information. Similarly, Martín et al. [MCSL17] applied several architectures of recurrent neural network (RNN) and CNN models to detect the service type (e.g., HTTP, SIP ...) of network flows, while using packet-header information as learning data. On the other hand, Lim et al. [LKH+] and [LKK+19], use the packet-payload for training two deep learning models, with focus on predicting different applications (Facebook, Google, Wikipedia, Yahoo..etc.) that all use the same service type (HTTP). Similarly, Rezaei et al. [RKL20] proposed a packet payload-based CNN model to classify various mobile applications such as Google Map, Google Music, YouTube, HTTP, etc.

As the performance of these NTCs is very promising, we have performed a thorough analysis and comparison of this existing work along many different configuration parameters to see whether it can serve well the STC task that we need for our MaaS. Thus, we describe these approaches in more detail in Chapter 5, where we also explore further types of deep-learning models. In Chapter 6, we show how we have adapted the deep learning model-based STC in our proposed application monitoring framework DyMonD.

3 Application Monitoring as a Network Service

The first step of our research was to determine where within the communication layer to conduct the main phases of application monitoring, namely data collection and performance metric calculation. We argue that the network switches can potentially be involved in both. In the following, we first outline how some of the switch-based approaches for monitoring the network that we have introduced in Section 2.4 can be exploited for application monitoring. In addition, we present a new solution in which monitoring functionality is loosely integrated into software-based switch components, allowing for a flexible and adjustable deployment of data collection and analysis functions. We then compare all approaches and analyze the corresponding trade-offs.

3.1 Collecting Application Monitoring Data via Port Mirroring

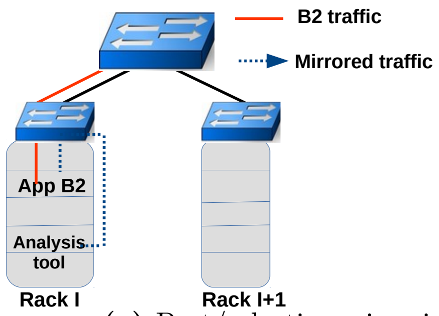
As already discussed in Section 2.4, port mirroring can be used for data collection in the network so that relevant information is forwarded to an analysis component. Thus, we have a deeper look at port mirroring techniques before we look at how to use them for the collection phase of application monitoring.

3.1.1 Port Mirroring Principles

Using standard port mirroring, the switch forwards ***all*** messages of a flow not only to the indicated destination but also to a secondary destination. Figure 3.1a shows an example of port mirroring of all network traffic of component *B2*. Port mirroring is simple, and does not impose major computational overhead at the switch because it does not perform any kind of reformatting of the mirrored messages. However, port mirroring might have a significant communication overhead as it mirrors all packages of a flow. A further disadvantage of mirroring is that the machine to which messages are forwarded has to be directly connected to the switch as mirroring does not change the routing information of the mirrored package. In the example in Figure 3.1a, the switch to perform the mirroring is the rack switch on which the analysis tool runs; thus, there is a direct connection. Furthermore, in case of software switches that connect components running on the same physical machine, mirroring is also a possibility. Port mirroring runs with lower priority compared to the normal forwarding. When a switch becomes busy, then switching the normal traffic flow takes high priority and the performance of mirroring may be degraded or in extreme circumstances be temporarily suspended.

Reducing message overhead: *Selective* mirroring uses the matching capability of commodity cloud network switches to copy and forward only those network packets to the analysis tool that match predefined criteria, thus reducing the number of packets to be transferred. Selective mirroring can be defined in the switch through the flow rules. For example, suppose that B2 in Figure 3.1a is a web-server and we only want to mirror *data messages* to the analysis tool that are relevant to calculate the performance metrics of interest. As such, we can add two OpenFlow rules to B2's ToR switch, one for each flow direction B2 $\rightarrow$ client(s) and client(s) $\rightarrow$ B2. We will show shortly how such rule would look like for a specific metric. Furthermore, messages can be truncated so that only the first X bytes are mirrored, with X being a parameter. For instance, if the packet header suffices for the analysis, we can use the truncation option to mirror only the packet header. We refer to it as *"header mirroring"*, and avoid sending possibly large and mostly irrelevant payloads to the analysis tool.

One has to note that selective mirroring is coupled with the switch forwarding path as the flow rules are executed at the time messages are processed. This may lead to a processing overhead and thus, delay in routing the message. Furthermore, flow rules only apply to the packet header; a deep inspection is not possible. Thus, in some cases one might have to send more packets than actually necessary as filtering at the header-level might be quite coarse-grained. Just as basic port mirroring, selective port mirroring requires a direct connection between switch and the mirroring destination.



(a) Port/selective mirroring

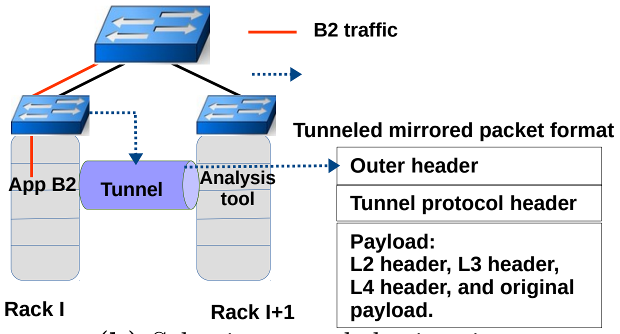

B2 traffic
Tunneled mirrored packet format
Outer header
Tunnel protocol header
Payload: L2 header, L3 header, L4 header, and original payload.
App B2
Tunnel
Analysis tool
Rack I
Rack I+1
(b) Selective tunneled mirroring.

Figure 3.1: Monitoring options in the network

Selective tunneled mirroring: If the analysis tool should reside anywhere in the network, packet *tunneling* can be used. Tunnels, in conjunction with OpenFlow, can be used to create a virtual overlay network with its own addressing scheme and topology [ZKC+15]. Figure 3.1b shows an example where the switch of B2 is programmed to tunnel messages to the analysis tool. Tunneling protocols such as GRE [FLH+00] or VXLAN [MDD+14] encapsulate network data and protocol information in other network packet payload. Tunneling protocol adds two headers to each encapsulated packet to allow the encapsulated packets to arrive at their proper destination: 1) the tunneling protocol header that indicates the protocol type used by the encapsulated packet, and 2) an outer header which contains the Ethernet and IP headers of the sending switch and tunnel destination, and the payload contains the original packet (starting from the L2 header). At the final destination, de-capsulation occurs and the original packet data is extracted. Flow rules similar to the one described earlier in this section are added to the switch in order to define the selection criteria for network packet encapsulation.

To enable tunneling, the switch will be configured to set up a tunnel between itself and the host where the analysis tool resides. Then, OpenFlow rules similar to the one described earlier in this Section 2.1.1 are added to the switch in order to define the selection criteria to encapsulate the network packets and send them through the tunnel to the analysis tool.

All mirroring approaches can be used not only by software switches but also by OpenFlow capable hardware switches as we only use standard OpenFlow mechanisms to enable mirroring.

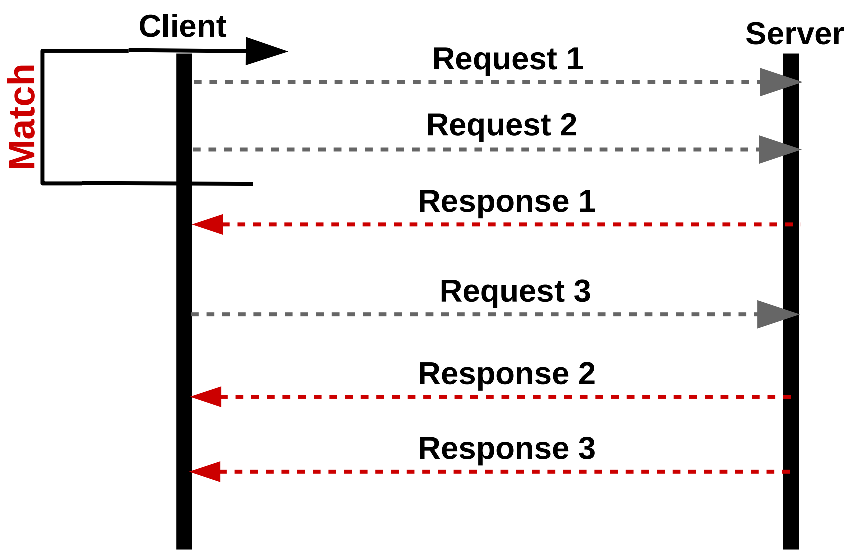


Figure 3.2: Request/response packet pair matching methodology

3.1.2 Using Mirroring Techniques for Application Monitoring

In this sub-section we discuss how these different selective mirroring options can be used for application performance logging. To better illustrate and compare the approaches, we choose request service times for HTTP requests as an example of a performance metrics to measure. Given the ubiquity of HTTP traffic in many cloud centers [MN15,DP11], providing transparently and on-demand request service times to web applications would be a crucial feature of any monitoring service [CC16].

HTTP request service time analysis The request service time can be defined as the time difference between the last request data packet received by the web-server and the first data response packet initiated by the web-server. The idea is to filter exactly those data messages at the switch that are needed to measure request service time and forward them to the analysis tool.

In many HTTP versions, a client connection to the web-server can have at most one outstanding request; that is, a client can only send a new request once it has received a response for the outstanding request. Thus, by having access to the flows from client to server and from server to client, one can take the time difference between the observed request and response as the request service time. If a client is allowed to have multiple outstanding requests (as shown in Figure 3.2), then one can simply assume that the first response refers to the first request, the second response to the second request, etc. In fact, as also the client needs to know to which request to match a response, some servers guarantee that they will send responses only in the order they received requests even if they execute the requests concurrently. In newer versions of HTTP, each request/response tuple is associated with a unique ID. When the client sends a request, it gives it an ID, and that ID will be put in the server's answer. Therefore, this ID can be used to match the responses with requests without the need to respect the requests' order.

Most request/reply protocols, including HTTP, use TCP as underlying communication mechanism. Thus, the header information of messages follows TCP format, and the HTTP (or other protocol specific) headers are within the data payload. Requests and responses could be spread across several TCP messages. While unlikely, even the protocol specific header could be spread across more than one TCP message. Thus, we have to be careful of how we detect the right messages.

To filter the HTTP data messages needed to the calculate the request service time, some form of deep packet inspection is needed. Deep inspection occurs when we look at the message payload past the TCP header. As TCP headers have a fixed size, it is quite straightforward to perform deep inspection and extract the HTTP specific headers to detect the TCP packet that contains the last part of the HTTP request header for the flow from client to server, and the packet that contains the first part of the HTTP response header for the flow from server to client. Request service time can then be determined by taking the time difference between the arrival of these two packets. We assume that the timestamps of the request and response packets are captured by the same monitoring entity. Otherwise, a clock synchronization mechanism between the entities that capture those timestamps should be in place [GS20, PPN+20, JG17].

Data collection using the different mirroring approaches The question now arises how we can use the various mirroring techniques to send the relevant data to the analysis tool so that it can perform the response time analysis as just described. Using port mirroring, all messages are sent to the analysis tool, which then has access to all messages to perform the analysis. When using selective mirroring, the idea is that we only send the relevant packets to the analysis tool. However, selective or selective tunneled mirroring cannot do deep inspection but has only access to the TCP headers. Therefore, in our selective mirroring approach, our filtering rules rely on the assumption that if a request or reply message is split into n TCP-packets ($n \geq 1$), then all data packets have the TCP "ACK" flag set and the TCP flag "PSH" is set to true in the last of these TCP packets. We have confirmed that this assumption holds for the HTTP implementation we have deployed.

Thus, given the above example of B2 being a web-server, for the particular task of HTTP request service time, we set up two flow rules. The first is to forward packets that are sent to the B2, and have the ACK and PSH TCP flags set. This is to filter the last HTTP request data packet. The second rule is to mirror the ACK packets sent by the B2 to the client(s). This is to ensure capturing the first HTTP response data packet. Note that we use wildcards "*" for the client IP to cover all the client connections.

Rule1:
Conditions: *TCP-protocol, Source IP = *, Source Port = *, Dest. IP = B2's IP, Dest. port =8080, TCP-Flags = ACK and PSH*
Actions: *Forward to B2 and Analysis Tool.*
Rule2:
Conditions: *TCP-protocol, Source IP = B2's IP, Source Port = 8080, Dest. IP = *, Dest. port =*, TCP-Flags = ACK*
Actions: *Forward to original destination and Analysis Tool.*

In this scenario, the TCP-Flags ACK and PSH are used to distinguish data packets from TCP control messages that are of no relevance for monitoring. As most ACKs are piggypacked on data packets, these mirroring rules should hopefully not send too many unnecessary messages. Note that apart of the approach that retrieves the HTTP IDs, the analysis node only needs the message headers. Thus, we can truncate the mirrored messages to contain only the headers. When HTTP IDs are used, we have to keep at least the amount of the payload that contains the HTTP header.

For both the port mirroring and selective mirroring, when the analysis node receives those packets, it first organizes them into distinguishable flows. Each flow is defined by basically 4 fields: Source IP, source port, destination IP and destination port. Then, the analysis tool constructs individual HTTP pairs according to the applied HTTP version protocol as described above and calculates the corresponding service times. In case of mirroring without selection, the analysis tool has to handle and inspect a significantly larger number of messages than when selective mirroring is used.

Avoiding deep packet inspection An estimate of the web-server response time can be determined, without the need for deep packet inspection, by continuously observing the intervals between ACKs sent from the web-server side for each client. Such approach has been used by [LW15]. Following this approach, one can only mirror the header information of the web-server ACK packets to the analysis tool, i.e. header mirroring. Therefore, *Rule2* in the previous example is only needed in this case while setting the length of the mirrored packet to the length of the packet headers.

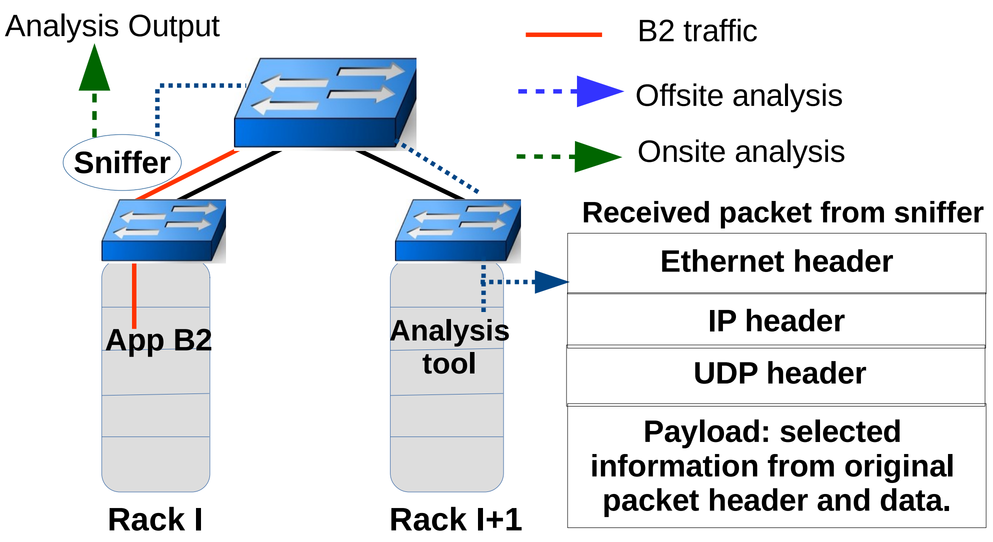


Figure 3.3: The overview of the proposed sniffer.

3.2 Sniffer

While mirroring and selective mirroring are only capable of forwarding (hopefully efficiently) relevant messages to an analysis tool that then does the actual analysis, we propose in this section a new approach that provides more flexibility and allows the switch to perform some analysis locally. Our proposal requires the switch to be provided as a software switch. We refer to this approach as *sniffer*. The idea is that the sniffer "attaches" to the ingoing and outgoing ports of the software switch and inspects the ingoing and outgoing messages on these ports. This is somewhat conceptually similar to how Wireshark/TShark sniffs the messages at the network cards of the end-hosts, and we have actually closely looked at the various mechanisms that have been developed to do such sniffing efficiently. Figure 3.3 shows the principle design. The sniffer is implemented as an independent process on the node running the software switch. For instance, assuming that all ToRs in Figure 2.3 are virtualized on computing nodes, a sniffer process can be deployed on B2's ToR switch host and instructed to sniff all web traffic traversing the virtual switch port connected to B2. In principle, the sniffer can implement any kind of semantics; for example, the sniffer can simply forward selective messages to an analysis tool, aggregate and reformat logging messages that only contain information relevant for the analysis (as depicted in Figure 3.3), or just perform the analysis by itself. For the latter case, tools such as TShark could be deployed on the switch node. In our implementation, we follow a flexible approach that allows a wide range of possibilities.

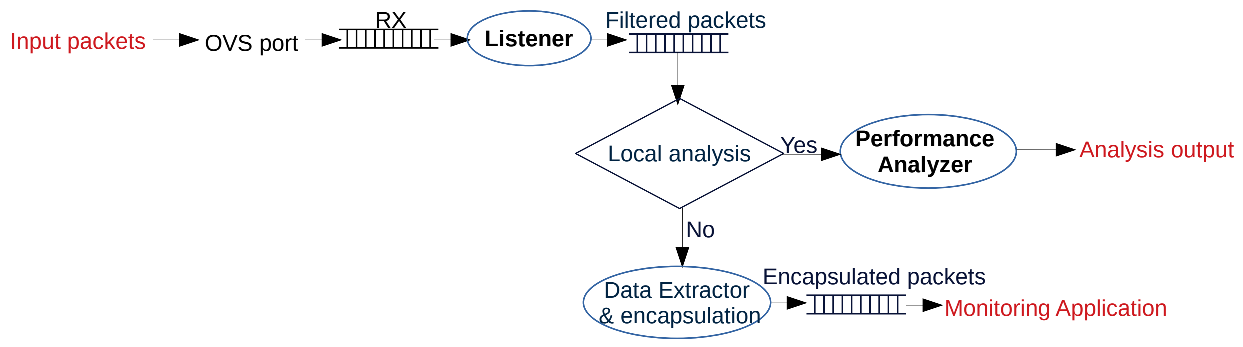


Figure 3.4: Sniffer architecture.

While the proposed sniffer can simply forward selective messages to the analysis tool conceptually similar to the mirroring approaches discussed in Section 3.1, it has the advantage of not having the monitoring functionality coupled with the forwarding path of the switch as it is an independent process at the host of the software switch. Another advantage of the proposed sniffer approach over the mirroring approaches is that it can perform a fine-grained filtering of the network messages based on non-header fields. However, in contrast to the mirroring approaches, our proposed sniffer can only be used in software switches.

In Section 2.4 we discussed several switch enhancements proposals [WGH+15, ZWG+18, CLKdR16, MHM+14]. Our sniffer also enhances the switch functionality. But again, it is *completely decoupled* from the switch forwarding path and enhances the switch functionality with *zero* modification to the switch source code. Thus, the deployment of our proposed sniffer is applicable in a wide range of settings.

3.2.1 Design

Here we describe the internal design of our proposed sniffer within OVS as an example of the software switch. Figure 3.4 shows how packets flow through the sniffer. The sniffer separates the actual sniffing from any additional tasks. A *listener* thread keeps inspecting the received packets' buffer of the predefined switch port, filters relevant messages, and saves the needed traffic packets into a shared memory space, i.e. the "filtered packets" queue in Figure 3.4. From there, further extraction, analysis and forwarding is performed by extra thread(s) as needed. We have implemented the listener in a separate thread as it has to work at the speed of the OVS ports. Thus, we wanted to make the listener task as simple as possible, allow for straightforward multi-threading and avoid interference with analysis functions.

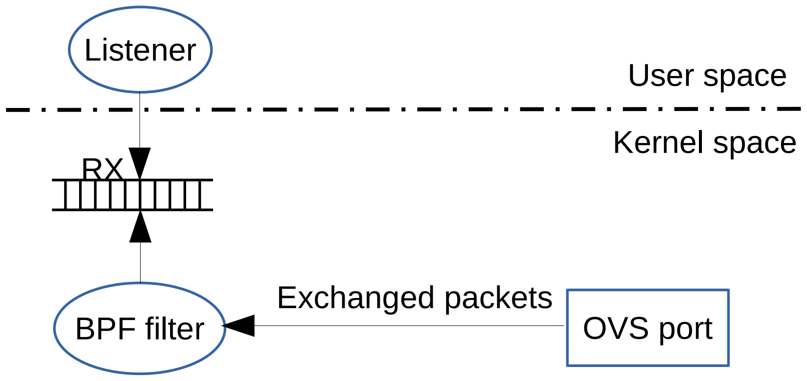


Figure 3.5: Packet capturing overview

The listener thread sniffs the packets by inserting some filtering code (such as BPF [Lin19]) into the kernel of the software's switch host at run-time. BPF supports filtering packets, allowing a userspace process to supply a filter program that specifies which packets it wants to receive. For example, for capturing HTTP service request times, the listener thread injects BPF filter code to receive only packets that initiate a TCP connection to port number 8080 over the OVS port that is connected to the web-server. In this case, BPF returns only packets that pass the filter that the listener thread supplies. This avoids copying unwanted packets from the operating system kernel to the listener thread, greatly improving performance. This code will copy each exchanged packet at the monitored OVS port that passes the filter code and place it into a buffer (RX buffer) where the userspace listener thread will read the buffer and get the packets. Figure 3.5 illustrates this process. Note that tcpdump and TShark are imposing similar kernel filtering code as the one we used in the sniffer to collect the network packets. Thus, there is packet copy overhead for these commercial tools as well.

Taking the same example of capturing HTTP service request times, we have implemented two options for processing the collected data by the listener thread. In the *online* option, a *performance analyzer* thread extracts the relevant information from the messages deposited by the listener in the "filtered packets" queue. Such a thread distinguishes the different client connections and captures the arrival time of messages as described in Section 3.1. In order to detect requests and responses, it performs a deep inspection of TCP packets, as the http headers are embedded in the payload of these packets. It matches request/response for each client connection and calculates service times. Only simple data structures are maintained.

In the *offline* option, a *Data Extractor & Encapsulation* thread extracts the relevant information, such as source and destination IP/port pairs, and determines timestamps of data packets, but does not do the matching itself. Instead, the extracted data is encapsulated and placed in the "encapsulated packets" queue to be forwarded to a remote analysis tool (see Figure 3.4), similar to the selective forwarding mechanisms described in Section 3.1.1. However, our sniffer can perform fine packet filtering based on non-header fields such as the TCP payload size and data. For instance, to measure the HTTP request service time, the sniffer forwards the extracted information to the analysis tool only for HTTP data packets (i.e., with TCP payload size > 0). The analysis tool, in turn, will perform request/response packet pairs matching using the received data and calculate the corresponding HTTP request service time. The sniffer sends the extracted information using UDP.

In the performance analyzer component, whether it resides within our sniffer or is on a separate analysis node, we have to distinguish the messages from the different connections of interest. Thus, when information about a request message arrives, it is stored in a connection specific data structure until the response arrives. Only the time difference between a request and its response needs to be kept track of. If several requests are queued when a response arrives, the match is done as described in Section 3.1.2.

Note that the precision of the measurements might depend on where timestamps are taken. For the mirroring approaches, the times are taken when the mirrored messages arrive at the analysis tool. For port sniffing, the sniffer process can take the time. In both cases, this is not the time when the message was sent by the original source nor the time the destination receives the message. For example, the time taken by the sniffer for the request is before the message arrives at the server, and for the response it is after the message is sent by the server. Our assumption is that message delay times in the network are negligible compared to request execution times, especially if the switch in charge of mirroring or sniffing and the analysis tool are close to the server under observation.

3.3 Evaluation

In this section we present an evaluation of the approaches presented in Sections 3.1 and 3.2, and compare their performance also with a software instrumentation-based tool as well as network tools that can run on the application end host such as TShark and tcpdump. Table 3.1 enlists these mechanisms and their characteristics such as where information capturing takes place, and whether the data collection and analysis are done at the same location (i.e. onsite analysis). For a fair performance comparison of all the evaluated monitoring mechanisms, we have implemented and configured all of them to log HTTP request service time into a file. The location of the analysis file depends on the location of the analysis process, i.e. at a separate analysis node, the same host where the application is running, or at the host of the switch. For all switch-based approaches, the analysis process follows the

Monitoring mechanism	Data collection location	Onsite analysis	Location
Tomcat logging	Application platform	Yes	Host of application.
TShark	Application host	Yes	Host of application.
Tcpdump	Application host	No	Host of application.
Port mirroring	OVS host	No	Analysis node.
Selective mirroring	OVS host	No	Analysis node.
Header mirroring	OVS host	No	Analysis node.
Tunneled mirroring	OVS host	No	Analysis node.
Sniffer	OVS host	Yes/No	Onsite: Host of the switch. Offsite: Analysis node.

Table 3.1: A list of evaluated approaches along with their characteristics.

approach depicted in Figure 3.2 for matching the HTTP responses with their requests in order to calculate the HTTP request service time.

Compared Approaches For a platform-specific *software instrumentation* at the end host, we enabled the access log valve in Apache Tomcat server to log HTTP request service times. We refer to this as *Tomcat* in the performance graphs.

For *networking monitoring tools deployed at the web-server host*, we use *TShark*, the command line interface to Wireshark, and *tcpdump*. With TShark, we can do the analysis in an online fashion, i.e., TShark sniffs the messages, analyzes requests and either visualizes them or logs them to a file. Visualization was considerably more expensive. Thus, our evaluations show the overhead when results are dumped to a file. Tcpdump is only a message capturing tool with filtering capability. Such capability is only used to filter messages that are relevant for the analysis. Tcpdump does not have an analysis engine, and is thus only instructed to dump all packets to/from the web-server port to a disk file, that can be then fed into any offline analysis tool.

For application monitoring in the network, we have evaluated port mirroring, selective port mirroring of whole matched packet as well as for the header only (i.e. header mirroring), tunneled mirroring using GRE/VXLAN and our sniffer. As both tunneling approaches have very similar performance results, we only show VXLAN in the graphs for better readability. For all approaches except of the sniffer, the mirrored packets are sent to the analysis tool. For the sniffer approach, we show the results when the sniffer performs the analysis itself and when it sends relevant data to the analysis tool (similar to what the selective mirroring approaches do). The remote analysis tool used for mirroring and by our offsite sniffer performs the analysis and dumps the results to a file.

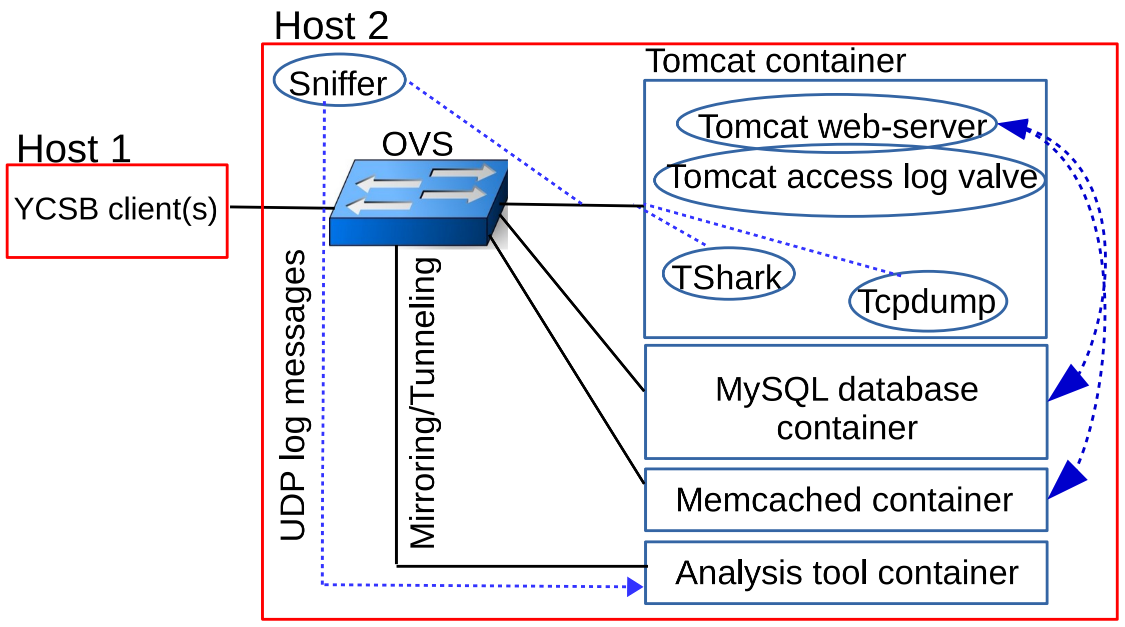


Figure 3.6: Test application architecture.

Test environment Our basis for evaluation has been the YCSB benchmark [CST+] on an extended architecture as depicted in Figure 3.6. While YCSB is originally a database benchmark where the YCSB client sends requests to a database, our extended version has added a Tomcat web-server as frontend for the client and a caching server. The clients submit a predefined workload of HTTP requests to the web-server whereby each request retrieves data from either a MySQL database or a Memcached server. The experiments use the YCSB read-only workload with 3 million scan requests and zipfian distribution for record selection over a 10GB database (10 million records). Each test scenario runs such workload for two minutes and we report the average of 5 workload runs for each test scenario. We also show the error bars. We have tested with up to 30 client threads, where the web-server gets saturated even without monitoring enabled.

In order to compare the performance of different monitoring approaches, we run our YCSB benchmark with and without monitoring. We then measure the overhead for each of the monitoring approaches by analyzing the client perceived performance. That is, we check how the different approaches affect the latency observed at the clients. We also compare the performance of our sniffer to TShark and tcpdump in terms of resource utilization and to the mirroring approaches in terms of communication overhead. Finally, we show how OVS forwarding performance is affected in case of the mirroring approaches.

The experiments are performed using two DELL hosts with dual Intel(R) Xeon(R) CPU

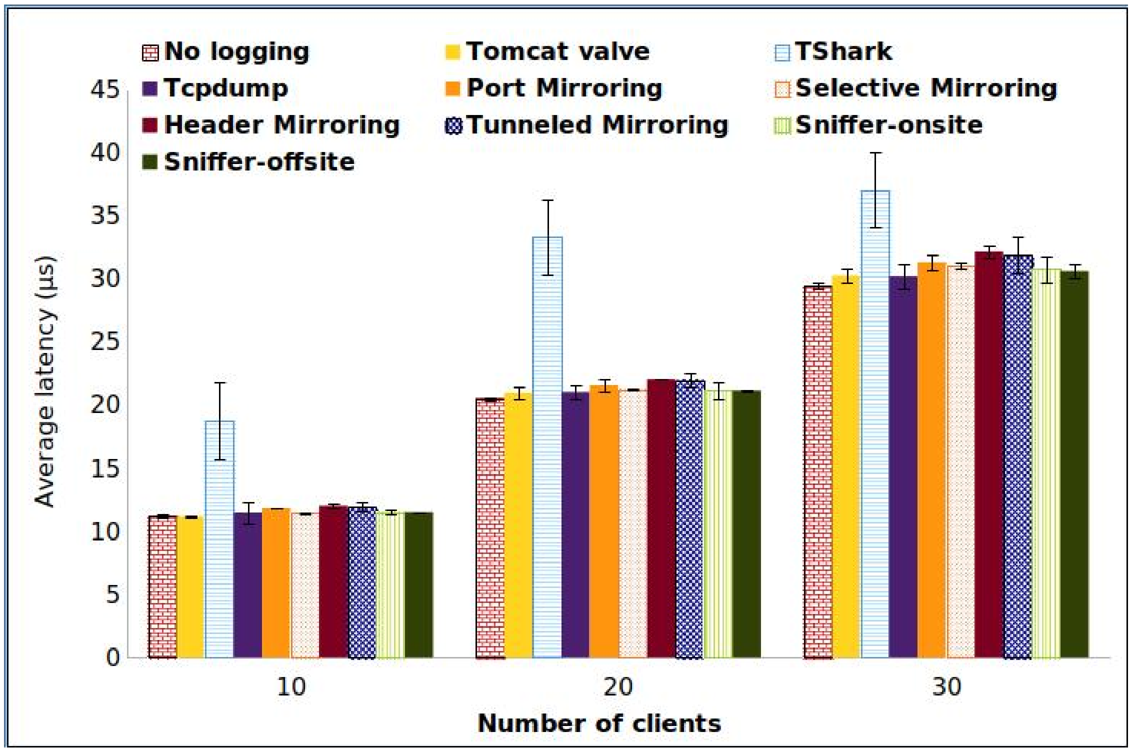


Figure 3.7: Average latency reported by YCSB client under various monitoring approaches.

E3-1220 v5 @ 3.00GHz CPUs (4 cores per socket), a Broadcom NetXtreme BCM5720 Gigabit Ethernet Dual Port NIC, and 32.8GB memory, with the clients on one machine and all server components on another machine together with the OVS software switch. This resembles the scenario where the cloud provider has large end-host machines that host many components. Each server component runs in its own docker container (docker-ce version 18.03.1) with predefined available resources and a separate core. All docker containers are connected by 10 Gigabit Ethernet OVS ports. We used OVS version 2.9.90. 16GB of RAM are assigned to Memcached 1.5.12, the frontend web-server employs Apache Tomcat 9.0.13 and backend database system is MySQL 5.7.24. A separate analysis tool component is used for some of the evaluated approaches as shown in Table 3.1.

3.3.1 Application Latency

We have examined the impact of monitoring on the latency at the YCSB client. Ideally, monitoring has little to no impact on the performance observed at the client side. Figure 3.7 shows the end-to-end latency observed by the YCSB client while increasing the workload, i.e., adding more client threads. Of all approaches, Tomcat valve works best by having nearly no impact on performance. This is because the access log valve does not need to perform any sophisticated message analysis. This is only possible because the valve is tightly integrated into Tomcat's software and can very easily intercept the events of receiving a request and sending a response. The other two end-host mechanisms, both application independent, negatively affect performance. Tcpdump has much lower impact than TShark, though. For example, with 20 clients TShark has 67% more latency compared to no monitoring, while tcpdump increased the latency by around 1%. Note that tcpdump stores the logged messages locally and the file needs to be retrieved from there before analysis takes place. That is no analysis or message sending occurs. In contrast, mirroring-based approaches and the two variants of our sniffer perform significantly better than TShark (by at least 15%), and only slightly worse than tcpdump (up to 6.5% for the header mirroring approach at the highest workload). This is very promising given that they do so much more than tcpdump (forwarding and reformating the messages, or even performing the analysis). The mirroring approaches perform quite similar with selective mirroring always being better than tunneled and header mirroring, as tunneling has the extra overhead of encapsulation and header mirroring needs a truncation processing of the messages. Mirroring all messages is more expensive than selective mirroring because more messages are forwarded but also better than tunneling and truncation because of its simplicity. The two variants of the sniffer approach are overall slightly better than the mirroring approaches and basically perform the same as tcpdump with less than 2% worse performance.

In general, for all switch-based monitoring approaches, there is a delay induced for executing the forwarding rules and copying messages, but in different contexts. As we mentioned before, with mirroring, the switch has to push the packets to an additional port. Additionally, VXLAN has the additional work of encapsulation and header mirroring has the additional work for truncating the mirrored packet. In our sniffer implementation, the kernel makes the packet copy for the listener thread as described in Section 3.2. Thus, although with our sniffer the copying process is not in the OVS forwarding path to the original destination, it runs on the same core as the OVS switch. Additionally, the sniffer executes further analysis and/or encapsulation and forwarding actions. Thus, the sniffer has an impact on the performance, albeit very minimal. In the future, we will consider implementing the listener thread with a Zero-copy fast packet processing library such as DPDK [CAR] that provides an access to the packets in the kernel space with no copying overhead.

Overall and compared to platform-dependent approaches, the results for the switch-based solutions are very promising and indicate that moving the sniffing tasks to the network is effective with no significant client perceived performance overhead.

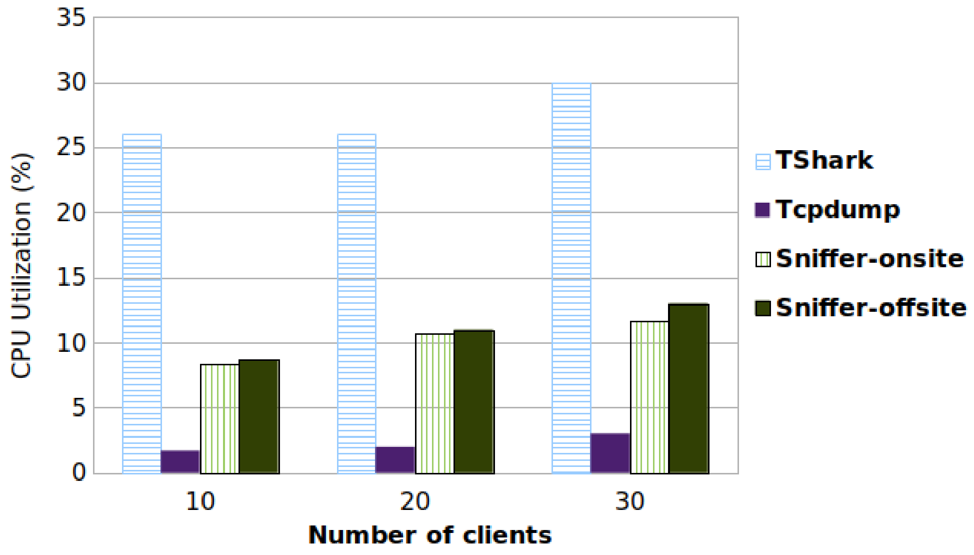


Figure 3.8: Analysis tool CPU utilization.

3.3.2 Computational Overhead

In this section we evaluate the computational overhead for the monitoring activities where it is possible. As Tomcat valve and the mirroring-based approaches are tightly integrated with the web-server and the switch respectively, it was not possible to measure the overhead. But TShark, tcpdump and both versions of our sniffer use an independent monitoring process, so we can measure the computational overhead. Note that we ignore the computational overhead of the analysis node as this is a remote activity with less impact on the monitored application.

Note that the sniffer is executed on the host of the switch while TShark and tcpdump are executed on the application host. The Linux top command is used to measure the CPU utilization of each running process. The results shown in Figure 3.8 indicate that TShark has the highest CPU utilization consuming about 27% of the CPU resources of the web-server host on average. In fact, we believe that this is the reason for the poor client-perceived performance that we discussed in the previous section. One reason for TShark's bad performance is its poor software architecture: it is single-threaded and has poor memory usage as it keeps messages in main memory for a long time. In contrast, our sniffer employs multi-threading to avoid interference between different tasks and has carefully designed data structures for message management. We note that TShark has frequently crashed during the experiments. In addition, it also missed messages at higher rates. Meanwhile, our onsite sniffer, that performs the analysis locally and writes it to a local file, performs much more efficiently than TShark by consuming only 10% of the host CPU resources. We note that in our evaluation tests, we enabled only the TShark analysis features that correspond to the ones performed by our sniffer and thus, the performance advantage of our approach is not attributed to analysis complexity. Nonetheless, the sniffer requires more CPU time than tcpdump, which is very much expected, because tcpdump actually does not do any sophisticated analysis tasks but just dumps the data into a file. Tcpdump consumed about 3% on average of the CPU resources of the web-server. Interestingly, when the sniffer only reformats messages and sends them to a remote analysis tool, it requires on average around 1% more CPU utilization compared to the onsite sniffer version. It seems like creating messages and sending them to a remote site is more CPU intensive than performing the analysis locally, at least for the relative simple analysis of measuring request response times.

3.3.3 Switch Overhead

Here we compare the performance of port mirroring, selective mirroring, tunneled mirroring and header mirroring in terms of their impact on the OVS forwarding performance. To do that, we use Iperf [ESn16] to measure core link performance. We deploy the Iperf server, the Iperf client and an analysis tool process, each in a separate docker container, all connected through 10 Gigabit Ethernet OVS ports. We work within a single host as the focus is on the performance of the OVS software. We run experiments with up to 30 concurrent client connections.

Figure 3.9 shows forwarding latency when 30 clients are connected with the server using OVS without any mirroring or tunneling, using OVS with port, selective and header mirroring, and using OVS with tunneling (VXLAN). As expected, OVS without enabling any mirroring mechanisms performs best. Port mirroring, selective and tunneled mirroring are slightly worse than OVS. Header mirroring is the slowest.

Selective mirroring and port mirroring add around 0.2 and 0.6 microseconds latency overhead, respectively. Tunneled mirroring incurs a higher delay of 1 microsecond because it has to reformat messages (i.e. encapsulation). We believe that the client-perceived performance impact observed in Figure 3.7 for mirroring and tunneling might be partially due to this forwarding delay. However, the switch processing needed for truncating messages in header mirroring has a serious impact on the switch latency with an increase of 9.7 microseconds. In fact, not shown in a figure, the OVS CPU utilization is also 50 times higher for header mirroring compared to the other mirroring approaches. The performance impact on the client-perceived performance observed for the header mirroring in Figure 3.7 is around 9% at the maximum tested workload.

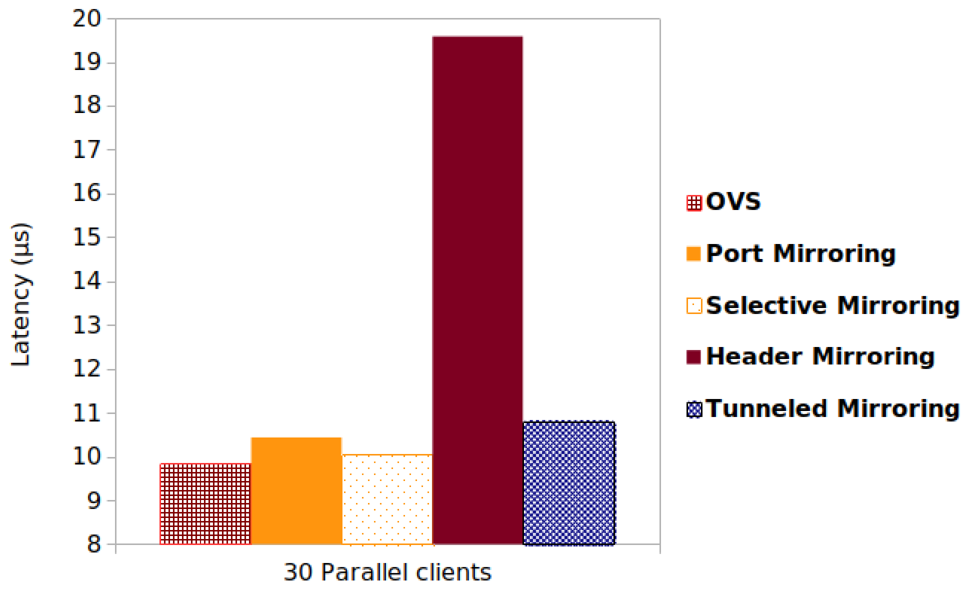
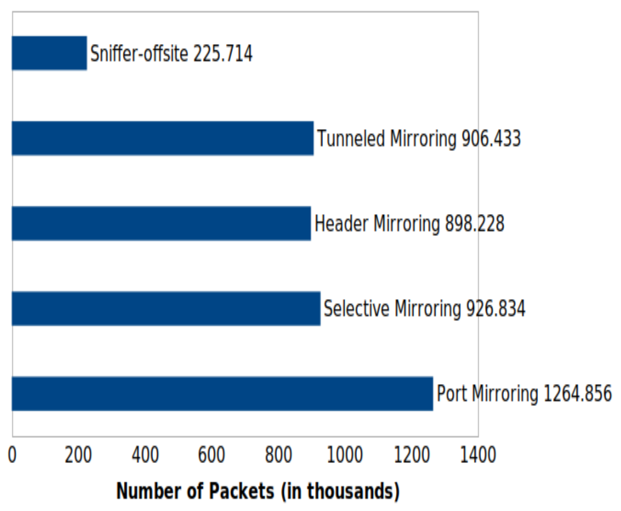


Figure 3.9: OVS link latency impact of different mirroring-based approaches

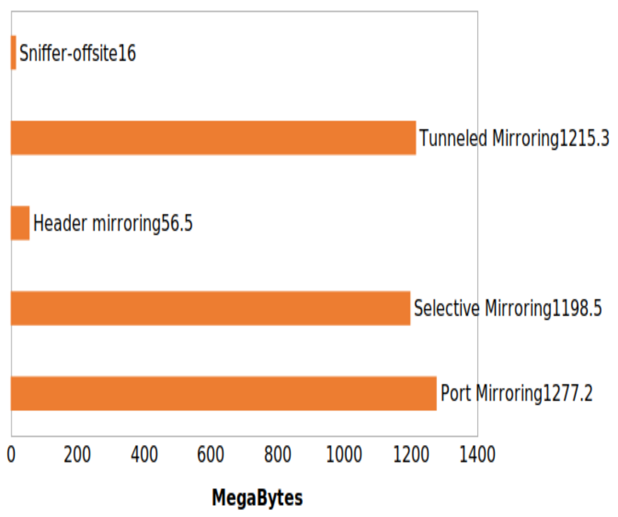
3.3.4 Communication Overhead

To analyze the communication overhead induced by sending messages to a remote analysis tool, we have collected the number of received packets and bytes at the analysis node while running YCSB with 20 clients. Figure 3.10 shows the number of transmitted packets and bytes. Figure 3.10a shows that port mirroring sends the most packets to the analysis tool as it forwards all packets over the monitored link. All the selective mirroring variants involve similar packet count since they use the same filtering rules. Overall, about 30% less messages than with port mirroring are sent. The offsite analysis sniffer transmits the fewest packets as it further eliminates sending the acknowledgement packets that do not have any data (i.e. with TCP data size = 0), which constitutes less than 12% of the number of messages port mirroring is sending.

Figure 3.10b reports the communication overhead in terms of total message sizes. VXLAN and selective forwarding send only 5% respectively 6% fewer bytes than port mirroring although they transmit 30% fewer packets. The reason is that these approaches avoid sending the control messages but these are typically small in size. Furthermore, VXLAN adds some bytes to each packet for encapsulation (i.e. the outer and VXLAN headers in Figure 3.1b) which leads to more bytes than selective mirroring. On the other hand, header mirroring sends around 95% fewer bytes compared to mirroring the whole packet, because the TCP header is only about 5% of the maximum segment size of TCP



(a) Number of packets received by analysis tool.



(b) Number of MBytes received by analysis tool.

Figure 3.10: Communication overhead when the analysis is conducted by a remote tool

packets. The offsite analysis sniffer sends the fewest bytes, representing only 1.25% of what port mirroring typically sends. The reason is that it is customized and extracts only the target fields to be sent to the analysis node.

3.4 Summary

Conducting complex application-independent analysis on the end host, as done by TShark, can have a considerable negative impact on application performance. Network-based application monitoring approaches decouple the data collection and analysis from the end components, allow for flexible placement of data collection and even analysis somewhere in the network.

Port mirroring has shown better performance than tunneling in our experiments as it introduces less overhead in the switch. However, port mirroring produces more traffic, which may negatively affect the overall cloud performance should the analysis tool reside on a different node than the switch. Selective and tunneled mirroring reduces slightly this communication overhead. Header mirroring has the least communication overhead between mirroring approaches. However, it has significant impact of the switch latency.

Compared to mirroring and tunneling, the sniffer has the advantages that: (a) it does not introduce any direct delay at the switch, and (b) it can perform some analysis locally or send selective information for remote analysis which significantly reduces the communication overhead. The disadvantage of the sniffer is that it can be only implemented in software and not on SDN-enabled hardware switches.

Overall, we believe that a network approach provides us considerably more flexibility in the placement of monitoring functionality. For hardware switches, selective mirroring is probably the most efficient approach if the analysis tool resides on a node that has a direct link to this switch, but tunneling provides flexibility for the location of the analysis tool that is probably worth the overhead. For software switches, we believe that our sniffer is the preferred route to go because of performance and flexibility.

4 Monitoring as a Service (MaaS)

In this chapter, we propose an architecture for a Monitoring-as-a-Service (MaaS) platform that can provide performance measurements on a per-component basis, and present a prototype implementation. The MaaS has as its core building block the sniffer that we have introduced in the previous chapter. Care has been taken to follow a design that allows for scalability of the different components. In its current format it already supports the monitoring of various service types providing a wide range of performance metrics. A focus of the design has been extensibility so that new service types and new performance metrics can be integrated with acceptable overhead. We also analyze the performance impact the MaaS has on the application under observation.

4.1 Overall Architecture

Figure 4.1 illustrates the overall MaaS architecture and its deployment inside the cloud network. It follows a distributed architecture that is composed of three main software components: *monitoring agent*, *visualization frontend*, and *controller*. Such distributed architecture provides considerable flexibility in the placement of the components within the cloud infrastructure. It also allows for scaling-out the different components individually. A

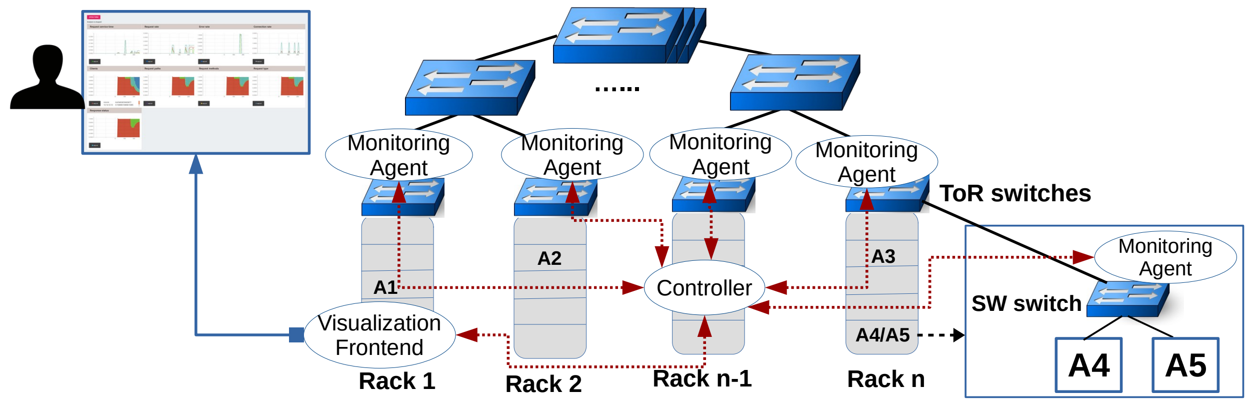


Figure 4.1: MaaS prototype architecture.

monitoring agent is attached to each software switch in the network and performs message analysis as described in the previous chapter. Users submit their monitoring requests via the visualization frontend which relays them to the controller. The controller knows about all agents in the network and sends requests to the appropriate agents, which return their results to the controller who forwards them to the visualization frontend for visualization.

Assumptions For simplicity, we assume that each component has its own unique IP address across the cloud network that allows it to communicate directly with other components. Thus, we use this IP address as an identifier of a component. For instance, in Figure 4.1, each application component of A1-A5 should have a unique IP across the cloud network. Furthermore, as MaaS leverages software switches as it loosely attaches monitoring agents to these switches in order to intercept message exchange among components, we assume that MaaS knows about all software switches deployed in the cloud network. In addition, we assume that each connection between two components passes through at least one software switch. We believe that such an assumption holds for many clouds or will do so in the near future as software switches become the prevalent solution for the lower levels of the cloud infrastructure. Should there still be flows that only go through commodity hardware switches, existing SDN technology can still be exploited in order to route relevant flows to MaaS [SDGK21, LTRW]. This can be done by dynamically adding forwarding rules to the flow tables of the hardware switches as described in Section 3.1.

4.1 Overall Architecture

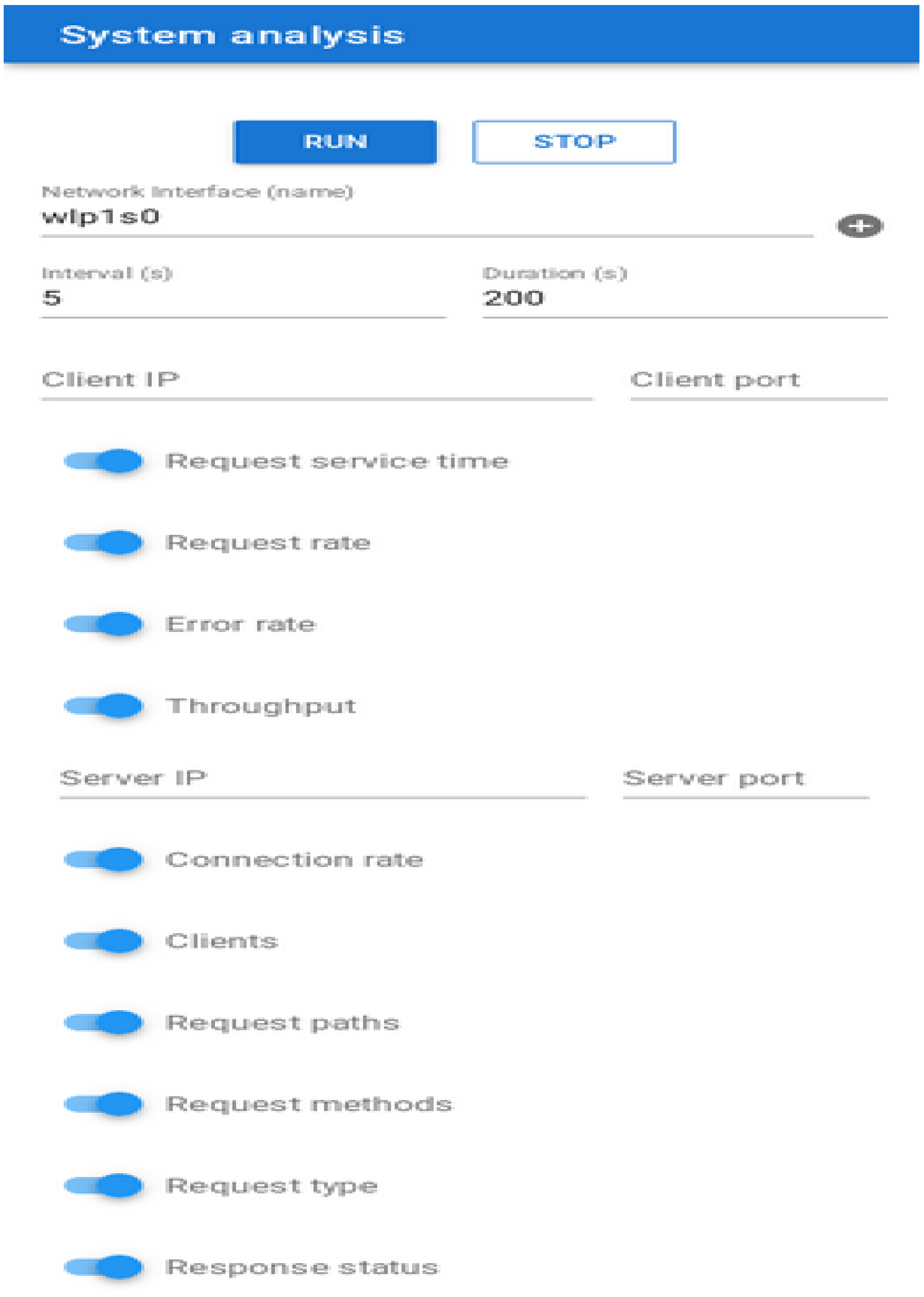
In the following we provide some details of our implementation of this architecture[1].

Visualization Frontend The visualization frontend is implemented as a web-server. Users connect via a web-browser to the frontend where they can indicate what they want to monitor. If there are many users, the frontend can be easily scaled to several instances and the clients be distributed across these instances. Frontends can be deployed on any node in the network. The user can indicate one or more services they want to monitor and what performance metrics they want to collect through a form, as depicted in Figure 4.2. They also indicate how long they want to monitor the service(s) and at which time intervals they want to receive the updates to the performance metrics. They can specify that they want to have the metrics for all clients connected to the monitored service or only for a specific client.

Once performance metrics are available for visualization, the frontend displays them in near real-time as depicted in Figure 4.3. As the frontend receives new updates to the metrics at each interval, it updates the graphs that are presented to the user. The graphs show aggregated results for each performance metric over the last time interval by reporting various statistics such as the average and cumulative distribution, as we will describe later in this chapter. This dashboard can be dynamically configured by adding and removing performance metric graphs. We will provide more information about the graphs that are visualized in the next section.

Controller The controller is the broker between frontend and monitoring agents and can reside at any node in the network. It assigns monitoring requests to relevant monitoring agents and returns the results to the frontend. Thus, the controller is itself a backend server to the visulatization frontend as well as a client to the monitoring agent server. We use a controller between frontends and monitoring agents instead of letting frontends directly communicate with the agents, because the controller has a wide range of responsibilities that are not necessary compatible with the web-server design of the frontend. First, the controller might itself perform some analysis tasks to reduce the computation done by the agents, as we discussed in the previous chapter regarding distribution of analysis between sniffer and analysis tool. Further, the controller must know and be able to communicate with all deployed agents. It also has to determine the switch/agent that connects to a service for which monitoring is requested. The details of this process will be discussed in Chapter 6.

[1]Note that a considerable part of this implementation has been realized through undergraduate research projects. Therefore, we do not present the implementation details but only give a high-level overview of the components to better understand the capabilities of the MaaS.



System analysis

RUN STOP

Network Interface (name)
wlp1s0

Interval (s)
5

Duration (s)
200

Client IP

Client port

Request service time

Request rate

Error rate

Throughput

Server IP

Server port

Connection rate

Clients

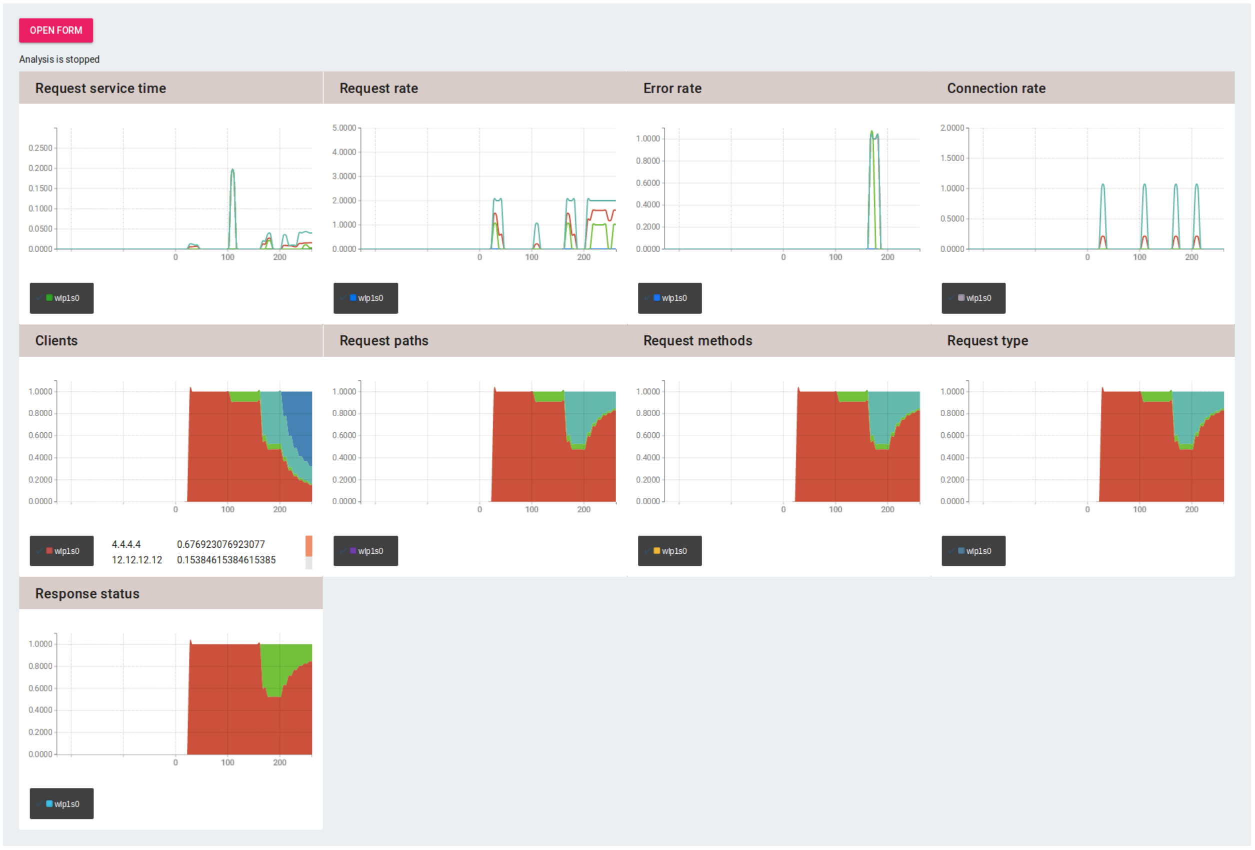
Request paths

Request methods

Request type

Response status

Figure 4.2: MaaS User interface to request monitoring [Fes].



4.4.4.4	0.676923076923077
12.12.12.12	0.15384615384615385

Figure 4.3: MaaS dashboard [Fes].

Monitoring agent Finally, the monitoring agent is an extension of the sniffer that was presented in the previous chapter. It is deployed at the host of each software switch. Figure 4.1 shows a monitoring agent at each ToR switch and the software switch that resides on the machine that hosts A4 and A5. It analyzes the messages that travel through the switch and performs performance analysis on the fly by analyzing specific network messages. Assuming the distributed application A1-A5 in Figure 4.1, the traffic between A3 and A4 could be monitored by the agent of the ToR switch of Rack n or by the agent of the switch embedded in the machine hosting A4. The monitoring agent computes the requested performance metrics at given time intervals for the duration of the monitoring and sends them to the controller for dissemination. The requested performance metrics are computed and stored at user-defined time intervals and the results are sent for visualization to the MaaS user on a regular basis. The same monitoring agent can execute several monitoring requests simultaneously. The monitoring agent is implemented as a server that continuously listens for incoming analysis requests. Then, it spawns a set of threads to efficiently distribute and separate the work required to capture the network packets from the other data extraction and analysis tasks as described before in Section 3.2.

4.2 Performance Metrics

Given a service, the message exchange between client and server typically follows a specific communication protocol most commonly built over TCP/IP. Many services use HTTP [MN15, DP11, CC16] or have their own proprietary protocol. In each of these protocols, the message payload often follows clear patterns. As pointed out in Section 2.5, considerable application relevant metrics can be extracted by looking at message content providing information about the performance of the individual components and the system overall. In this section we want to motivate the possibilities by outlying some of the metrics we can collect through message inspection for many services, and how we have implemented them in our MaaS prototype.

Many cloud application components are built from well-known services (e.g., web-server, database systems such as PostgreSQL, caching services such as Memcached or Redis, etc.) that all use a request/reply communication pattern with their clients. Providing on-demand performance metrics for such request/reply-based services will be a crucial feature of any monitoring service. Therefore, we have chosen to start with those services as a proof of concept for our MaaS.

So far, our MaaS prototype is able to calculate the performance metrics as described in Table 4.1. As we can see, most of them are common for any service that is based on request/reply patterns, which allows us to develop a generic monitoring platform to derive them. More precisely, we distinguish between two main categories: (1) *general performance metrics* for any request/reply based service, and (2) *protocol-specific performance metrics* that can be quite specific for a particular service.

General performance metrics Application performance metrics such as the request service time, number of bytes transmitted per time unit between the service and its client(s), and the service load in terms of the number of received requests per time unit, the error rates for these requests, the number of connected clients and the number of distinct open connections by those clients to the server, all are examples of application performance metrics that can be applied to any service that is based on the request/reply communication protocol. Some of these general performance metrics can be measured without deep packet inspection, as the needed information is in the packet header such as the throughput which is based

Metric name	Description
General performance metrics	
Service time	The time elapsed between the submission of a service request and its corresponding response.
Throughput	The number of bytes transmitted between a service and its client(s) in both directions (per time unit)
Connection rate	The number of distinct open connections to a service (per time unit)
Request rate	The number of service requests (per time unit)
Clients	The number of distinct hosts (IPs) connecting to a service (per time unit).
Error rate	The number of failed requests per time unit.
Protocol-specific performance metrics	
Request type	The ratio of each of the request types offered by a specific service. Examples include GET, POST, etc. for the HTTP communication protocol, SQL query type for DB systems such as SELECT, DELETE, INSERT, etc, and GET and SET for caching services.
Response status	The ratio of each of the service's response status.
Request path	The path to requested resource, as mentioned in service's request URL. It can be found, for instance, in services that use HTTP as the communication protocol. For example, the path of the requested resource in the HTTP request with HTTP URL of "https://www.overleaf.com/login" is "//www.overleaf.com/".
Request method	The called function in the service's request, such as login, register, authorize, etc. It can be found, for instance, in services that use HTTP as the communication protocol. For example, the called function in the HTTP request with HTTP URL of "https://www.overleaf.com/login" is "login".

Table 4.1: Performance metrics for request/reply based services.

on the length of the packets transmitted by the server as well as the connection and client rates that depend on the IP/port information in the packet header. In contrast, service time, request rate and request error rate may require a deep packet inspection to identify the service individual requests, responses, and the response status code. An example of how this can be done for HTTP request service time is described in Section 3.1.2.

Protocol-specific performance metrics Each service protocol could have its unique parameters that are needed to be monitored. For instance, each request/reply-based service usually has different types of the possible requests and responses. For instance, the clients of HTTP-based services can send a GET request to retrieve data from a specific resource, a POST request to create/update a specific resource, or a DELETE request to delete a certain resource. Similarly, GET, SET and DELETE requests could be initiated to key-value services to read the value, add a new key-value record, and delete key-value record, respectively. The response messages also may have different codes which indicate different statuses. For instance, HTTP response messages have different codes to indicate the different types of errors for the failed requests such as receiving a not understandable request or unavailability of the web-server to handle the request. Other examples of protocol-specific performance metrics include the frequency of requests for individual objects/methods. For instance, HTTP request messages contain the full path, in terms of a URL, of the requested resource and the called function such as *login, register, authorize, etc.*. Keeping track of the number of requests per URL and/or method, calculating their frequencies on an interval basis can be used to measure the popularity of the resources and the load for the different methods. In a Memcached service, one might be interested to keep statistics about access frequencies of certain objects. Again, if the structure of the request messages is known to the monitoring service, this information is easily obtainable.

Thus, there is a similarity regarding the semantics but the implementation details can be different for each service and likely need deep packet inspection. For example, the response status performance metric will have similar semantics for each request/reply based service, but the response status codes will not be identical. In summary, the MaaS must know how to parse the service's messages according to its communication protocol, to be able to extract the needed information for the performance metrics shown in Table 4.1 in the service context. That is, while we do not need to instrument the services that we monitor, we still need to be aware of the service-specific communication protocol for some of the performance metrics.

Aggregated results: Typically, administrators are interested in aggregated information. Therefore, for each performance metric, statistics like the maximum, minimum, average and the cumulative distribution are produced over a given observation window. In case of long-lasting observations, values are given periodically as aggregates over predefined observation intervals. Space and computation overhead to keep track for such aggregated information is expected to be small. Thus, it should be possible to maintain them even at high throughput rates.

4.3 Extensibility

So far, our MaaS prototype derives the performance metrics shown in Table 4.1 for three main services, namely HTTP, MySQL and Memcached, as a proof of concept. However, the MaaS implementation is designed in a modular way that allows for dynamic integration of code components for new communication protocols or new performance metrics that are not yet supported. We present here the high-level idea for extending the MaaS platform for new protocol parsers and performance metrics.

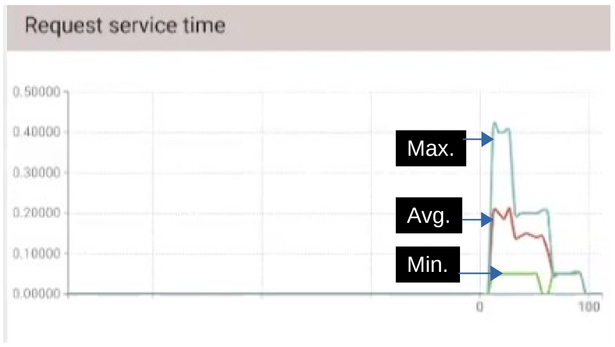
4.3.1 Integrating New Communication Protocol Parsers

As mentioned before, the request/reply-based protocols have a common semantics which enables us to develop a common platform for monitoring them. The MaaS offers a standard interface for request and reply messages, although the format of the messages is then protocol-specific. Thus, our MaaS framework provides an API that allows to define parsers for request/reply-based communication protocols that are not supported yet by the MaaS prototype. The protocol parser API wraps all the common internal structures necessary to encode request and response message formats for any request/reply based services. For instance, the protocol parser API allows for declaring protocol-specific attributes such as request method types, protocol version, response status codes and request/response pairs. The protocol API also contains methods that define how to identify and parse the protocol request and response messages, as well as a list of the supported performance metrics. Furthermore, we have developed a library to parse Ethernet, IP, TCP, and UDP headers. Based on this library and the protocol parser API, new protocol parsers can be written with minimal code. For instance, we have integrated monitoring support for the Redis protocol with only 110 lines of code using the protocol parser API.

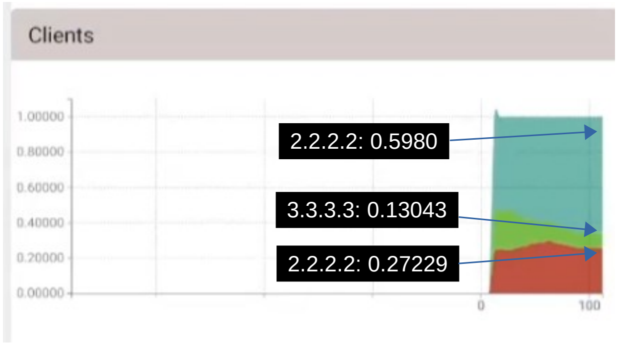
4.3.2 Integrating New Performance Metrics

The MaaS also provides an API to define new performance metrics either by extending/combining existing ones or creating completely new ones. As one performance metric can be useful for different protocols, each performance metric is assigned to different contexts depending on the attached protocol message format. Therefore, the MaaS has also a common API for performance metrics.

Performance metric types: The performance metric API provides two main types of



(a) Example of an average metric.



(b) Example of a cumulative distribution metric.

Figure 4.4: Examples of implemented performance metric types.

metrics:

- *Average metric type* At each time interval, three different values are reported for a particular performance metric: the maximum, average and minimum values over the last time interval. An example of the average metric type is request service time and it is illustrated in Figure 4.4a where three lines of maximum, average and minimum service time are shown. Throughput, connection rate, request rate and error rate are currently defined to have the average metric type in our MaaS prototype.

- *Cumulative distribution metric type* At each time interval, all received values from the start of analysis are displayed with their cumulative distribution. For instance, Figure 4.4b shows an example of a cumulative distribution metric type. In Figure 4.4b, almost 60%, 13% and 27% of the connections to the monitored service since the start of the analysis are belonging to the clients with IP addresses of "2.2.2.2", "3.3.3.3", and "4.4.4.4", respectively. For that, a list of key-value pairs is reported where the keys are the possible values, and the values reflect the cumulative distribution. Response status, request type, path and method all belong to the cumulative distribution metric type in the current implementation of the MaaS prototype.

Events: The performance metric API is implemented as an *event-based system* where data processing takes place at particular key points triggered by a family of events such as receiving new request/response, having a new client connection to the monitored service, etc. Each performance metric registers to some events of interest and provides a notify function to be called for each event it is registered to. For instance, data processing for "service time" performance metric is needed once a new response is received at the monitored service, while the connection rate performance metric is updated once the monitored service receives a new

Event	**Performance metric**
New flow	Connection rate. Clients. Throughput.
Flow update. **(i.e. received a data packet over an already open flow)**	Throughput.
Request received	Request rate. Request type. Request path. Request method.
Response received	Service time. Response status. Error rate.
Timer expired. **(when one second has elapsed. Mainly used for rate computation)**	Throughput. Connection rate. Error rate. Request rate.
Interval elapsed	All requested performance metrics.

Table 4.2: Events used for performance metric update and computation.

connection. Table 4.2 lists the currently implemented events and their related performance metrics.

Data formatting for visualization: Each defined performance metric type has to specify the output format and a function to send this output for visualization. The output format for the two implemented average and cumulative distribution metric types are already defined. The average metric has three output values to produce, which are the average, minimum and maximum, while the cumulative distribution metric type output format is defined to be a hash table of key-value pairs. For instance, the clients performance metric produces the output as a hash table with distinct client IP addresses as the keys, and the total number of connections opened by each client as the values. Any performance metric, extending one of these two already defined metric types, will by default have their output data sent to the visualization frontend, using the message format described above. However, new output formats can be defined as needed.

Defining a new performance metric with one of the existing types is quite easy. For instance, a new metric to measure the request rate per client, i.e. the ratio of one client's requests to all received requests by the server per time unit, extends the cumulative distribution metric type and implements the functions for the relevant events, such as the arrival of new request, timer and interval expiration.

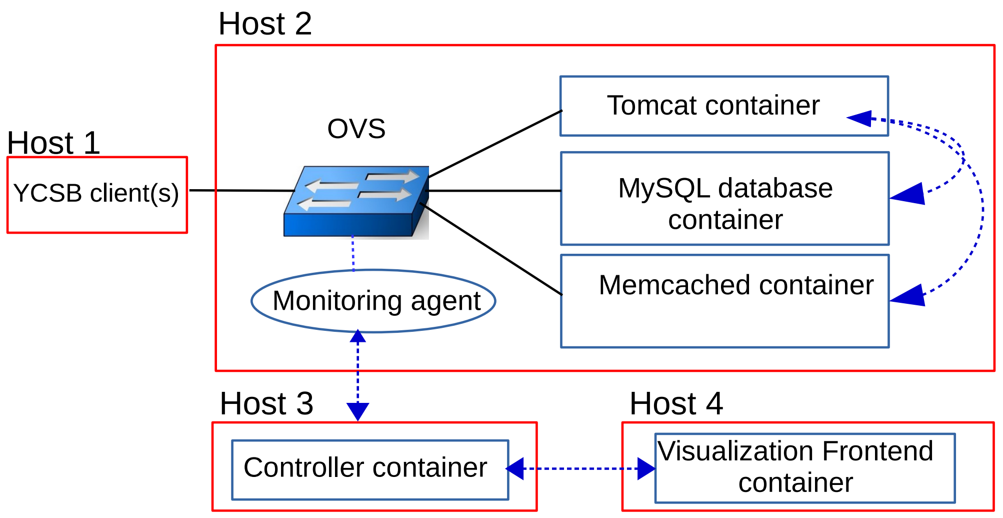


Figure 4.5: Testbed architecture for MaaS prototype.

4.4 Evaluation

In this section we evaluate the overhead the MaaS prototype has in terms of the latency perceived by the clients of the monitored application as well as resource consumption and communication. Our evaluation uses the YCSB benchmark [CST+] described in Section 3.3. We have extended the setup illustrated in Figure 3.6 where four hosts are used to accommodate the MaaS prototype components. That is, the YCSB client is deployed at the first host while the YCSB application components are deployed in separate containers in the second host together with the OVS software switch as illustrated in Figure 4.5. The monitoring agent component of the MaaS prototype is co-located with the host of the OVS switch that connects the YCSB components, i.e. host 2. The third machine hosts the controller component and the fourth machine hosts the visualization frontend component of the MaaS prototype. We use the same workload configuration as described in Section 3.3, that is 16GB of RAM are assigned to the YCSB cache component and the YCSB read-only workload is used with 3 million scan requests and zipfian distribution for records selection over a 10GB database (10 million records). We have tested with 30 client threads, the maximum workload of the YCSB application in our configuration setup. In order to evaluate the overhead of the MaaS prototype, we run our YCSB benchmark with and without MaaS monitoring. For MaaS monitoring, we increase the monitoring overhead by incrementally increasing the number of collected performance metrics. Table 4.3 shows which application performance metrics are collected for each monitoring configuration. All of the listed application performance metrics are collected for the YCSB web-server component. We set the monitoring overall duration and the interval to 120 and 5 seconds,

Monitoring workload	Application performance metrics
1	Service time[a].
2	All of above. Client.
4	All of above. Throughput. Connection rate.
6	All of above. Request rate. Error rate.
8	All of above. Response status. Request path.
10	All of above. Request type. Request method.

Table 4.3: Performance metrics extracted for each monitoring configuration.

[a]Note that we follow the approach depicted in Figure 3.2 for matching the responses with their requests in order to calculate the HTTP request service time.

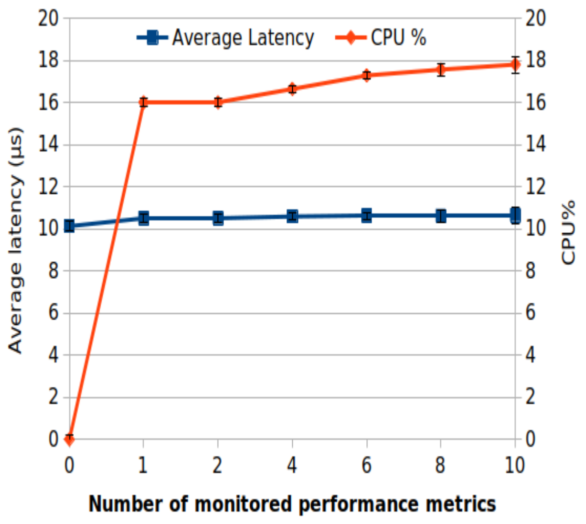
respectively.

The experiments are performed using DELL hosts with dual Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz CPUs (4 cores per socket), a Broadcom NetXtreme BCM5720 Gigabit Ethernet Dual Port NIC, and 32.8GB memory with docker-ce version 18.03.1, OVS version 2.9.90, and MySQL 5.7.24 for the backend database system. All docker containers are connected by 10 Gigabit Ethernet OVS ports.

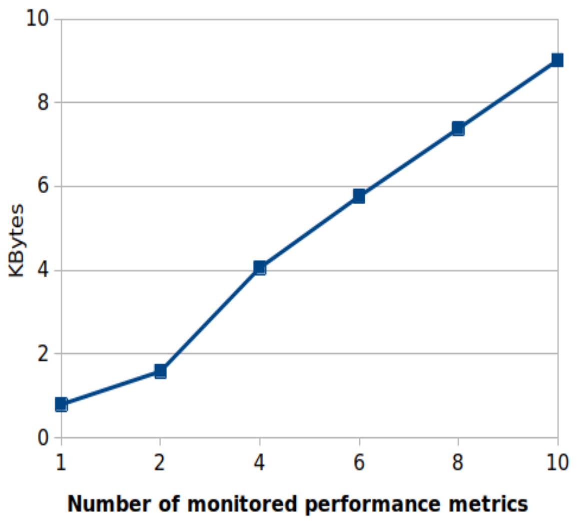
4.4.1 Application Latency

We have examined the impact of the MaaS prototype on the latency at the YCSB client (the blue line). Figure 4.6a shows the end-to-end latency observed by the YCSB client while increasing the number of collected performance metrics. We run each configuration 5 times. We report the latency average of 5 runs and show the error bars in Figure 4.6a.

As we can see in Figure 4.6a, initiating monitoring has a small impact on application latency. That is, when the MaaS prototype is instructed to monitor one performance metric,



(a) Average latency reported by YCSB client and the CPU utilization of the monitoring agent for various monitoring configurations.



(b) Monitoring agent communication overhead for various monitoring configurations.

Figure 4.6: The MaaS prototype overhead

the perceived latency at the YCSB client increases by 3.7%, compared to when no monitoring is enabled. The reason becomes clear when we discuss the computational overhead in the next section. However, the number of collected performance metrics has little further effect on latency. Only a further increase of 1.4% can be observed when the MaaS prototype collects ten performance metrics. The results are very promising and indicate that the MaaS prototype has little client perceived performance impact even with high monitoring demands.

4.4.2 Computational Overhead

In this section we evaluate the computational overhead for the monitoring activity of the MaaS prototype. We ignore the computational overhead of the controller and visualization frontend components as they are remote activities with no impact on the monitored application. We have measured the CPU consumption of the monitoring agent as it needs to run on the OVS host, and in our setup it is the same host where the application components are residing. The Linux top command is used to measure the CPU utilization of the monitoring agent process while increasing the load on the monitoring agent.

The results shown in Figure 4.6a (orange line) indicate the base load of the agent with only one performance metric requires 16% CPU utilization. This includes the parsing of the messages and the maintenance of the various data structures, that does not change much while increasing the number of collected performance metrics. Thus, when more and more performance metrics are collected, the CPU utilization increases further but only slightly. Interestingly, the deep packet inspection processing required for collecting some performance metrics at loads 6, 8 and 10, such as error rate, request path and method increases the CPU consumption not significantly. We note that our experiment in Chapter 3, Figure 3.8 shows a CPU utilization of 13% for the offsite sniffer at similar workload (30 clients and collecting one performance metric). The CPU increase we see here is due to the refactoring we performed to allow for extensibility. Nevertheless, the CPU utilization of the monitoring agent has a minimal impact on the monitored application as also shown in Figure 4.6a.

4.4.3 Communication Overhead

Here we analyze the communication overhead induced by the MaaS prototype. In particular, we have collected the number of transmitted bytes by the monitoring agent during the full 2-minutes monitoring duration. Figure 4.6b shows the number of transmitted bytes while increasing the number of collected application performance metrics. The number of transmitted bytes are linearly increasing with the number of performance metrics as more performance data is reported to the controller for visualization. Recall that at each monitoring interval, the monitoring agent serializes the collected performance data to the controller. In our experiment, this interval was set to 5 seconds. Note that the monitoring agent is reporting three values for each requested average type performance metric and a list of key-value for the cumulative distribution metric type at the end of each monitoring interval as described in Section 4.3, and the size of the latter list is depending on the distinct values collected for the performance metric. For example, the reported list for the "client" performance metric equals the number of connected clients during the monitoring interval. The maximum communication overhead in our test setting is around 9KB for the 2-minute experiment and the monitoring interval of 5 seconds and collecting the complete set of the currently implemented performance metrics while running the maximum YCSB workload. Note that the communication overhead takes place on a dedicated and separate communication link between the monitoring agent and controller components, and only happens at the specific time intervals. We consider this to be a very small number.

4.5 Summary

In this chapter we have presented the design and implementation of the *MaaS* prototype. It is built with the network monitoring functionality proposed in Section 3.2 as core building block, where monitoring agents are co-located with software switches in order to extract performance metrics from the message flows between application components in a non-intrusive manner and send the calculated metrics to the administrators for visualization in near real-time. The developed MaaS prototype allows users to choose to monitor different service types and performance metrics in a user-friendly manner.

The MaaS prototype has a distributed architecture that provides considerable flexibility in the placement of the components within the cloud infrastructure. The MaaS prototype is currently supporting a variety of common application metrics for request/reply-based services. In addition, the MaaS prototype is built in a modular way that allows for the integration of new services and performance metrics to be monitored that are not yet supported. The evaluation results show that the MaaS prototype has little impact on the monitored application latency that is around 5% for the maximum tested monitoring load, has reasonable CPU utilization and imposes a small communication overhead. We believe this monitoring overhead is acceptable since the MaaS prototype will be enabled for short monitoring durations.

5 Flow-based service type Identification using Deep Learning

One of the crucial features of our MaaS is to provide the application call graph of a distributed application at run-time. One important aspect of such a call graph is to identify the service type each of the components provides. While this is an easy task when instrumentation is used, it is not trivial if we only want to rely on the message flows exchanged between the components. In Section 2.6 we have provided an overview of a whole range of mechanisms that were developed in the context of network traffic classification (NTC), and that can potentially be used for service identification as is needed for an MaaS. The recent approaches on using deep learning models (DLMs) seem the most promising as they do not require major feature engineering or deep knowledge of the services that need to be identified. However, the few approaches that we are aware of that have been proposed so far, have not been used for exactly the same purpose that we envision, and differ significantly from each other in regard to what information from the packets they use, and what assumptions they make about the information available. Thus, we needed some guidelines on selecting the best approach. Therefore, this chapter provides a detailed study of the trade-offs between the many design options when building a DLM for service identification, considering, for instance, what part of the message to take as input (header vs. payload-based), whether to choose uni- or bidirectional flows, and whether including port information has an impact. We also analyze the impact of encrypted messages and what happens if we do not have available the first messages in a flow that set up a connection. Furthermore, additionally to looking at a range of DLMs that have been previously used for NTC, we also analyze a model based on Bidirectional LSTM that we believe is particularly promising for service identification.

As such, this chapter provides first a short overview of the principle ideas of deep learning for NTC. We then describe the message flows we have created for a wide range of service types that are commonly used by distributed applications in the cloud. From there, we describe in detail how we transform the message flows into input datasets for learning the DLMs and discuss a wide range of parameters that we consider influential for model learning. We then outline the different DLMs that we consider for our analysis. Finally, we provide a performance comparison of these different deep learning approaches that demonstrates their trade-offs, analyzing the sensitivity of the different approaches to crucial parameters such as dynamic configuration of service port numbers, flow direction, the location of packets in the flow stream and secured communications.

5.1 The Principles of Using Deep Learning for NTCs

As mentioned in Chapter 2, NTCs classify a network flow, that is a sequence of messages, to be of a certain class which can be a service type, an application or an application type. In most NTC approaches, a message flow is first transformed via feature engineering into a feature vector where the individual features provide information about the characteristics of the flow. Example features could be the total number of messages, message inter arrival time, average packet size, etc.

The limited number of deep learning proposals we are aware of take a different approach [LKH$^+$] [LKK$^+$19] [MCSL17] [WZZ$^+$17]. Their idea is to consider the message packet content itself as a feature vector. The sequence of message packets that make up a network flow can then be considered as a matrix with the features as columns and the messages building the rows building a pseudo-image that can be fed into learning algorithms that take images as learning input. Or the flow can be considered as a sequence of features, just as video frames or text, which can be fed to sequence-based learning models. How approaches differ is in what information from the packets they take to build each feature vector. Some approaches only use the packet header [MCSL17] [WZZ$^+$17], while others consider the payload of the packet [LKH$^+$] [LKK$^+$19].

In header-based approaches, the feature vector is extracted from the TCP/UDP header information of the packet. Examples of header-based learning features include source/destination port numbers, number of transmitted bytes, and time passed since the last packet was transmitted. The flows belonging to the same service type should have a local similarity in the values of those header-based features, which enables the DLMs to identify different service types. Header-based analysis has several advantages: (i) it requires small model training time as the number of extracted features are small, and (ii) it is a good option for secured communications since it uses only the packet header information which is not encrypted. However, as the learning features are limited, the STC accuracy might not be good enough. Additionally, when network settings change, a given service type might have different header-patterns and hence retraining becomes necessary. In particular, existing header-based approaches have shown to perform well when including the service port numbers in the learning features. This works well for services with a predefined port but performance is not clear if services are configured to use dynamic port numbers.

In payload-based approaches, the learning features are extracted from the first bytes of the message payload, i.e., the header information of the service type's communication protocol. Thus, the URL string can be found for HTTP-based protocols, and the put/get headers for caching services, etc. This promises to find recurring patterns within the service type's communication protocol. However, the use of packet payload requires longer training time and might have limited identification performance with encrypted payloads.

From here the basic principle is the same as with traditional ML approaches. The DLM is trained by feeding it with a large set of labeled network flows. The trained model is then the classifier with which unlabeled network flows can be classified. Clearly, the choice of DLM has an impact on the classification accuracy of the NTC [PHW11,MCSL17,LKK$^+$19,LKH$^+$].

5.2 Data Generation and Service Types

In this study, we have built datasets by collecting Packet CAPture (PCAP) traces using tcpdump from a wide set of applications that use various service types. From these traces we generate the network flows with the corresponding service labels that are used to train the DLMs. How exactly the network flows are extracted from the traces is described in the next sections.

Having a traditional service architecture in mind, we have aimed at having classical service types such as HTTP-based services, database systems, and caching services in our repertoire. We also wanted to see how good the STC is if the services are conceptually very similar. Overall, we have collected PCAP traces and extracted message flows that were labeled with the following service types: *HTTP*, the four relational database systems *PostgreSQL* [Pos], *MySQL* [MyS], *DB2* [DB2] and *MonetDB* [Mon], the distributed NoSQL data store *Cassandra* [Cas], the two key-value caches *Memcached* [Mem] and *Redis* [Red], and the distributed compute platform *Spark* [Spa]. As Cassandra is a multi-node system we distinguish message flows between clients and Cassandra nodes and flows between Cassandra nodes themselves that are caused by data and group maintenance.

We created these traces by running a wide range of applications. The TeaStore benchmark [vKES+19] consists of six web-services offering functionality such as recommendation, authentication, persistence, etc., and one MySQL database service The YCSB benchmark [CST+] is a database benchmark that we adjusted so that it has a web-based front-end, and we run it with the database systems PostgreSQL, MySQL, DB2, and Cassandra. Furthermore, we developed in-house three small-scale database applications: (i) a Netflix application that is mocking a video streaming service, (ii) a University application that offers course registration for college students, and (iii) a Venues management application that is used to host and organize events for institutions. Each application has its own database, pre-populated with records for manipulation. We run a variety of workloads with different combinations of READ, INSERT, and UPDATE queries and created traces for PostgreSQL, MySQL, DB2, MonetDB, and Cassandra. In addition, we have extended the YCSB and applications developed in-house to not only use database backends but also caching services in order to get traces for Memcached and Redis. The caching systems save a database query result in a key-value pair where the key is a hash code of the database query itself and the value is the query result. The query result is serialized in different formats such as JSON string (JS), LinkedList (LL) and Arraylist (AL) in order to evaluate the capability of DLMs of recognizing the cache service despite the usage of different data formats. Furthermore, we consider Spark as an example of a distributed data processing service that is commonly used to efficiently store and process large datasets in the cloud. For Spark, we do not use the aforementioned applications, but have run several Spark-bench workloads [LTW+17], including machine learning, data generation and heavy-computation workloads. Overall, we have created 15065 network flows by running these applications in their different configurations. For HTTP traffic, we additionally use the dataset provided by the UPC's Broadband Communications Research Group [CBB]. It contains traces for plain web traffic out of popular web applications, including Facebook, Yahoo, Wikipedia, and others. We have extracted a subset of around 5000 unencrypted flows from the UPC dataset. All this data builds our base dataset that we use for most of our evaluations.

In order to analyze the STC capability of identifying the service type in case of secured traffic, we collected a significant number of additional flows for the relational database systems PostgreSQL, MySQL and DB2, and for HTTP, which we assign a different set of service labels (e.g., SDB2 for encrypted DB2). To collect the traces for the database systems, we ran our three in-house applications and the YCSB benchmark using TLS for the client/database connection. TLS stands for Transport Layer Security and is the primary encryption protocol used for HTTP connections, and the most common security protocol when connecting to relational database systems. With TLS, the client and server exchange a series of security parameters prior to a data transfer in what is known as the handshake process which includes verifying each other through authentication TLS certificates, establishing the encryption algorithms they will use, and agreeing on session keys. We have used a variety of TLS versions across clients and databases to introduce variability into our dataset. Newer versions of TLS introduce more secure options for clients and new acceptable cipher-suites (encryption methods). For secured web-based services, we use the public "ISCX VPN-nonVPN (ISCXVPN2016)" traffic dataset, that consists of captured PCAP files for many different applications [DLMG]. We have integrated all the "browsing" PCAP files, that include HTTP flows that are encrypted by TLS (i.e. HTTPs) as well as unencrypted HTTP flows. The ISCXVPN2016 dataset contains HTTPs traffic that is encrypted with different versions of TLS as well. For our experiments analyzing service identification with encryption, we use both the base dataset as well as these additional flows.

5.3 Dataset Preprocessing

After collecting the raw network traces, the next step is to transform them into a *flow-based dataset* that can be fed into the algorithm that trains the DLM. In this section, we describe this transformation and various crucial design parameters. We create network flows out of the raw traces where each network flow and its service label becomes one input record for the model learning process. A network flow is generated for a communication link, that is a pair of two communication endpoints (client and server) defined by their IP addresses and port numbers. This network flow consists of a sequence of feature vectors, where each feature vector represents one of the packets exchanged between the two endpoints. Figure 5.1 illustrates such a network flow. The length N of the sequence, that is the number of packets that are extracted from the traces, is a configurable parameter.

Should the raw network trace contain more than N packets then we extract N packets either from the beginning of the trace, which includes the messages exchanged at the connection setup, or somewhere in the middle of the trace and discard all other packages. We use Min-Max Normalization to scale the feature values between 0 and 1 to enhance the learning process.

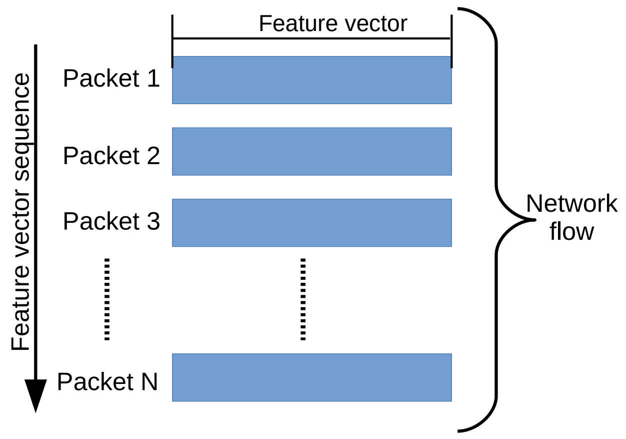


Figure 5.1: The network flow composition in our flow-based dataset.

5.3.1 Header- and Payload-based data extraction

As mentioned in Section 5.1, we have formulated two main dataset types: *header-based* and *payload-based*. In the header-based dataset, only information from the TCP/UDP packet header is taken to create the feature vector. In contrast, for the payload-based data, the first bytes of the payload are used for the feature vector as they contain the header information of the communication protocol of the service type (e.g., HTTP headers).

To represent the network flow profile in the header-based approach, we follow a similar approach to that of [MCSL17], and explicitly extract four meaningful features from each packet's header: the number of bytes in the packet payload, the TCP window size (set to zero for UDP), the packet inter-arrival time, and the packet direction. The latter takes into account that given two endpoints both can send messages to the other. Thus, we mark in which direction the message travels.

In contrast to [MCSL17], our default evaluations use a dataset that does not include the packet port number as a learning feature as we expect that including the port number might lead to poor learning outcome when dynamic port numbers are deployed. However, we have constructed a variant of the header-based dataset that considers the service port number as a fifth feature to investigate its effect on the STC performance.

For our payload-based dataset, we adopt the pre-processing methodology proposed in [LKK+19, LKH+], where each byte of the payload data of a packet is converted into an image pixel (i.e., 256 possible values). According to a pre-defined image size value X, the first X bytes of packet payload are extracted as the pixels of an image. In case the packet payload data is less than X, we use zero-padding to match X. These X pixels thus represent the feature vector for that packet. We discard any packet with no payload such as the flow's control packets.

In both cases, the sequence of feature vectors can be structured as a matrix, with each row representing one packet, and be considered as one pseudo-image, or it could be considered as a sequence of N images. Each image has 4(5) pixels in the case of the header-based approaches or X pixels for the payload-based approaches. This allows us to work with deep learning algorithms that take as an input images, and with algorithms that take a sequence of images.

5.3.2 General Design Parameters

When evaluating the performance of the approaches, we have considered a wide range of parameters for flow extraction. We have fixed some of them after some preliminary testing, the effect of others will be presented in detail in Section 5.5.

Number N of analyzed packets per flow: The length of the flow sequence has shown to be an important parameter [MCSL17, LKK+19]. We considered 20, 60 and 100 packets for each network flow for both header-based and payload-based datasets. Our preliminary results show that longer sequences are good for payload-based models, while header-based approaches work better with lower number of N. The more packets are considered per flow, the more the header-based model has difficulty to distinguish between different service types. It seems that the very few first packets of a flow are the ones that have most of the information required to infer their service for the header-based approaches. The larger number of packets considered per flow, the more the model matches with several service types. Thus, N is set to 20 for the header-based dataset and 100 for the payload-based dataset in the experiments presented here. Note that if there are not 20 resp. 100 packets in the trace to create the flow, then we do not create a record for that flow in the dataset.

Extracted payload size X: For the header-based approach, X is fixed as we explicitly extract the features. In contrast, X is a configurable parameter for the payload-based datasets. Thus, we have tested with extracting the first 9, 12, 16, 20, 25, 36 and 1024 bytes of each packet. Our preliminary results showed that a relatively large packet size is beneficial. Therefore, X is set to 36 in the experiments presented in this chapter. However, compared to [LKK+19], very large X values, such as 1024 bytes, were not beneficial for our STC. We believe the reason is that [LKK+19] classifies different applications, most of them running over HTTP, while we aim in classifying service types, such as HTTP or MySQL. Thus, for us the relevant information can be found in the first few bytes of the payload, which holds the header information of a service type, e.g., the HTTP or MySQL header.

Larger payloads will have a larger portion of application-specific data which is good for application identification, yet misleading for service type identification.

Flow direction: Given a pair of communicating endpoints (e.g., a client and a server), a bidirectional flow reflects the sequence of the packets as they are exchanged in both directions. In contrast, unidirectional flows consider only the packets that go in one direction, and there are typically two such unidirectional flows for each endpoint pair. In our implementation we build first the bidirectional flow for each endpoint pair from the trace containing all packets in proper sequence. From there, we create the corresponding two unidirectional flows, each of them containing all the packets in one direction. Unidirectional flows are guaranteed to have the packets that belong to a single service request or response one after the other in the sequence, while in bidirectional flows they might be interleaved with messages that travel in the other direction. Bidirectional flows might better reflect the timing in handshake protocols at the beginning of a connection or can better correlate requests and responses. In the literature, [LKK+19, LKH+] only construct unidirectional flows and [MCSL17, WZZ+17] only construct bidirectional flows. In contrast, we have created both unidirectional and bidirectional datasets for both our header- and payload-based approaches.

Position of packets in the flow: In the published work both for header- and payload-based datasets, the N packets constructing each flow are always extracted from the beginning of the connection (i.e., when a client connects to the server and starts sending requests). Thus, the flow contains the packets of the handshake protocol to set up the connection. However, the service type identification might become necessary at a random time after establishing a connection, and when sniffing the network packets to create the flow only happens at that time, then the flow does not contain these handshake messages. As this might influence the STC performance, our default datasets contain the first messages exchanged for a connection while derived datasets do not contain these first messages but are extracted from the middle of the flows missing the handshake messages. To do that, we have discarded the first 20 packets of each flow. Note that in all cases we take a consecutive sequence of packets as they occur in the trace.

Secured payloads: As the feature vectors are extracted from the packet header in the *header-based dataset*, having encrypted payload will not have a serious impact on the STC classification performance. On the other hand, having encrypted payloads in the *payload-based dataset* might limit the STC classification performance. Thus, we take a step further and have a preliminary look at how good the STC can detect the service type in case of encrypted communication for the payload-based dataset. In our analysis we first look only at non-encrypted datasets, but then also perform a study on encrypted data.

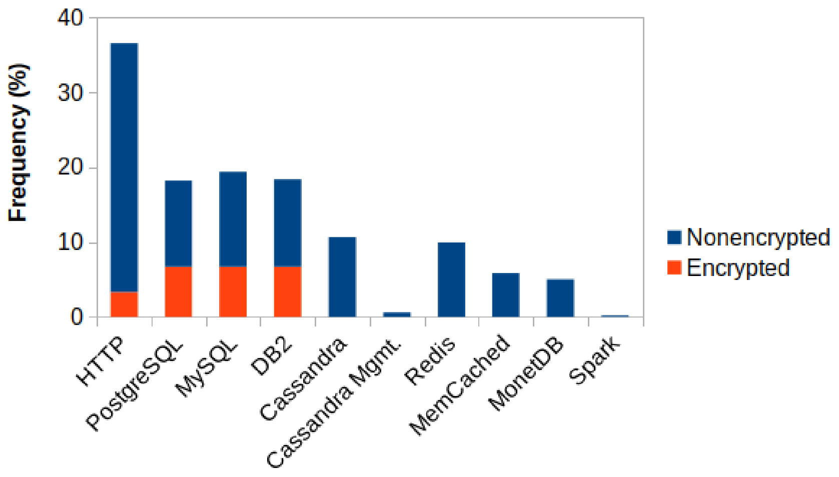


Figure 5.2: Frequency distribution of service types.

5.3.3 Final Flow-based Datasets

Given those considerations, we have constructed two main datasets: one for the header-based approach and the other for the payload-based one. Each flow in the header-based datasets includes a sequence of 20 feature vectors, and each feature vector consists of four features extracted from the packet's header. Each flow in the payload-based dataset contains a sequence of 100 payload images, and each payload image is constructed using the first 36 bytes of packet payload. We have constructed several variations of these two datasets that take into consideration the different design parameters that we want to study such as flow direction and position of extracted packets as just discussed.

In total, our dataset contains around 20,000 unencrypted unidirectional network flows with 10 distinct labeled services, and around 7000 encrypted unidirectional network flows with 4 additional service labels, and correspondingly half the numbers for the bidirectional datasets. Figure 5.2 shows the frequency of each service type and the corresponding percentage of encrypted traffic in our dataset. The flows are annotated with the ten service labels shown at the X-axis in Figure 5.2, in addition to four labels for the secured HTTP, PostgreSQL, MySQL and DB2 flows. HTTP flows, both encrypted and unecrypted, make up around 38% of all our flows. Spark and data management flows between Cassandra contribute less than 1%. The rest is equally distributed among other service types.

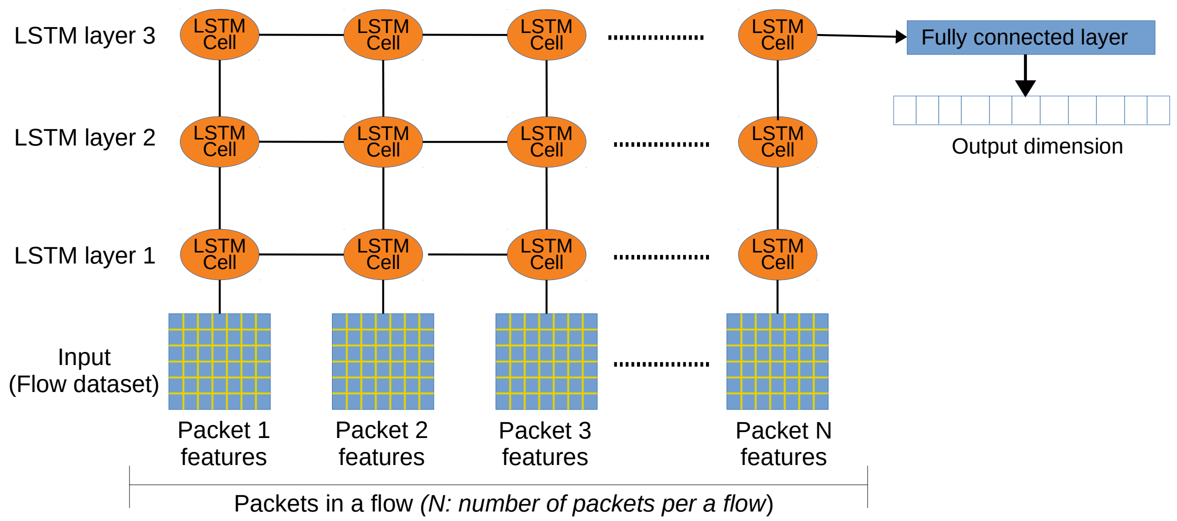


Figure 5.3: Three-layer Long Short Term Memory (LSTM) architecture (figure adapted from [LKK+19]).

5.4 Deep Learning Models

This section describes the different DLMs that we deploy. We have chosen the best performing models from the literature for NTC, namely, multi-layer Long Short Term Memory (LSTM) [LKK+19] and a combined model of convolutional neural network (CNN) and LSTM [MCSL17, LKK+19]. We also propose employing additional deep learning architectures that have not previously been considered for NTC, namely, CNN+Bidirectional LSTM and multi-layer Bidirectional LSTM. These models are widely used for video classification, image classification, speech and text composition.

***-Multi-layer Long Short Term Memory (LSTM)*:** The LSTM model [HS97] is a variant of recurrent neural network (RNN) that is well-suited for time-series data and easier to train. The LSTM model captures important features from inputs and preserves this information over a long period of time through their memory gates. Hence, LSTM utilizes a circulation structure to reflect previous learning data into the current ones for sequential data learning. According to [LKK+19], the three-layer LSTM model architecture provides best application identification results. As LSTM is accepting input in sequence format, the sequence of feature vectors representing the packets of the flow is fed to the input layer of the multi-layer LSTM model where each feature vector is assigned to one cell of the first LSTM layer as shown in Figure 5.3. From there, the data is processed by LSTM layers' cells sequentially until a final classification result is produced. As described in Section 5.3, the feature vector itself is represented as image pixels that are extracted from the packet header

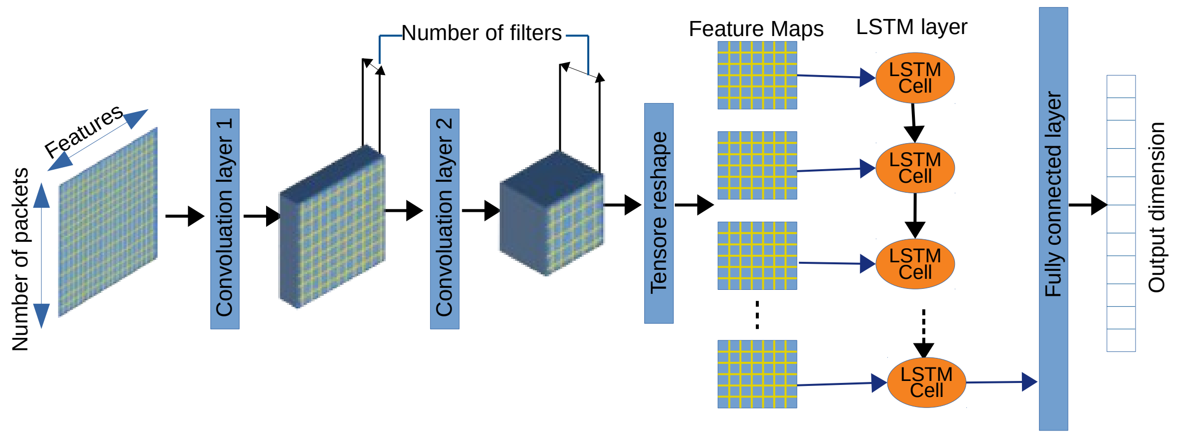


Figure 5.4: Combined convolutional neural network (CNN) and LSTM model architecture (figure adapted from [MCSL17]).

or payload for header-based and payload-based datasets, respectively. We refer to this model as 3-LSTM.

***- Combined CNN and LSTM model*:** This architecture integrates CNN and LSTM models. The feature maps of input data are first extracted through the convolution layer, and then used as a refined sequential data input to the LSTM model as shown in Figure 5.4. CNNs are commonly used for image classification [MCSL17]. Thus, the input is a single image and we use the matrix representation of a flow as input where each feature vector of a message is one row in the matrix, as shown in Figure 5.4. A kernel (filter) action is used to automatically produce feature maps by extracting location invariant patterns from the image. Chaining several CNNs allows automatic extraction of complex features. Interestingly, the matrix formed by the sequence of packets can present a correlated local behavior, similar to traditional images. A reshaping process is then performed on the output of the last convolution layer into a sequence of matrix-like feature maps that can act as the input to the LSTM layer. According to [MCSL17], chaining two CNN layers and one LSTM layer achieves good classification performance. We refer to this model as CNN+LSTM.

***-Combined CNN and Bidirectional LSTM model*:** This architecture combines CNN and Bidirectional LSTMs [GFS05]. The latter is an extension of the traditional LSTM that can improve model performance on sequence classification problems. A Bidirectional LSTM (BiLSTM) connects two hidden regular LSTM layers of opposite directions to the same output as shown in Figure 5.5. The fist LSTM layer is applied on the input sequence (i.e.,

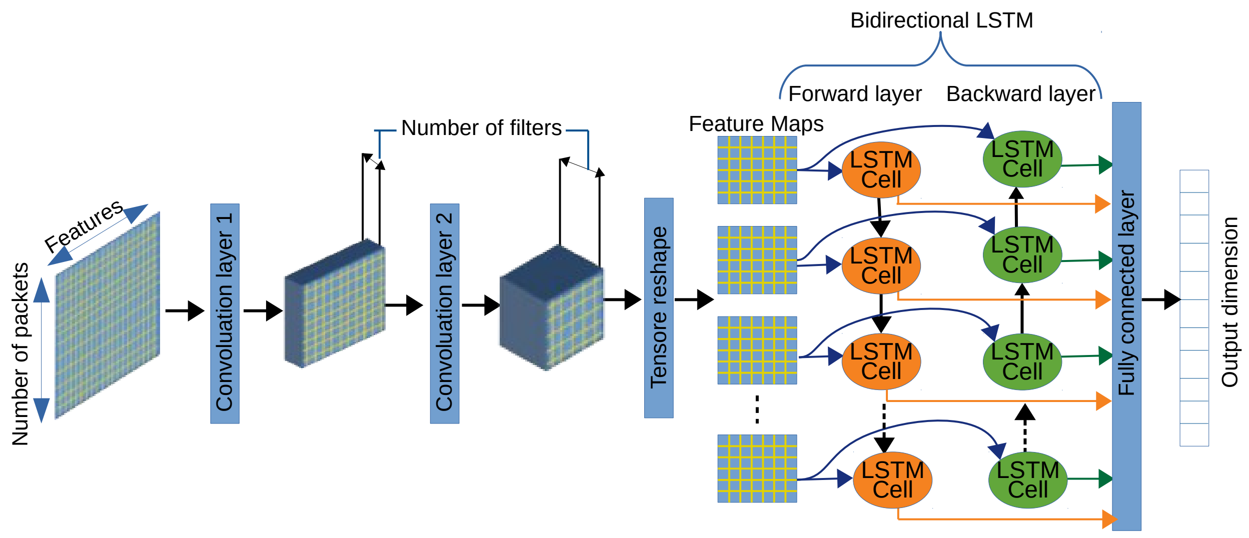


Figure 5.5: Proposed CNN and Bidirectional LSTM model architecture.

forward layer), while the reverse form of the input sequence is fed into the second LSTM layer (i.e., backward layer). Applying the LSTM twice improves the model prediction performance by having an additional layer of learning long-term dependencies in the sequential input, as the Bidirectional LSTM's output layer can get information from past (backward) and future (forward) states. This can provide additional context to the network and result in better learning on the relevant features. As the first layer is a CNN, we provide our input flows again in the form of a matrix. This architecture is referred to as CNN+BiLSTM and is illustrated in Figure 5.5.

***-Multi-layer Bidirectional LSTM*:** Here, we propose chaining multiple layers of Bidirectional LSTMs. In particular, we have constructed single, two and three layer Bidirectional LSTM models to sequentially process the flow sequence data in the datasets, where the output of each Bidirectional LSTM layer is fed as an input to the next Bidirectional LSTM layer. Similar to the multi-layer LSTM model, the feature vectors representing the packets in a flow are fed as a sequence into the input layer of the multi-layer Bidirectional LSTM model where each feature vector is connected to one cell of the forward and backward layers of the Bidirectional LSTM. We refer to this model architecture as M-BiLSTM, where M is the number of layers.

5.5 Experimental Evaluation

This section reports on our experiments to evaluate the STC performance for the header-based and payload-based datasets discussed in Section 5.3, while applying the DLMs described in Section 5.4.

We start by giving a description of the training and validation process applied to our models along with the complexity of the model training time, and subsequently present how good the different models work with the different variations of the header- and payload-based datasets. We first show the classification performance of the different DLM architectures listed in Section 5.4 for both uni- and bidirectional versions of the header- and payload-based datasets. Then we show the impact of including the service port as a learning feature in the header-based dataset, the STC classification performance for each service type, and the impact of the position of extracted packets on the STC performance for both header- and payload-based datasets. All these experiments consider only the records in our flow-based dataset with plain data, i.e. unencrypted payloads. We then take a step further and have a preliminary look at how good the learning model can detect the service type in case of encrypted communication for the payload-based dataset.

5.5.1 Model Training and Validation

In this section we describe the various methods we have used to train and validate the considered models. In addition, we show the required training time for different models and datasets.

Validation Environment The training and validation of the service type identification models is performed on a Ubuntu 16.04 LTS machine with 64GB RAM and two GPU cards (NVIDIA GTX 1080Ti 12 GB), using Keras 2.3.1, Numpy 1.19.5, Pandas 0.24.2, and scikit-learn 0.20.3 with a TensorFlow-gpu 2.1 backend, operated with Python 3.7.7.

Dataset split The first step in our model training and validation process is to split the constructed flow-based dataset into *learning* and *testing* datasets. The learning dataset is used to train, tune and validate the model, while the test dataset is used to evaluate the trained model performance. Table 5.1 indicates for each service type which applications are used for training and which for testing. Note that for each service type, the testing dataset uses traces of applications that are not used for training the DLM.

	YCSB	Teastore	Netflix	University	Venues	Public datasets [CBB, DLMG]	Spark-bench workloads [LTW+17]
HTTP	Test.	Test.	-	-	-	Train (67%) Test (33%) (TLS 1.1 &1.2).	-
PostgreSQL	Train/Test (TLS 1.3)	-	Test (TLS 1.2).	Train (TLS 1.2).	Train (TLS 1.2).	-	-
MySQL	Train/Test (TLS 1.3)	Test.	Test (TLS 1.2).	Train (TLS 1.2).	Train (TLS 1.2).	-	-
DB2	Train/Test (TLS 1.1)	-	Test (TLS 1.1).	Train (TLS 1.1).	Train (TLS 1.1).	-	-
MonetDB	-	-	Test.	Train.	Train.	-	-
Cassandra	Train/Test.	-	Test.	Train.	Train.	-	-
Memcached	Train/Test.	-	Test (JS, AL and LL).	Train (AL) Test(LL & JS)	Train (AL) Test(LL & JS)	-	-
Redis	Train/Test.	-	Test (AL & JS).	Train (AL) Test(JS).	Train (AL) Test(JS).	-	-
Spark	-	-	-	-	-	-	Train (KMeans) Train (SparkPi (67%)) Test (Linear regression) Test (SQL) Test (SparkPi(33%))

Table 5.1: Services and applications used for model training and testing.

For our learning dataset, the flows need to be labeled with the correct service. Since the network flows are extracted from applications we deployed and developed, we are aware of the port number assigned for each service in each application (the default port for the service are used in most situations). For example, 3306 is used for MySQL. Thus, we label flows with port number 3306 as MySQL. Furthermore, We have collected the PCAPs as per each service type. With this in mind, we assume the output of our labeling tool is a best approximation to the ground-truth service. Using such controlled and deterministic environment to have a ground-truth labeling for network datasets is common in the literature [Net12, ERP+21, CVW+21].

Models Tuning Each of the models has various hyper-parameters that determine the network structure (e.g., number of filters) and how the DLMs are trained (e.g., type of optimizer). The performance of a model can vary considerably according to the selected set of hyper-parameters. Finding the optimal hyper-parameters is a time-consuming tasks for any DLM. We use the sequential model-based optimization SMBO [TP12] to find the optimal hyper-parameters for the aforementioned DLMs with respect to our datasets. SMBO is a Bayesian optimization technique that uses information from past trials to refine the next set of hyper-parameters to explore based on the given dataset. Since Bayesian optimization is not a brute force algorithm as compared to manual, grid and random search, it is a good candidate for efficiently performing hyper-parameter optimization while maintaining the quality of the results.

Hyper-parameter	**Values**
Number of filters	32, 64
Kernel size	3×3, 5×5, 7×7
Kernel initializer	normal, uniform, glorot_uniform
Recurrent initializer	normal, uniform, glorot_uniform
Kernel regularizer	0.0, 0.01, 0.03
Recurrent regularizer	0.0, 0.01, 0.03
Dropout rate	0.0, 0.2, 0.3, 0.4
Output activation type	tanh, relu, softmax
Output size	64, 128, 256
Optimization type	adam, rmsprop
Batch size	32, 64, 100
Epochs	50, 100, 200

Table 5.2: Examined parameter space

We have considered a total of ten common standard hyper-parameters as listed in Table 5.2 for SMBO optimization for all described models in Section 5.4. We have additionally investigated two hyper-parameters for CNN+LSTM and CNN+BiLSTM, which are the *number of filters* in each CNN layer and *kernel size* used in each filter.

Table 5.3 lists all hyper-parameters chosen by SMBO for the payload-based dataset and the different models. The values for the header-based dataset are basically the same except for the 1-BiLSTM model, where the LSTM layer units, output size and epochs values are 256, 128 and 200, respectively and the optimization method is Rmsprop.

Models validation We have validated the produced hyper-parameters by *k-fold cross-validation* [TGB18]. In k-fold cross-validation, the dataset is randomly partitioned into k sub-datasets. Then, a single sub-dataset is retained as the validation data for testing the model, and the remaining $k-1$ sub-datasets are used as training data. The cross-validation process is then repeated k times, with each of the k sub-datasets used exactly once as the validation data. The k results can then be averaged to produce a single estimation. The advantage of this method is that all observations are used for both training and validation, and each observation is used for validation exactly once. For all models, we set the CNN layers' activation function to Rectified Linear Units (ReLU), the hyperbolic tangent function as the activation function of the LSTM layers, and the learning rate of the optimizer to 0.001.

In summary, we separate each flow-based dataset into learning and test datasets. Further, we separate 20% of the learning data for validation, and the verification of the model is performed on the basis of the given SMBO hyper-parameter set and the k-fold value. Then,

	3-LSTM	CNN+LSTM	CNN+BiLSTM	1-BiLSTM	2-BiLSTM	3-BiLSTM
Kernel size of 1st layer	NA	7X7	7X7	NA	NA	NA
Kernel size of 2ed layer	NA	3X3	3X3	NA	NA	NA
Number of filters	NA	64	64	NA	NA	NA
Kernel initializer	G	U	U	G	G	G
Recurrent initializer	G	U	U	G	G	G
Kernel regularizer	0.0	0.03	0.03	0.0	0.0	0.0
Bias regularizer	0.0	0.03	0.03	0.0	0.0	0.0
Recurrent regularizer	0.0	0.03	0.03	0.0	0.0	0.0
1st LSTM layer units	256	64	64	128	64	256
2ed LSTM layer units	256	NA	NA	NA	256	256
3rd LSTM layer units	128	NA	NA	NA	NA	128
Dropout rate of 1st layer	0.2	0.4	0.4	0.2	0.2	0.4
Dropout rate of 2ed layer	0.3	0.2	0.2	NA	0.2	0.2
Dropout rate of 3rd layer	0.4	NA	NA	NA	NA	0.3
Output activation type	S	S	S	S	S	S
Output size	64	256	256	256	256	64
Optimization type	A	A	A	A	R	A
Batch size	100	32	32	100	100	100
Epochs	100	100	100	100	200	100

Table 5.3: The optimal hyper-parameter values for the payload-based dataset (U: uniform, N: normal, G: glorot uniform, S: softmax, R: rmsprop, A: adam).

	Header-based dataset		**Payload-based dataset**	
	Unidirectional flows	**Bidirectional flows**	**Unidirectional flows**	**Bidirectional flows**
3-LSTM [8]	89.383	42.001	178.505	156.567
CNN+LSTM [1]	177.276	55.714	286.817	130.528
CNN+BiLSTM	183.835	56.225	373.106	169.644
1-BiLSTM	113.427	54.891	153.570	65.837
2-BiLSTM	304.489	104.17	568.37	253.859

Table 5.4: Training time (in seconds) of different DLMs for header-based and payload-based datasets.

the model is trained by using the optimal hyper-parameters and the model performance is evaluated using the test data.

Training time We have compared the time required for training the final models. Table 5.4 shows the training time for the different models on the different datasets. We can see that the training time increases with the increase in the number of learning features (header-based has 4 compared to 36 in payload-based), flows in the training dataset (unidirectional has double as many flows as bidirectional) and model network complexity. In terms of model network complexity, the number of model layers and their type influence the model training time. While 3-LSTM and CNN+LSTM have the same number of layers, the latter requires more model training time for most of the datasets. In general, the Bidirectional LSTM layers are more complex to train than the regular LSTM because they require training double the number of layers compared to regular LSTMs. For the bidirectional header-based dataset using CNN+LSTM, replacing the LSTM layer with a Bidirectional LSTM increases the training time by less than 1%, while the training time of CNN+BiLSTM is 30% higher than that of CNN+LSTM for the bidirectional payload-based dataset, which has more learning features. Looking at the approaches without CNN, adding more Bidirectional LSTM layers significantly increases the training time for all datasets. 1-BiLSTM has lower learning time for the payload-based datasets, but it has increased learning time for header-based datasets.

5.5.2 Performance Metrics

Before we show in the next sub-sections how good the different models predict the service types, we introduce the performance metrics we have considered in our evaluation.

Our setup is, in principle, a multi-class classification problem. For each class label L, the true positives TP are the flows of type L that are correctly identified and the true negatives TN are all the flows of type $L' \neq L$ that are correctly identified as not being of type L. On the other hand, the false negatives FN are flows of type L that are incorrectly labeled as not L, and the false positives FP are flows not of type L that are wrongly classified as L.

The performance is measured using the following metrics: (i) *accuracy* which is the ratio of all correctly classified flows to all flows, (ii) *recall* indicates the ratio between the correctly identified flows of type L to all flows that actually belong to type L, (iii) *precision* is the ratio of the correctly identified flows of type L to all flows that are identified of being of type L, and (iv) *F1-score* which is the *harmonic mean* of precision and recall, and does not include the number of true negatives. The following equations show the mathematical representation of these four performance metrics:

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$F1 - score = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{4}$$

Among all metrics, the F1-score is considered more meaningful in scenarios where the dataset is unbalanced, i.e. when there are classes with many samples and classes with few samples. All the performance metrics have as best value 1 and as worst value 0. In many of the figures in this section, we show the performance metrics aggregated over all possible classes using a *weighted average*. Weighted average considers the number of samples in each class when calculating the average. That is, a class with few samples has less of an impact on the weighted average than other classes with more samples.

$$Weighted\ average(P) = \sum_{i=1}^{n} P_i w_i / \sum_{i=1}^{n} w_i \tag{5}$$

P is one of accuracy, precision, recall or F1-score, n is the number of different classes in the dataset, P_i is the performance metric value for class i, and w_i is the number of samples in class i. The weighted average is a more accurate representation of the model aggregated performance than the normal average, in particular for unbalanced datasets.

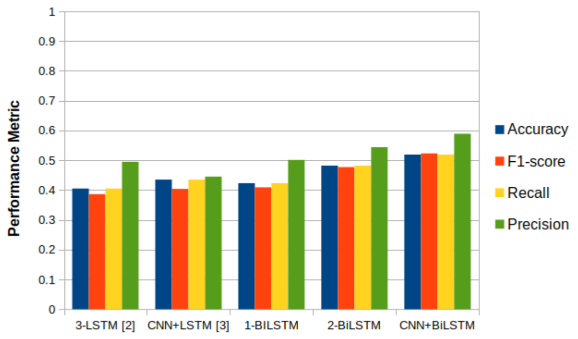
We have used scikit-learn[1] to calculate the weighted F1, precision, and recall scores. We run each test five times and report the average of those aggregated metrics. The standard deviation of the reported averages are always below 0.1.

5.5.3 Performance Comparison of the Different DLMs

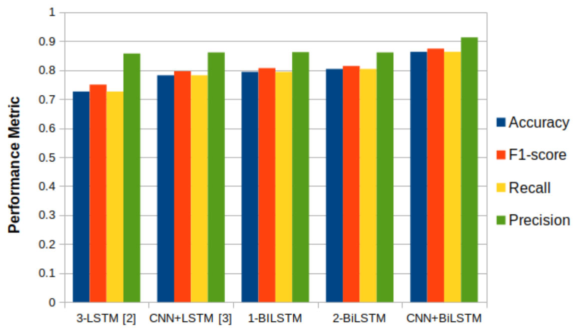
In our first experiment, we apply the different DLMs described in Section 5.4 to the datasets discussed in Section 5.3 where the flows contain the first handshake messages that are exchanged when the connection is setup, and we provide the performance metrics averaged over all service types using Equation 5. This will give us a first impression how well the different DLMs perform for header- and payload-based datasets. We do not report the results of the 3-BiLSTM model given their similarity to those of the 2-BiLSTM model.

a) Header-based dataset: Figure 5.6 shows the aggregated service type identification performance for header-based datasets with uni- and bidirectional flows. Using unidirectional flows for the header-based dataset achieves maximum accuracy and F1-score of around 52% using the CNN+BiLSTM model (Figure 5.6a). Meanwhile, the bidirectional flows improve the STC performance to be higher than 72% accuracy and 75% F1-score for all models as shown in Figure 5.6b. It seems that the correlation between the incoming and

[1] `https://scikit-learn.org/stable/`



(a) Unidirectional datasets.



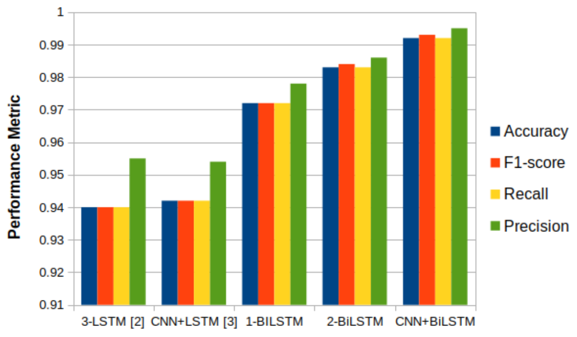
(b) Bidirectional datasets.

Figure 5.6: Aggregated performance for the header-based datasets.

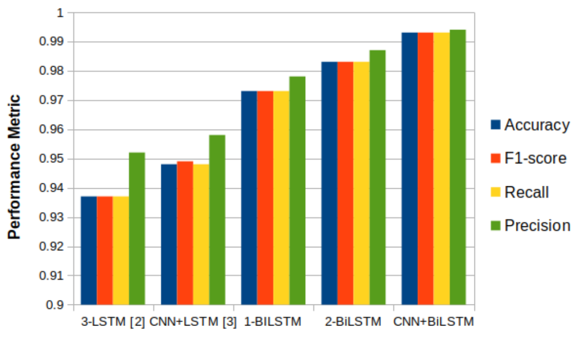
outgoing messages for a service type is crucial for better recognition of the service type in the header-based dataset. Combining CNN with LSTM improves the classification performance in terms of F1-score compared to the multi-layer LSTM model for both variants of the header-based dataset, with more improvement seen in the bidirectional variant. Replacing the LSTM layer by a BiLSTM layer in that combined model, i.e. CNN+BiLSTM, further increases the F1-score by 12% and 7.8% for the uni- and bidirectional header-based datasets, respectively. However, discarding the CNN layers and having only the BiLSTM layer in 1-BiLSTM achieves around 11% and 6.7% lower F1-score compared to CNN+BiLSTM for the uni- and bidirectional header-based dataset, respectively. Adding more BiLSTM layers improves the F1-score for the unidirectional flows while only a slight increase is seen for the bidirectional flows. Overall, 2-BiLSTM and CNN+BiLSTM models provide the best results for the bidirectional header-based dataset with accuracy and F1-score higher than 80%.

Service port as a learning feature: Recall that by default, we do not consider the endpoint's port numbers as a feature. To understand their impact, we have regenerated our bidirectional training data to include the default port numbers for each service and retrained our models.

We consider two types of test data; the first contains the same service port numbers as of the training dataset, while the second has alternative standard port numbers for each service, e.g., 8079 instead of 8080 for HTTP and 33060 instead of 3306 for MySQL. We run the two tests against CNN+BiLSTM, the best model in our previous experiment. The first test data results in 94% accuracy and 93% F1-score, while using slightly alternative standard port numbers for the services negatively affects the model performance and decreases the identification accuracy and F1-score to around 60% and 57%, respectively, that is worse than if the port numbers are not part of the feature vector at all. This shows that the performance



(a) Unidirectional datasets.



(b) Bidirectional datasets.

Figure 5.7: Aggregated performance for the payload-based datasets.

of header-based DLMs that consider the service port numbers is tightly correlated to the static service configuration, and if production systems do not use standard ports, it is better to not include them in the learning process.

b) Payload-based dataset: Figure 5.7 shows the aggregated service type identification performance of the evaluated DLMs for payload-based datasets with unidirectional (Figure 5.11a) and bidirectional flows (Figure 5.11b). In general, for all models, the classification performance of unidirectional and bidirectional flows is quite similar. The CNN+BiLSTM achieves the highest performance for both variants of payload-based dataset with accuracy and F1-score always higher than 99%. Employing one BiLSTM layer performs better than using three-layer regular LSTM. Adding more BiLSTM layers only slightly improves the classification performance for both variants of the payload-based dataset. However, combining one BiLSTM layer with CNN increases the classification performance by around 2% and 5% for most of the performance metrics compared to 1-BiLSTM and CNN+LSTM, respectively.

c) Summary: In general, payload-based approaches significantly outperform the header-based ones with accuracy and F1-score always higher than 93%. Furthermore, for header-based approaches, working on bidirectional flows that correlate the incoming and outgoing messages of the service type is important for high service type identification accuracy, which is not so important for payload-based approaches to work well. Adding CNN and BiLSTM layers boosts the classification performance compared to the multi-layer regular LSTM model for both variants of the header- and payload-based datasets. CNN+BilSTM can achieve the best service type identification performance with F1-scores above 87% and 99% for the bidirectional header-based dataset and both variants of the payload-based dataset, respectively.

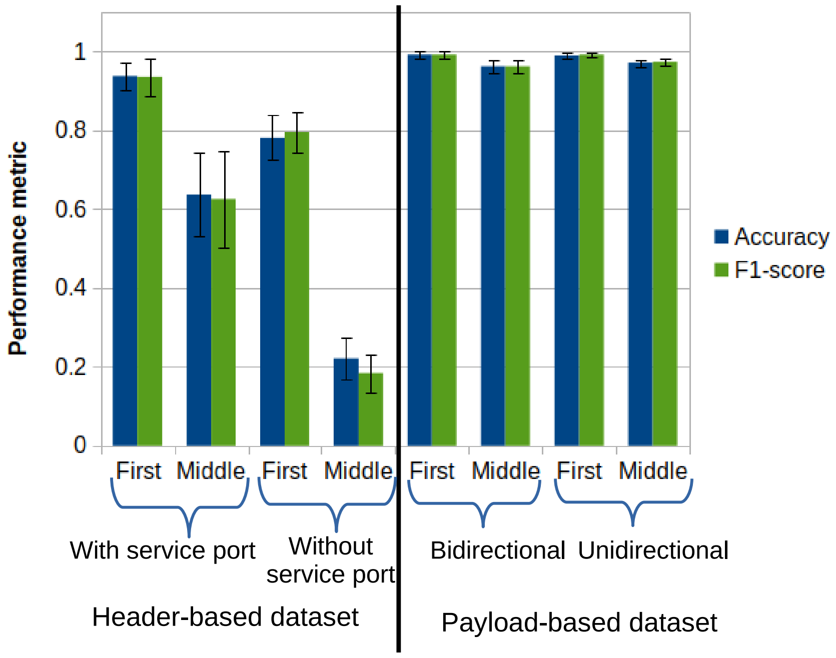
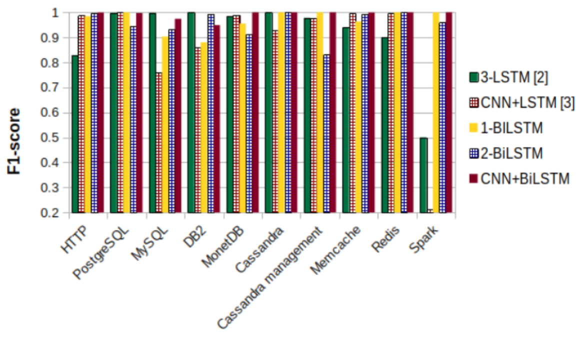


Figure 5.8: The impact of the extracted packets' position in the flow in both header-based and payload-based datasets.

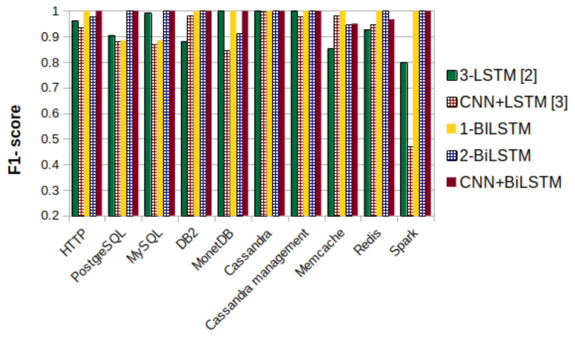
5.5.4 The Impact of the Packet Position in the Flow

The results so far have used the datasets where the flows are extracted from the beginning of the connection, that is, they contain the packets exchanged for connection setup. We now want to compare this with the case when the service identification is required at random time after establishing a connection as detailed in Section 5.3.2. To have a fair evaluation for the header-based dataset, we have considered two scenarios: 1) The feature vectors include the standard service port numbers both for training and test data, thus evaluating the impact of the packet position in the best case for header-based models, and 2) the feature vectors do not contain the port information, which is the better solution if applications generally do not use standard ports. Figure 5.8 shows the aggregated accuracy and F1-score for the bidirectional variant of the header-based dataset, and both the uni- and bidirectional variants for the payload-based dataset, employing the CNN+BiLSTM model. We denote results for when the flow contains the first handshake messages as "First", and when it does not contain those handshake messages as "Middle". We have excluded the unidirectional header-based dataset from this and all further experiments, as we have shown in Section 5.5.3 that its classification performance is considerably worse than the bidirectional variant.

In the header-based dataset that considers port numbers, not having the first messages of



(a) Unidirectional dataset.



(b) Bidirectional dataset.

Figure 5.9: F1-score for individual service types for the payload-based datasets.

a flow decreases the F1-score by 23%, while around 75% F1-score loss happens in the header-based dataset that excludes the port numbers. This shows that header-based approaches in scenarios with dynamic ports can only work reasonably well if they can learn the service type's flow characteristics from the first handshake packets between the service and client. Once the model misses those first packets from the service flow, it is no more able to recognize the service type. But even if standard ports are used and the model learning considers them, not having access to the handshake messages significantly reduces performance. On the other hand, the payload-based dataset performance is only slightly affected when skipping the first packets in the flow. This is because the model can infer the service type from the service type headers in the request and response messages. This holds for both uni- and bidirectional formats.

5.5.5 Performance on a Per-service Basis

We have now a closer look at the performance of the individual services.

a) Payload-based dataset: Figure 5.9 shows the F1-score for each service label achieved by the different models for both the uni- and bidirectional variants of the payload-based dataset. We had seen in Figure 5.7 that the average F1-score over all service types for the payload-based dataset is high for all models. Figure 5.9 now shows that for both uni- and bidirectional payload-based datasets most models have relatively little variation among the different services with CNN+LSTM being an exception as it has poor performance for Spark in both unidirectional and bidirectional variants. 3-LSTM also has low classification performance for Spark in the unidirectional payload-based dataset (Figure 5.9a). Nevertheless, the majority of services are classified correctly most of the time. This is

		Predicted											
		HTTP	**PostgreSQL**	**MySQL**	**DB2**	**MonetDB**	**Cassandra**	**Cassandra-MG**	**Memcached**	**Redis**	**Spark**	**Sum**	**F1-score**
Actual	**HTTP**	193	3	0	0	1	0	0	1	1	0	198	0.987
	PostgreSQL	0	252	0	0	0	0	0	0	0	0	252	0.906
	MySQL	0	0	200	0	0	0	0	0	0	0	200	1
	DB2	0	42	0	158	0	0	0	0	0	0	200	0.882
	MonetDB	0	7	0	0	193	0	0	0	0	0	200	0.982
	Cassandra	0	0	0	0	0	402	0	0	0	0	402	0.995
	Cassandra-MG	0	0	0	0	0	0	22	0	0	0	22	1.0
	Memcached	0	0	0	0	0	0	0	336	115	0	451	0.853
	Redis	0	0	0	0	0	0	0	0	738	0	738	0.927
	Spark	0	0	0	0	0	4	0	0	0	8	12	0.800
	Overall F1-score												0.949

Table 5.5: Confusion matrix for 3-LSTM with bidirectional payload-based dataset.

		Predicted											
		HTTP	**PostgreSQL**	**MySQL**	**DB2**	**MonetDB**	**Cassandra**	**Cassandra-MG**	**Memcached**	**Redis**	**Spark**	**Sum**	**F1-score**
Actual	**HTTP**	198	0	0	0	1	0	0	0	0	1	198	1
	PostgreSQL	0	200	52	0	0	0	0	0	0	0	252	0.884
	MySQL	0	0	200	0	0	0	0	0	0	0	200	0.884
	DB2	0	0	0	200	0	0	0	0	0	0	200	1
	MonetDB	0	0	0	0	200	0	0	0	0	0	200	1
	Cassandra	0	0	0	0	0	402	0	0	0	0	402	1.0
	Cassandra-MG	0	0	0	0	0	0	22	0	0	0	22	1.0
	Memcached	0	0	0	0	0	0	0	451	0	0	451	1
	Redis	0	0	0	0	0	0	0	0	738	0	738	1
	Spark	0	0	0	0	0	0	0	0	0	12	12	1
	Overall F1-score												0.976

Table 5.6: Confusion matrix for 1-BiLSTM with bidirectional payload-based dataset.

interesting as all database systems are exchanging similar content (they all receive the same SQL queries and return the same results). The same holds true for the two caching systems that have been analyzed, where their communication protocols have sufficient differences for the DLMs to distinguish between them.

Most of the models perfectly identify the Cassandra management and Spark flows despite the very few number of flows for learning given for these two classes. One reason for this might be that the communication between Cassandra nodes is likely very different to standard request/reply protocols, and thus, identifiable even with the low number of training records.

It is also interesting to have a closer look at some of the incorrectly classified data. Table 5.5, 5.6 and 5.7 show confusion matrix examples with the bidirectional payload-based dataset for 3-LSTM, 1-BiLSTM and CNN+BiLSTM, respectively. The rows correspond to the true values for each flow, and the columns correspond to the predicted values. All numbers on the diagonal are correctly predicted values (true positives). Any number off a diagonal has been incorrectly classified as the column service type while true service type is the row value. This can be thought of as the false positives for the column service type, and the false negatives for the row service type.

We can observe that the FNs of some database systems went to other database systems. For example, 21% of DB2 flows were misclassified as PostgreSQL by the 3-LSTM model

		Predicted											
		HTTP	PostgreSQL	MySQL	DB2	MonetDB	Cassandra	Cassandra-MG	Memcached	Redis	Spark	Sum	F1-score
	HTTP	198	0	0	0	0	0	0	0	0	0	198	1
	PostgreSQL	0	252	0	0	0	0	0	0	0	0	252	1
	MySQL	0	0	200	0	0	0	0	0	0	0	200	1
	DB2	0	0	0	200	0	0	0	0	0	0	200	1
Actual	MonetDB	0	0	0	0	200	0	0	0	0	0	200	1
	Cassandra	0	0	0	0	0	402	0	0	0	0	402	1
	Cassandra-MG	0	0	0	0	0	0	22	0	0	0	22	1
	Memcached	0	0	0	0	0	0	0	451	0	0	451	0.950
	Redis	0	0	0	0	0	0	0	47	691	0	738	0.967
	Spark	0	0	0	0	0	0	0	0	0	12	12	1
	Overall F1-score												0.992

Table 5.7: Confusion matrix for CNN+BiLSTM with bidirectional payload-based dataset.

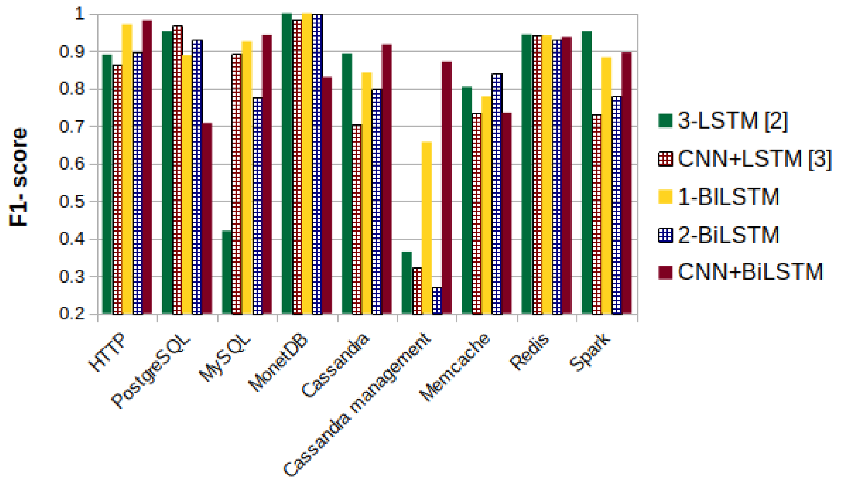
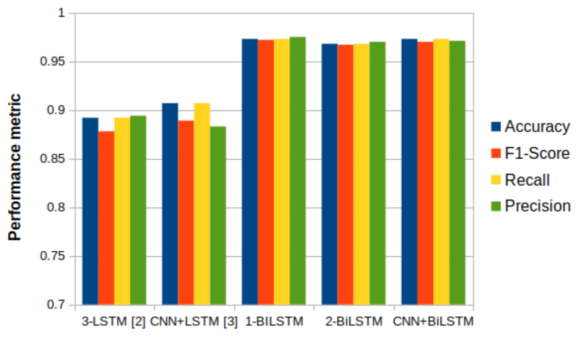


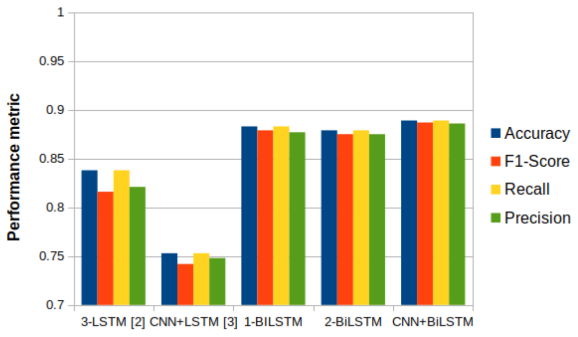
Figure 5.10: F1-score for individual service types for the header-based bidirectional dataset.

(Table 5.5), while around the same percentage of PostgreSQL flows were misclassified as MySQL by the 1-BiLSTM model (Table 5.6). We found a similar pattern for Memcached and Redis. For instance, the CNN+BiLSTM model misclassified around 6% of Redis cases as Memcached (Table 5.7) while the 3-LSTM model misclassified around 25% of Memcached flows as Redis (Table 5.5). While still being a false negative, classifying a service as being of the "same kind", in our case a database system or a cache, might be viewed as more acceptable than classifying it as something completely different.

b) Header-based dataset: For the header-based dataset, Figure 5.10 shows a lot more variation, and Cassandra management flows are not well classified by most models. Here, the lack of a large number of flows might make it harder to learn the proper patterns. When inspecting the misclassification cases, we found them to happen across all services with no clear preferences for services that are similar. It appears that the header data does not reveal too much commonality among services of the same kind.



(a) Δ_1 dataset.



(b) Δ_2 dataset.

Figure 5.11: Aggregated performance for the encrypted bidirectional payload-based datasets.

5.5.6 Secured Payloads

In this section we have a look at how good the learning models can detect the service type in case of encrypted communication for the payload-based dataset as detailed in Section 5.2. We took the bidirectional payload-based dataset and considered both the dataset that extracts the messages from the beginning of the connection, denoted as Δ_1, as well as the one that extracts the messages from the middle of the connection, denoted as Δ_2.

a) Aggregated Performance: We first apply the different DLMs described in Section 5.4 to both datasets Δ_1 and Δ_2, and we provide the performance metrics averaged over all service types using Equation 5. This will give us a first impression how well the different DLMs perform for classifying secured traces. Figure 5.11 shows the aggregated accuracy, F1-score, recall and precision for each of the two datasets and the five DLMs we consider. The models that contain at least one BiLSTM layer have a F1-score at least 5% higher than the other models for the two datasets and outperform CNN+LSTM by more than 12% for all metrics on Δ_2. CNN+LSTM and 3-LSTM have similar performance when they have access to the handshake messages (i.e., Δ_1). However, CNN+LSTM performs significantly worse on mid-flow traces (Δ_2). All models perform worse when the packet payload is encrypted and when handshake messages are missing; yet the models with at least one BiLSTM layer remain above 87% for all performance metrics, which is significantly higher than the other competing models. The models with only one BiLSTM layer are performing slightly better than the models with more BiLSTM layers for both Δ_1 and Δ_2 datasets. The reason is not clear to us. We would like to note that the performance for Δ_2 is still considered better than the header-based approaches for middle-flow packets.

b) Performance on a per-service basis We have now a closer look at the performance for the individual services. Figure 5.12 shows the F1-score results for the different models and the individual services while learning from the *first* packets of traffic flows, i.e., Δ_1. Interestingly, most of the models are still capable to identify most service types with a F1-score of more than 91%. Spark flows got a lower F1-score with most of the models compared to what we saw in Figure 5.9b where there is no encrypted data at all. We believe that Spark identification performance is highly related to the number of its learning records in the dataset. The Δ_1 dataset has more records as well as service classes compared to the dataset used in Section 5.5.5, while the number of Spark learning flows is still low. On the other hand, Cassandra management flows are still well recognized in the Δ_1 dataset despite the small number of Cassandra management records. It seems that Cassandra management communication protocol is quite different from the other service types in our dataset so that the model can perfectly identify it.

When looking at the confusion matrices (not shown), we could determine mainly two types of misclassifications:

- **Misclassifications between database systems:** Again, some of the misclassifications happened between the database systems as seen before in Section 5.5.5. For instance, 3-LSTM wrongly classifies PostgreSQL flows as Cassandra, which in turn lowers the F1-score of Cassandra. The CNN+LSTM model does a misclassification between the SMySQL and SPostgreSQL, where around 66% of SPostgreSQL flows went to SMySQL while 4% of SMySQL flows are misclassified as SPostgreSQL ones. Again, miscalssification between different database systems might be more acceptable than misclassifying, for instance, PostgreSQL as a HTTP-service.
- **Misclassifications between HTTPS and HTTP** All the models have misclassified some of the HTTPS flows as HTTP. For instance, around half of the HTTPS flows are classified as HTTP by the CNN+LSTM model, while around 26% and 35% of the secured HTTP flows are misclassified as plain HTTP by 1-BiLSTM and CNN+BiLSTM models, respectively. We consider misclassifications that happen between the secured and unsecured version of the same service type are acceptable in our context even more than misclassifying one database system for another. According to an analysis performed in [RKL20] to investigate why the secured traffic of HTTP is still identifiable by a multi-layer CNN model, the results show that unencrypted TLS handshake fields, including cipher information and other security parameters which are part of the payload information, reveal enough information for identifying such secured HTTP traffic.

Figure 5.13 shows the F1-score results of the different models while learning from the *middle* packets of traffic flows, i.e., Δ_2. The classification performance of all models

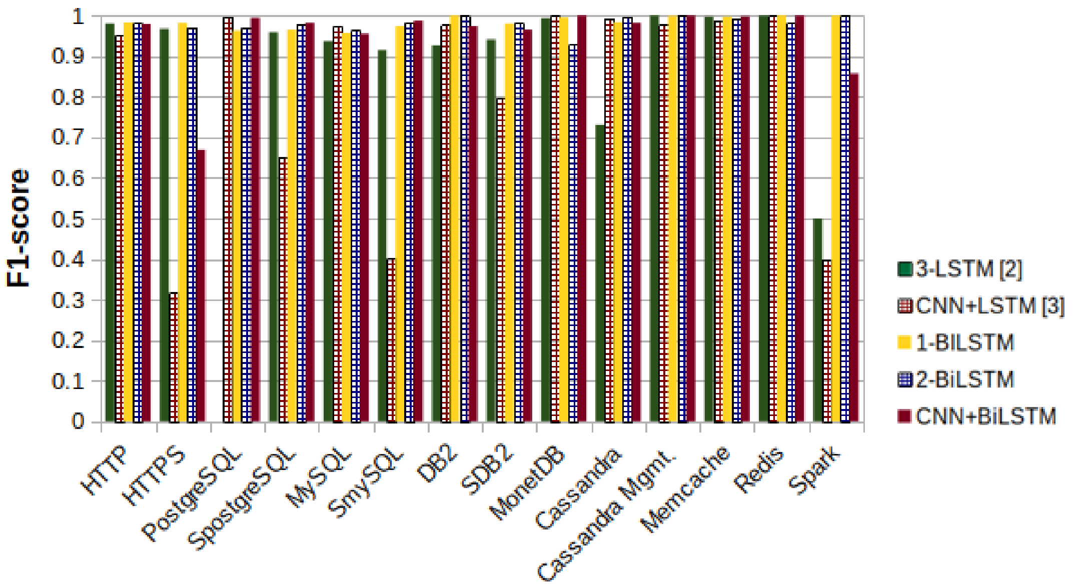


Figure 5.12: F1-score for individual service types using Δ_1.

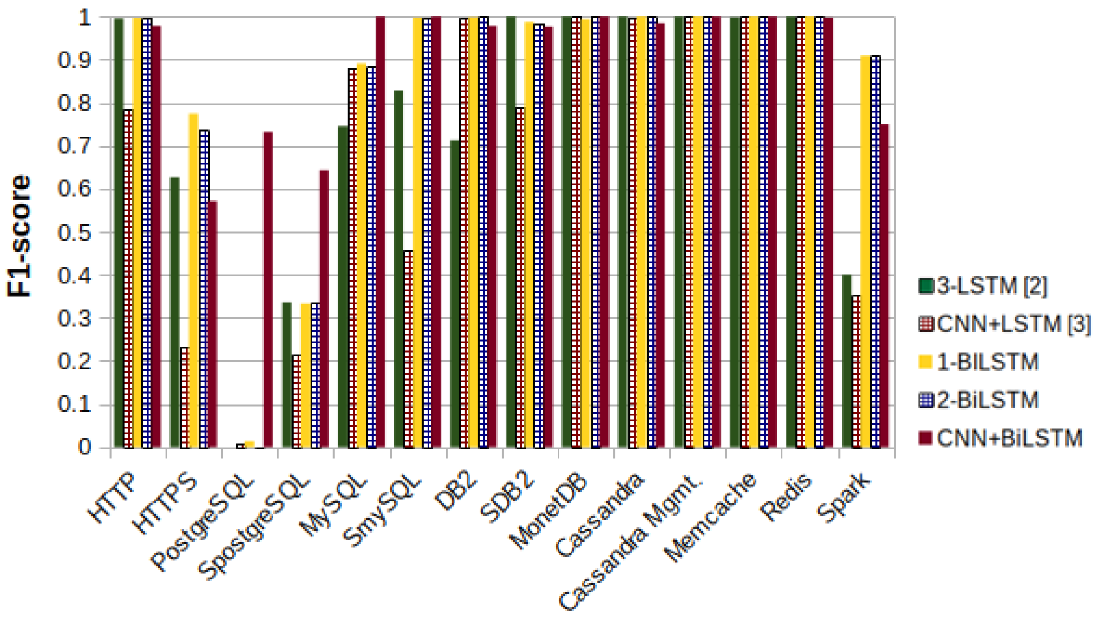


Figure 5.13: F1-score for individual service types using Δ_2.

remains quite similar to their performance for the Δ_1 dataset for most service types. Spark's F1-score are not quite as high and are, in fact, around 10% lower than when handshake packets are available. In other words, a fair amount of Spark traces are not identified as such. One reason could be the insufficient training data. The performance appears also worse for PostgreSQL and SPostgreSQL. All models have a significant decrease in the classification performance for these two service types in terms of F1-score with the maximum identification performance achieved by CNN+BiLSTM with 73.2% and 64.3% for PostgreSQL and SPostgreSQL, respectively. In general, CNN+BiLSTM performs very well for most of the service labels, including secured services such as SMySQL and SDB2, with F1-score higher than 97%.

***- Misclassifications between database systems*:** Similarly to the Δ_1 dataset, some misclassification also happened between different database systems for the Δ_2 dataset by most of the DLMs. For example, around 20% of PostgreSQL flows are wrongly classified as MySQL by the 3-LSTM and 1-BiLSTM models and 67% of SMySQL flows are identified as SPostgreSQL by the CNN+LSTM model, while 44.5% of DB2 flows are misclassified as PostgreSQL ones by the 3-LSTM model.

***- Misclassifications between the secured and unsecured version of the same service type*:** A significant number of PostgreSQL's flows are now wrongly classified as SPostgreSQL. This leads to a low recall and F1-score for PostgreSQL and a low precision and F1-score for SPostgreSQL. However, classifying unencrypted PostgreSQL flows as encrypted ones might not be as bad as classifying them as a completely different service; CNN+BiLSTM still determines that the flow is for a PostgreSQL database. A similar situation occurs in the other DLMs. For instance, around 79% of PostgreSQL flows are misclassifed as SPostgreSQL and around 66% of SPostgreSQL are misclassified as PostgreSQL by the 3-LSTM, CNN+LSTM, 1-BiLSTM and 2-BiLSTM models. Similarly, HTTPS F1-score has also shown a significant decrease by all the models when excluding the handshake packets (including the TLS ones). For instance, the CNN+BiLSTM has about 10% decrease in the HTTPS F1-score by having around 44% of HTTPS traces wrongly classified as unencrypted HTTP. However, the CNN+BiLSTM still detects that the service is a web-service.

c) Summary We were quite surprised by the encouraging results given the difference in message content between encrypted and nonencrypted data. The misclassification is not bad and often happens within the "same kind" of the service type. Of course, we have tested only with four different encrypted services for a preliminary evaluation and it might well be that if we add more encrypted services, that similarities between different encrypted services will occur, leading to more misclassifications. In addition, using the same kind of applications and workloads for collecting the traces of both the secured and unsecured database systems might be the reason behind the DLMs capability to classify secured database flows quite well.

However, we used different datasets for the HTTPS flows and still the DLMs can identify them as HTTP service.

5.6 Summary

In this chapter, we have provided a comprehensive study of the use of DLMs in service type identification of network flows. We have compared the performance of LSTM and a combination of LSTM and CNN models, some of them already proposed in the literature for NTC and some proposed by us, while using header-based and payload-based data for training. We have highlighted the trade-offs and the impact of various parameters on the classification performance such as the flow direction and the position of extracted packets in the flow. For instance, the results show that for header-based approaches, having access to the first handshake packets in the flow stream and working on bidirectional flows that correlate the incoming and outgoing messages of the service type are both important for high service type identification accuracy. Both these things are not so important for payload-based approaches to work well. We have also conducted a preliminary evaluation of how good the DLMs can detect encrypted communication for the payload-based dataset. The results show that the DLMs can identify the encrypted services with good accuracy.

In general, for both dataset types, DLMs with at least one BiLSTM layer provide the best performance with reasonable overhead. For instance, 1-BiLSTM always yields better identification performance than three layers of regular LSTM, i.e. 3-LSTM, and CNN+BiLSTM always performs better than CNN+LSTM for both header- and payload-based datasets. The additional backward layer in the BiLSTM makes it more capable of capturing additional features associated with the learning data than a regular LSTM. For example, CNN+BiLSTM has 10% better performance than CNN+LSTM for the bidirectional header-based dataset while having less than 1% more training time as shown in Table 5.4 and Figure 5.6b. However, adding more BiLSTM layers significantly increases the training time for all datasets with a moderate increase in the DLM performance. For instance, 2-BiLSTM triples the training time required for the unidirectional variant of payload-based dataset while enhancing the identification performance by only one percentage compared to 1-BiLSTM (see Table 5.4 and Figure 5.11b). For the payload-based dataset containing some encrypted network flows, having more than one BiLSTM layer does not improve at all the classification performance. Thus, it is not clear that adding more BiLSTM layers is beneficial.

In addition, DLMs with CNN layers perform as well as or even better than the ones without CNN layers for most of the evaluated header- and payload- based datasets. The kernel in the CNN layer can perfectly locate the similarity between the constructed images of the network flows that belong to the same service type.

Overall, CNN+BiLSTM yields excellent classification performance and boosts the service type identification performance in terms of F1-score by at least 5% for both encrypted and unencrypted flows and up to 10% for unencrypted flows compared to the models proposed in the literature with reasonable training time overhead. It is able to support a wide variety of applications, handles well encrypted network flows and does not require handshake messages.

In many payload based datasets, should data be mislabeled, the wrong label often belongs to a "similar" service type, e.g., another caching system or another database system, or even to the (un)secured version of the same service type. In contrast, the misclassification in the header-based datasets happen across many service types with no clear preferences for services that are of the same kind.

As such, we believe that for the service type identification that we are looking for in our MaaS using payload-based data is crucial as monitoring can start at any time and access to the handshake messages is unlikely. Furthermore, CNN+BiLSTM appears to be the most promising approach as it handles all kinds of datasets very well.

6 Dynamic Application Call Graph Formation and Service Identification Platform

In Chapter 1 we outlined three important features that a monitoring framework for distributed applications should provide: (1) a real time analysis of the call graph that shows how the different components call each other, (2) the service types each of the component offers, (3) and a wide range of performance measurements for each of the individual components and in regard to each other. In Chapter 4 we have shown a network-based MaaS that can provide performance measurements for the individual components using monitoring agents that sniff messages at the network switches. In this chapter, we show how we can extend this sniffer-based approach to provide the remaining functionality.

Recall that this thesis is aimed at monitoring service-oriented architectures, where the execution of an external client request leads to calls to various components, each of them providing a different service, resulting in a complex call graph. Knowing the structure of running applications and understanding how the different components call each other can help cloud providers in optimizing their resources and application administrators in monitoring the health of their applications at run-time [HLZ$^+$]. We believe that the cloud infrastructure should be able to infer the overall structure and dependencies between distributed components of the cloud applications, and the service type of each component without the need for any application-specific support, in order to serve an arbitrary number of different services and platforms.

Therefore, in this chapter, we first outline a generic application call graph and service type identification framework, referred to as *DyMonD*, that Dynamically Monitors an application, Discovers the service components, and visualizes application component call graphs, providing a global view of an application in the cloud at run-time. DyMonD also provides some application performance metrics such as the throughput and response time in order to show how it relates to the MaaS we presented in Chapter 4. DyMonD has basically the same architecture as the MaaS of Chapter 4, while extending the functionality of the monitoring agent and controller components to provide dynamic service identification and the visualization of the application components' call graph. Then, we describe how the DLM that we proposed in Chapter 5, namely CNN+BiLSTM, is adapted for DyMonD to classify the service type of each component. In addition, we show an additional module of DyMonD that uses natural language processing (NLP) to perform a deep-packet inspection, if possible, to determine more application-specific service types (e.g., "authentication" or "recommender") for microservice-based architectures. Finally, we provide a comprehensive evaluation for DyMonD in terms of its accuracy in detecting the correct call graph and service types as well as its overhead on the monitored application. We also present representative use cases that show DyMonD's usefulness.

6.1 DyMonD Overview

DyMonD is a network-based monitoring framework that observes network messages to provide information about how an application is executing at run-time. As an input, DyMonD requires the IP address of any component of the application as an entry point, e.g., the web-server to which external clients connect, a caching server or even the IP address of an external client. Then, DyMonD determines the components with which this entry component communicates, and iteratively, the components communicating with these components, in order to build the application call graph. For each node (component) in the call graph, DyMonD attempts to determine the service type, i.e. HTTP-based server, MySQL server, etc., and captures some performance metrics.

In this section we illustrate the capabilities of DyMonD using several distributed applications, and then show the overall architectural design of DyMonD.

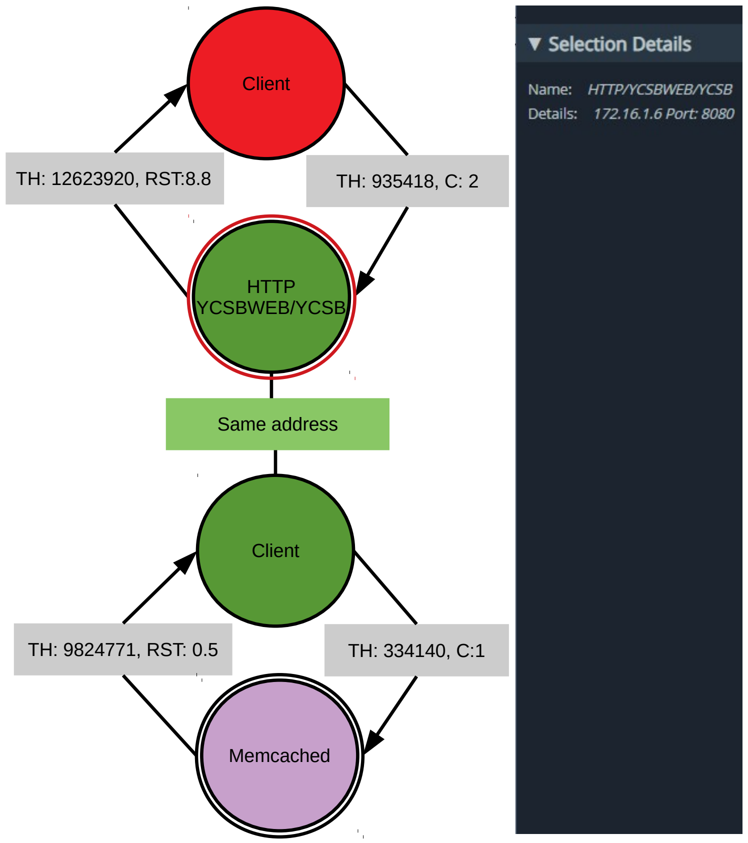


Figure 6.1: Call graph for three-tier web application.

6.1.1 Sample Application Call Graphs

Here we show the output that DyMonD provides for three different distributed applications and illustrate the iterative process used to determine the call graphs.

Simple three-tier web application We use here the extended version of YCSB benchmark we described in Section 3.3, where a web-server makes calls to a MySQL database server and a Memcached server. We use a workload of HTTP requests, where the web server first checks whether the requested data is in the Memcached, and only if it is not there, the web-server retrieves it from the MySQL database. The database schema and the query requests follow the YCSB benchmark.

Figure 6.1 shows the call graph that DyMonD generates for this application. DyMonD uses the visualization tool WebVOWL [WEB] with some minor changes to the source code as we will discuss in more detail in Section 6.4. The call graph is represented as directed graph where the colored nodes in the call graph represent the application components and the communication paths are shown as arrows, i.e. edges, between two nodes. Nodes with the same color indicate the same IP address and consequently refer to the same component. Each node is either labeled with the service type or as a client to another service. That is, the component represented by the green nodes is a HTTP server as well as a client to Memcached. The arrows between nodes of different colors show the average message throughput (TH) in byte/sec over the monitoring period. Additionally, an arrow from a server to a client indicates the average response time (RT) in microseconds, while an arrow from a client to a server indicates the number of open connections (C) to the service. As not all information might be nicely visible within the graph itself, users can click on individual nodes and edges and will see more detailed information in a side window.

In Figure 6.1, the database server is not included in the call graph because we started monitoring after the application was already running for a while. Therefore, all query results were already cached in Memcached and no further accesses to the database were needed during the monitoring period.

TeaStore, a microservice application The TeaStore benchmark [vKES+19] is an online storefront application that follows the microservice architecture where the different components use the REST API. TeaStore contains six microservices based on HTTP: A Web UI, an image provider, authentication, a recommender service, persistence, and a registry. Furthermore, it has a MySQL database system. All microservices as well as the MySQL server are deployed as separate containers and the *HttpLoadGenerator* [vK] is used to generate user requests.

Figure 6.2 shows the call graph produced by DyMonD. As mentioned before, in microservice-based architectures most components use the HTTP protocol and by default, are recognized as a HTTP service. Upon request, DyMonD can perform a deeper analysis (through natural language processing) and suggest more specific service types which we have enabled in this example.

To understand the process that DyMonD uses, assume that the IP address of the Web UI service was given as a starting point. Thus, during the first iteration, DyMonD detects that the given IP (depicted as brown node) is a web-server for a client host (red), and a client to 5 other web-servers. It suggests the specific service type to be "WebUI/...". The other web-services correspond to the image provider, recommender, authentication, persistence, and the registry services. In the second iteration, DyMonD considers the called services and finds what they themselves call. For example, the persistence service (pink) is identified

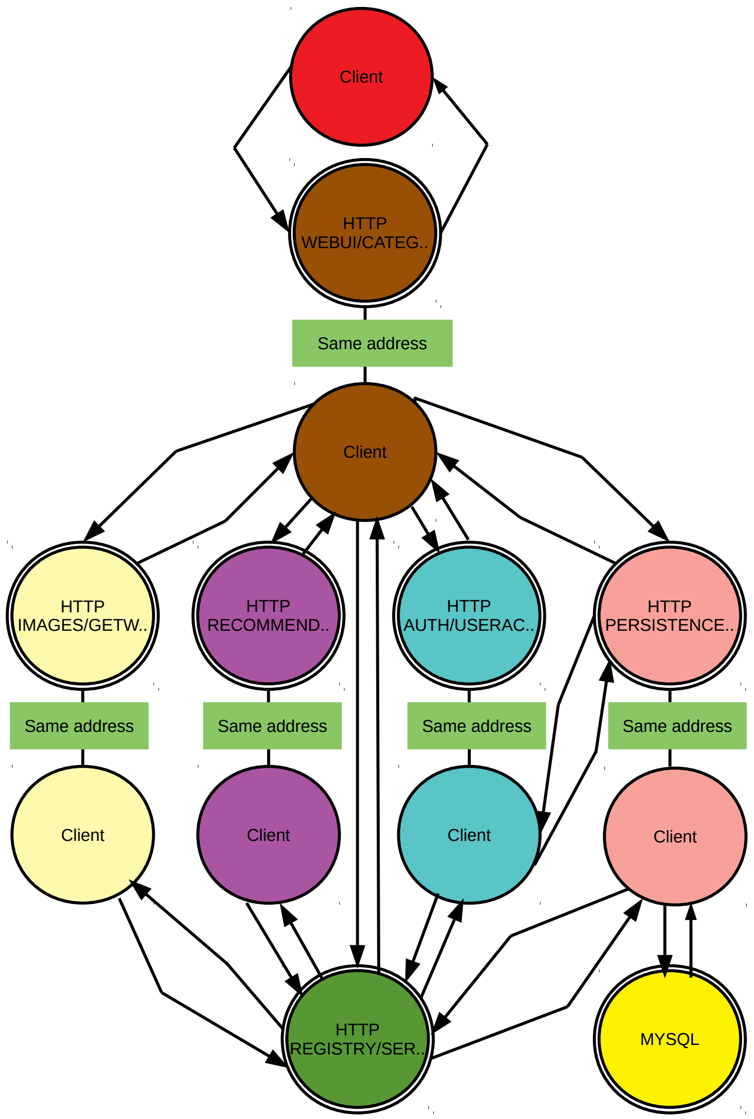


Figure 6.2: Call graph for TeaStore (Edge information is hidden for better readability).

as a client for the MySQL service (yellowish) and the registry service (green), while the authentication service (blue) is identified as a client for the persistence and registry services. Furthermore, the image provider (beige) and recommender (dark purple) are both clients of the registry service. The registry service is not a client to other entities.

SockShop, a further microservice application The SockShop [Weab] is another example of a microservices-based application that simulates an e-commerce website that

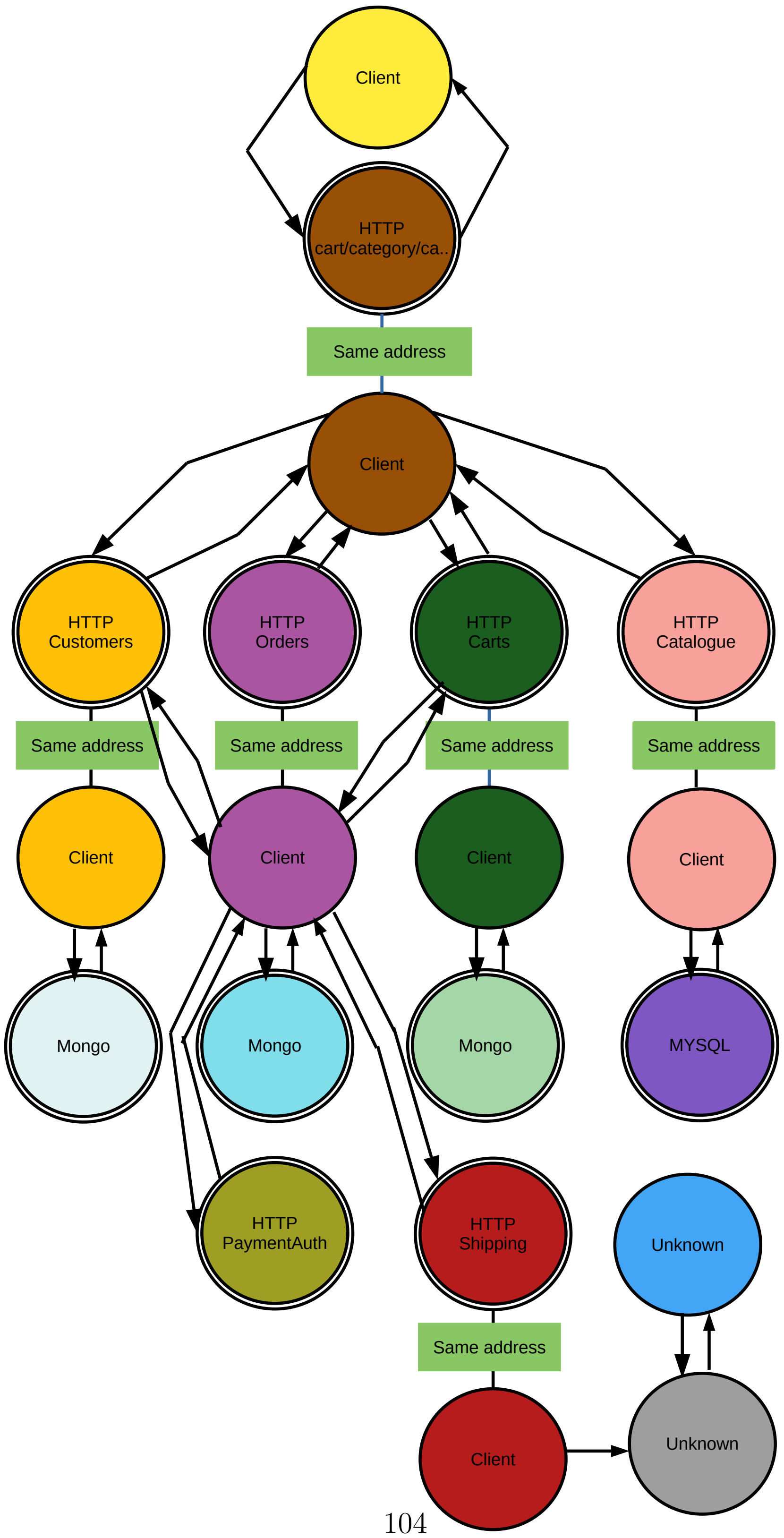


Figure 6.3: Call graph for SockShop (Edge information is hidden for better readability).

sells socks. Sockshop has 12 microservices: 7 HTTP-based services (front-end, catalogue, orders, payment, users, carts and shipping), 4 database systems and one queuing system. Each microservice is individually containerized and a Locust [HBHH] based load test script is used to simulate user traffic to Sock Shop.

Figure 6.3 shows the call graph produced by DyMonD where the IP address of the front-end service is given as the starting point. At first iteration, DyMonD detects that the given IP (depicted as brown circle) is a web-server to one client host (yellow), and a client to 4 web-services: customers, orders, carts and catalogue services. At the second iteration, DyMonD displays the services called by these 4 services. For example, the orders service is identified as a client to other 5 services: customers (light orange), carts (green), shipping (red), payment authentication and Mongo DB (light blue), while the catalogue, carts and customers services are all identified as clients to one DB service. At the third iteration, DyMonD detects that the shipping service (red) is calling another service (the grey one). As the service type is not recognized by DyMonD, the "Unknown" label is displayed. At the fourth iteration, the grey service is detected to have a communication with one component with un-recognized service type (blue). The two un-identified services are a queue and a queue master service to handle the processing of the shipping orders. The queue master service itself is not a client to any other services. Note that if DyMonD cannot identify a service, it is marked as "Unknown". This may happen if the underlying communication protocol was not part of the training set of DyMonD's DLM or the service type prediction score does not reach a preset threshold as we will describe in Section 6.5.

6.1.2 Design

Figure 6.4 illustrates the overall DyMonD architecture and its deployment inside the cloud network. DyMonD has the same architecture as the MaaS prototype described in Section 4.1. However, the functionality of the agent and controller have been extended to achieve DyMonD's functionality. As shown in Figure 6.4, a DyMonD agent is attached to each software switch in the cloud network and is responsible for capturing the messages of flows that are relevant for a particular call graph, followed by collecting messages from these flows in order to detect service types and measure performance. The DyMonD controller orchestrates the discovery process and controls the search for the application call graph. It finds the component locations with the help of DyMonD agents, asks the agents to monitor certain flows, collects and aggregates the results and outputs the call graph information. The visualization frontend receives the monitoring input information from the user such as the IP of the starting component and monitoring duration and renders the call graph information for visualization. In the following sections we explain each of the components in detail.

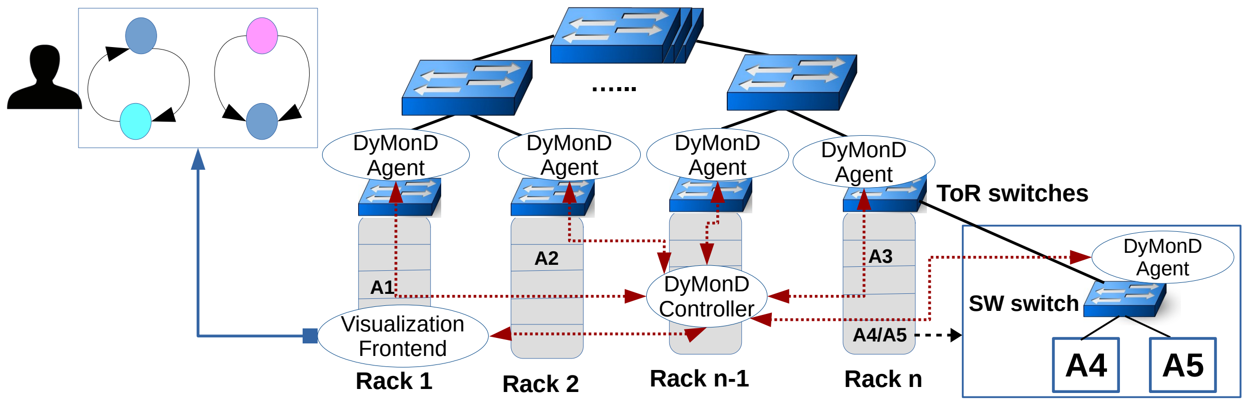


Figure 6.4: DyMonD architecture overview

We envision DyMonD to be used in two operation modes. In the *offline mode*, DyMonD monitors the traffic of the targeted application components for a certain predefined duration after which it produces the application call graph. In contrast, the *online mode* keeps monitoring the encountered application components and updates the detected call graph periodically. This includes adding any new components, and updating performance metrics as detected in the last time interval.

Assumptions Just as for the MaaS prototype presented in Chapter 4, we assume that each component can be identified by a unique IP and that each message flow goes through at least one software switch as this allows us to observe message flows with the sniffer approach presented in Chapter 3. Moreover, each component under consideration can be a client to several services, server for several clients, or both. We assume that each service is uniquely identified by both the IP address of the component where the service is running and the service port, while the client components are identified only by the IP address. In other words, each component can have at most one service port and/or one or more open client ports to other service(s). For instance, assume that A2 in Figure 6.4 is a web-server that is also a client to A3, which is a Memcached server. In this case, A2 should have only one service port opened to receive web requests, while one or more ports can be opened to the Memcached service deployed at A3.

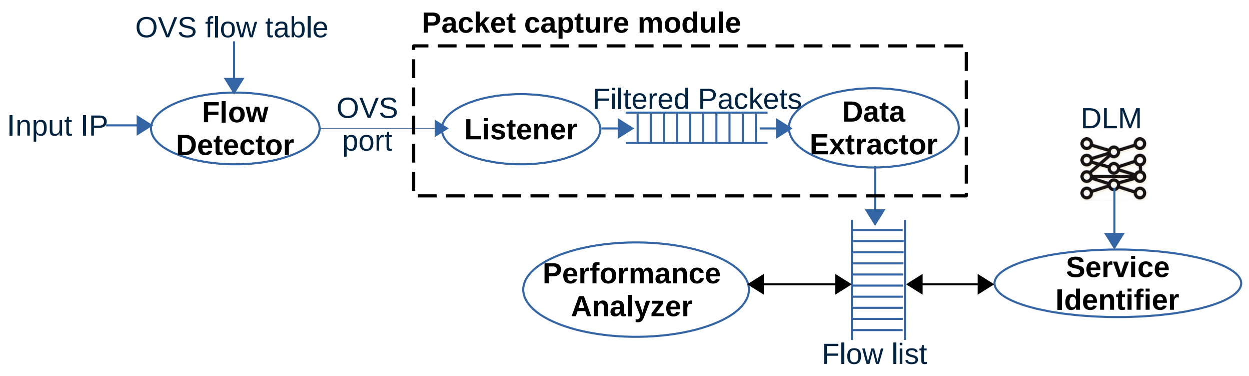


Figure 6.5: DyMonD agent architecture includes flow detector, packet capture, service identifier, and performance analyzer

6.2 DyMonD Agent

The DyMonD agent is basically an extension of the MaaS monitoring agent that was presented in Section 4.1. DyMonD agents are in charge of detecting flows and their service types, as well as measuring performance. An agent is deployed on the host of the software switch it is observing, in order to inspect incoming and outgoing messages, as shown in Figure 6.4. In this way, DyMonD conducts non-intrusive message monitoring at the software switch level. We describe the implementation of the DyMonD agent within OVS.

Figure 6.5 provides a high level overview of the DyMonD agent which consists of the following four modules: (i) the *flow detector module*, (ii) the *packet capture module*, (iii) the *service type detector model*, and (iv) the *performance analyzer module*. We can see that the DyMonD agent extends the sniffer design depicted in Figure 3.3. The packet capture and performance analyzer modules have been extended compared to the originally sniffer, while the flow detector and service identifier are new modules.

6.2.1 Flow Detector

The first task of the DyMonD agent is to detect all flows the switch handles that are related to a given IP, that is, where the component associated with the IP is either sender or receiver. This is how DyMonD finds all the components a given component communicates with, and from there iteratively the call graph. To detect the flows, the agent maps the IP of the component to the OVS port name (which is the basic input for the next packet capture module). As described in Section 2.1, when component C sets up a connection with another component that goes through a particular switch, the switch assigns one of its ports P to C. All messages to and from C go through P. The mapping between the IP address of C and P can be found in the OVS flow table. Thus, the flow detector accesses the OVS flow table to determine whether the switch it is attached to handles one or more of C's connections and if so, determines the associated port(s). For virtual switches implemented by network bridges on the host, the DyMonD agent runs a script to grab the virtual network interface associated with the IP from the host's IP routing table. The flow detector then passes it to the packet capture module. Note that in Sections 3.2 and 4.1 the focus was on the capability of our switch-based monitoring approach to deduce several application performance metrics, and thus, we assumed that the switch port to sniff is already known. However, the flow detector module could also be integrated into MaaS prototype agent to find the switch port that is connected to the application component if it is unknown to the MaaS service.

6.2.2 Packet Capture

This module sniffs all messages of the flows associated with a given port/network interface similar in concept to what was presented in Section 3.2. This time, the agent sniffs until a predefined number of packets have been captured or a predefined monitoring duration has passed. The number of packets should be sufficient to reliably determine the service type as required by the DLM (i.e., the 100 data packets required by the payload-based CNN+BiLSTM model) as well as to deduce the performance measurements.

The packet capture module itself inherits the multi-threaded architecture of the sniffer that is presented in Section 3.2 and runs separately from service detection and performance analysis modules to avoid interference. This is needed in order to work at the OVS port speed. The packet capture module has two main threads; the *listener* and the *data extractor* as depicted in Figure 6.5. The listener thread keeps sniffing on the given switch ports and saves the captured packets into a shared memory space. From there, the data extractor thread parses the packets, and extracts the needed information to construct the communication flows. The data extractor thread extracts information from each filtered packet as a 6-tuple: timestamp, source IP, source port, destination IP, destination port, transport protocol (i.e. TCP/UDP). If the packet has payload data, a portion of the payload is also stored in form of a flow record; such a record will be used by the service identifier as we explain below. The packets are then classified into distinct unidirectional flows, one for outgoing and one for incoming messages. For instance, in Figure 6.4, if component A1 makes requests to component A2, then a flow from A1 to A2 contains the requests, and a flow from A2 to A1 transmits the responses. Each unidirectional flow is uniquely identified by the source IP/port pair, destination IP/port pair and transport protocol (i.e. TCP/UDP).

Furthermore, the data extractor thread keeps track of aggregated traffic volume sent on this flow. This kind of information will be analyzed later for performance measurements. All this information is compiled in the "flow list" shared memory space as shown in Figure 6.5.

6.2.3 Service Identifier

The service identifier is implemented as a separate parallel thread that analyzes packet entries for each constructed flow in the "flow list" shared memory space. In particular, for each flow, the service identifier module performs the following tasks:

(1) Service type identification: The service identifier module determines the service type of each flow, e.g. MySQL, HTTP, etc. It first tries to recognize the flow's service type by standard ports (e.g., "if it's on port 80, it must be HTTP"). We keep a database with the most common services and their standard ports. If the port is not typically reserved for these well-known services, the service identifier feeds the stored flow's packet payloads to a pre-trained DLM, and the flow is labeled with the service type the DLM predicts. The learning model also indicates a prediction score which represents the confidence level of the prediction. The service detection module can be configured with a threshold T, in which case the flow is labeled as the "Unknown" service type if the prediction score is less than T.

(2) Server/client classification: To build a proper application call graph similar to the ones illustrated in Figure 6.1 - 6.3, DyMonD has to know for each flow, which end point is the server and which is the client. In case of the usage of the standard port, the position of the service port in the flow identifies the server entity. For instance, if a flow contains port 8080 as a source port, that means that the source IP is the server component for that flow. On the other hand, it is more challenging to differentiate between the server and client entities of the flows that do not use standard ports. Towards this end, we have extended the payload-based CNN+BiLSTM DLM presented in Chapter 5 to not only predict the service type of a unidirectional flow but also to predict weather a flow goes from a client to a server, referred to as *client flow* or from a server to a client, referred to as *server flow*. We will describe the details in Section 6.5.

(3) Fine-grained service type classification: Furthermore, the service identifier module optionally runs an NLP-based detection module to identify the application-specific service type for web-based services. The details of that NLP approach are given in Section 6.5.

6.2.4 Performance Analyzer

At the end of the monitoring duration, the performance analyzer uses the data gathered by the packet capture module as input to produce relevant performance metrics for each detected service. The performance analyzer provides a subset of the performance metrics listed in Table 4.1. In particular, DyMonD currently tracks three performance metrics for all service types, namely throughput, connection rate, and service time. For the latter, we follow the approach of observing time intervals between the server's ACK messages to estimate the server's response time as previously described in Section 3.1.2. As this approach does not require a deep packet inspection, it can be applied for services with and without secured payloads. If the message data is accessible to DyMonD, an alternative way to measure the service time is through matching requests and responses pairs as also described in Section 3.1.2. However, recognizing the requests/responses of each service requires the knowledge about the structure of the service type's communication pattern. We have already implemented the average service time using that approach of constructing request/response pairs for some common service types such as HTTP, Memcached and MySQL as described in Chapter 4. Defining the request/response message structure of other service protocols is possible through the API that allows for dynamic integration of code snippets as described in Section 4.3. Moreover, further performance metrics can also be added through the same API.

6.3 DyMonD Controller

The DyMonD controller is the broker between a user and the DyMonD agents. We first discuss the controller tasks assuming a single OVS switch connects all components, i.e., only one agent. Then, we describe how the controller handles multiple agents deployed on OVS switches across the cloud network.

6.3.1 Controller with a Single Agent Configuration

Figure 6.6 provides a high level overview of the DyMonD controller and Algorithm 1 outlines the iterative process that finds relevant information in a breadth first search. It consists of three main steps: (i) iteratively collecting all flows that will determine the call graph (lines 2-11), (ii) clean the flows from any inconsistencies (line 12), (iii) and then create the actual call graph data (line 13 with details in Algorithm 2).

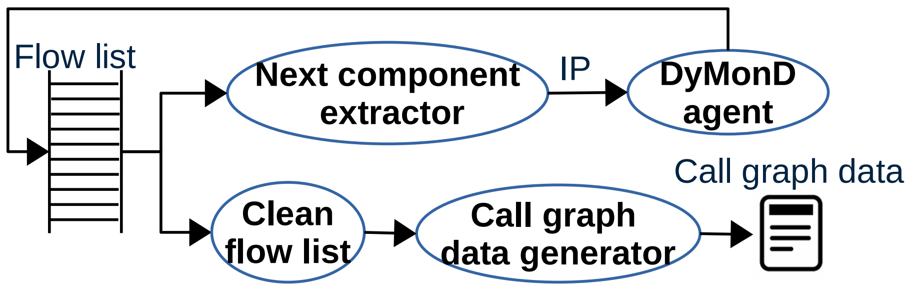


Figure 6.6: DyMonD controller architecture.

Collect the flows The DyMonD controller receives the IP of a start component from the user. It represents the entry point for the call graph. The controller adds the received entry IP to a queue (line 1) and forwards this entry IP to the DyMonD agent for analysis (lines 2-4). The agent performs the analysis for the received IP as described in Section 6.2. For each detected flow that has the IP as a source or destination, the agent sends the flow ID together with the detected service type, flow direction (i.e. client or server flow), prediction score and performance measurements back to the controller. The controller then consolidates the received flow information to one flow list (line 5). From here the controller executes the *next component extractor* to get further flows and with it further components (lines 6-11).

The next component extractor analyzes the flows it just received to infer the next components, i.e., it extracts the IP addresses with which the current component interacts and appends them into the queue. For the IP addresses that have not yet been previously considered, the steps are repeated and the agent is asked to collect the corresponding flows. The steps are repeated until no further components are detected. Alternatively, we can also indicate a depth d of the call graph, and when it is reached the analysis stops, even if further components are detected. Thus, the DyMonD controller follows a breadth first search with a depth of d levels. We note that a flow might be detected twice (once for the source IP and once for the destination IP). The controller will note that and only keep one record (line 5). Note that the information for the flow might be different in the two instances, e.g. different performance measurements and also different service labels. In regards to performance measurements, the controller will keep the latest values. In regard to the service labels, it will perform a cleaning as discussed next.

Clean the flows Once all flows have been detected, a "clean" function detects any inconsistent service labels in the flows and consolidates them. First, there could be different flows referring to a particular IP as a service, but the service types identified are different. For instance, some flows could label a component as a DB2 database service while others label it as MySQL. Even for the same flow as it is detected twice (once when searching for

Algorithm 1: Pseudo code of the DyMonD Controller

```
input  : Start $IPstart$
output : Call graph $G$ of $IPstart$
Data   : Empty queue $Q$, empty lists $ComponentFlows$ and $AllFlows$, empty sets
         $IPnext$ and $IPvisited$
1  $Q.enqueue$(IPstart);
2  while $Q$ is not empty do
3      $CurrentIP$=$Q.dequeue()$;
4      $ComponentFlows \leftarrow$ AGENT($CurrentIP$);
5      $AllFlows \leftarrow (AllFlows \cup ComponentFlows)$;
6      $IPvisited \leftarrow (IPvisited \cup CurrentIP)$;
7      $IPnext \leftarrow$ NEXT COMPONENT EXTRACTOR ($ComponentFlows$, $CurrentIP$);
8      for $IP$ in $IPnext \setminus IPvisited$ do
9          $Q.enqueue(IP)$;
10     end
11 end
12 $AllFlows \leftarrow$ CLEAN FLOWS ($AllFlows$);
13 $G \leftarrow$ CALL GRAPH PRODUCER ($AllFlows$);
14 return $G$;
```

flows of the client, and one when searching for flows for the server) there might be a mismatch. Finally, some flows might be wrongly labeled as a client resp. server flow leading to confusion of who is client and who is the server. Our clean function unifies the service labels. In particular, for each detected service *SIP/port* identified by its IP/port number, the clean function checks whether all the client and server flows of *SIP/port* have been labeled with the same service type. And if not, the clean function unifies the service type label by choosing the service type that has the highest prediction score, or the highest frequency among all the *SIP/port* flows. The latter means that if for instance a total of 10 client and server flows are detected for *SIP/port* and seven of these flows are labeled as "DB2", two flows are labeled as "MySQL" and one flow is labeled as "Unknown", then the clean function changes the service types of the "MySQL" and "Unknown" flows of *SIP/port* to be of "DB2" service type. Furthermore, in this cleaning process the controller validates the client/server labeling in each pair by checking the position of the service in the flow. That is, a flow with a service *SIP/port* as a source end-point will have the "server" label and a flow with a service *SIP/port* as a destination end-point will have the "client" label. At the end of the cleaning process, for each IP/port pair identified as a service, the flows involving this pair have the same service label, and all flows that have this pair as destination (resp. source) are labeled as client flows (resp. server flows).

Call graph creation The final step of the controller is to produce the call graph (line 13). The details if the graph producing algorithm are depicted in Algorithm 2. As described in Section 6.1, the call graph is a colored directed graph $G(N, E)$ that contains N as the set of colored vertices and E as the set of edges, where the colored vertices represent the application components identified by their IP address and the edges are the detected communication flows between them identified by the IP address pair of the two communicating nodes. Each vertex is labeled as a client or a server. The server component can have many connected clients and might be itself a client to other servers. Therefore, each application component can be represented by a maximum of two vertices in the call graph with same color, one labeled as a server and the other labeled as a client. For instance, there are two brown circles for the WebUi service of the TeaStore benchmark in Figure 6.2, one being a web-service and the other one being a client to four other services. The edges are labeled with the performance metrics as discussed in Section 6.1.1.

To create the call graph the controller iterates over the list of flows (lines 2-13). It extracts the IP addresses of the client and server end-points of each flow and creates the corresponding client (line 6 and 11) and server nodes (line 7 and 10). The process in both cases is similar. If no node is previously created with the extracted client resp. server IP, a new node is created with a unique color and labeled as client resp. server (lines 17-20 and 32-35). If a client node $Nclient$ is to be created and there is a server node $Nserver$ with the same IP as $Nclient$, $Nclient$ is created with the same color of $Nserver$ and an edge of "same address" is added between $Nclient$ and $Nserver$ (lines 21-26). The same happens for server nodes. That is, if a client node $Nclient$ already exists with the same IP address of server node $Nserver$ to be created, $Nserver$ is created with the same color of $Nclient$ and a "same address" edge is added between them (lines 36-41). If a client resp. server node already exists, nothing has to be done.

From there, for each client resp. server flow, the controller creates a new edge between the two nodes of that flow and labels it with the performance metrics collected for that flow (line 8 and 12). If the edge already exists, the controller only updates the edge labels according to the performance metrics of the current flow. Recall that the edges are identified by the IP addresses of the two communicating nodes. Therefore, all the client resp. server flows between two communicating components are represented by one client resp. one server edge. For the client edges the number of the client flows originating from a client to a server node are represented by the edge's "C" label. For instance, the two client flows detected between the external client of the YCSB web-server in Figure 6.1 are depicted by one edge that goes from the client node (red circle) to the web-server node (green circle) while C is 2 to indicate the two detected client flows. The same also applies to the edges between the different components that apply multi-threading or connection pooling for their communication. For instance, six client flows are detected between the SockShop front-end and customer web-

Algorithm 2: Pseudo code for call graph producer

```
input   : List of flows $AllFlows$
output  : Call graph G
Data    : Empty list of Nodes $N$ and Edges $E$
Function Main($AllFlows$):
    for $Flow$ in $AllFlows$ do
        $IPsrc \leftarrow$ Source IP of Flow;
        $IPdst \leftarrow$ Destination IP of Flow;
        if $Flow$ is a client flow then
            $Nclient \leftarrow$ Client Node ($IPsrc$);
            $Nserver \leftarrow$ Server Node ($IPdst$);
            $E \leftarrow$ Create/Update Client Edge($Nclient$, $Nserver$, $Flow$);
        else if (flow is a server flow) then
            $Nserver \leftarrow$ Server Node ($IPsrc$);
            $Nclient \leftarrow$ Client Node ($IPdst$);
            $E \leftarrow$ Create/Update Server Edge($Nserver$, $Nclient$, $Flow$);
    end
    $G \leftarrow$ Graph($N$, $E$);
    return $G$;
Function Client Node($IP$):
    if Node with $IP$ not in $N$ then
        $C \leftarrow$ Choose Unique Color();
        $Nclient \leftarrow$ Create Client Node($IP$, $C$);
        $N \leftarrow (N \cup Nclient)$;
    else if (A node $Nserver$ with $IP$ in $N$ of type server && $Nclient$ with $IP$ of type client not in $N$) then
        $C \leftarrow$ Get Color of($Nserver$);
        $Nclient \leftarrow$ Create client Node($IP$, $C$);
        $N \leftarrow (N \cup Nclient)$;
        $Edge \leftarrow$ Create Edge($Nclient$, $Nserver$, "Same address");
        $E \leftarrow (E \cup Edge)$;
    else
        $Nclient \leftarrow$ Get Client Node($IP$, $N$);
    end
    return $Nclient$;
Function Server node($IP$):
    if Node with $IP$ not in $N$ then
        $C \leftarrow$ Choose Unique Color();
        $Nserver \leftarrow$ Create Server Node($IP$, $C$);
        $N \leftarrow (N \cup Nserver)$;
    else if (A node $Nclient$ with $IP$ in $N$ of type client && $Nserver$ with $IP$ of type server not in $N$) then
        $C \leftarrow$ Get Color of($Nclient$);
        $Nserver \leftarrow$ Create Server Node($IP$, $C$);
        $N \leftarrow (N \cup Nserver)$;
        $Edge \leftarrow$ Create Edge($Nserver$, $Nclient$, "Same address");
        $E \leftarrow (E \cup Edge)$;
    else
        $Nserver \leftarrow$ Get Server Node($IP$, $N$);
    end
    return $Nserver$;
```

service (Figure 6.3). Similarly, all these six client flows are represented by one edge from the client node of the front-end component (brown circle) and the customers web-service (orange circle) with C label of 6 (not shown in Figure 6.3). For the server edges, the "RST" label represents an average of the service time of all server flows between a server and client node.

Finally, the constructed nodes and edges are consolidated into a graph data file G (line 14) that is sent to the visualisation frontend for visualization.

The complexity of the controller process depends on the number of different IP addresses, the number of flows that connect them and some configuration parameters such as the given monitoring duration for each detected application component. At every iteration the agent captures all flows for an IP address and repeats until enough messages are captured or a timeout of the monitoring duration occurs. The controller iterates over these flows to detect new IPs and generates the graph, which is a fast process. For instance, to form the call graph for the YCSB, TeaStore and SockShop applications in Figures 6.1-6.3, the controller needed 0.3, 0.5 and 0.8 seconds, respectively, to execute its functions (excluding the time taken by the agent). A more detailed evaluation of the complexity of the agent and controller analysis functions will be presented in Section 6.6.

6.3.2 DyMonD in a Multi-agent Settings

To cover a large cloud network, we propose the deployment of a DyMonD agent with each software switch, i.e., all top-of-rack software switches or software switches within larger end hosts that hold many containers or virtual machines. Looking back to Figure 6.4, each ToR-switch as well as the large host in rack 8 should hold a DyMonD agent. We are assuming that the DyMonD controller knows and is able to communicate with all deployed DyMonD agents. This can be through periodical heartbeats sent by the agents to the controller.

In addition to the steps described in the previous section, for each component C to be visualized in the graph, the controller has to first determine the switches that handle the flows of C. One option could be that DyMonD consults the control plane of the SDN technology to determine the switches that handle flows of a given IP [ESEFAM21]. When C starts sending packets to the network and no rule match is found for those first packets, the SDN switch forwards them to the SDN controller to request a new rule. That flow rule request message contains information such as the IP address of C (i.e. IP_C), the SDN switch identifier and switch port number that is connected to that host. This way, the SDN controller can identify the connected components to any SDN switches.

As an alternative, the DyMonD controller can multicast the IP address, IP_C, of C to all agents. The agents run the *flow detector* module as described in Section 6.2 and only the DyMonD agents whose switch handles flows involving IP_C will respond to the controller. Once the controller receives the confirmation message from the agent(s), it will command to the respondent agent(s) to start packet capturing and service detection as described in Section 6.2. Note that a connection might go through several switches. For example, in

Figure 6.4 when component $A1$ communicates with $A2$ the agents of ToR switches on both racks 1 and 2 will observe the message exchange. The DyMonD controller will detect the duplication and show the relevant information only once in the graph as described in the previous section.

As there may be hundreds of DyMonD agents in a cloud network, especially if it consists of many large machines that all have their own software switches installed, a hierarchy of DyMonD controllers can be provided for scalability. For example, an intermediate DyMonD controller can be deployed inside each rack. This will form a small cluster of DyMonD agents with one DyMonD controller as a cluster head, that will in turn communicate with one main DyMonD controller, reducing the number of nodes this main controller has to communicate with. Many different communication protocols can be used to route the traffic between the various controller levels in this hierarchy [ACA+15]. In addition, a fault tolerant mechanism, e.g., [SBS], can be employed to prevent the main controller from being a single point of failure.

6.3.3 Other Practicality Considerations

As mentioned in Section 6.1, we assume that each application component has one unique IP address as an identifier. In reality, application components may have multiple IP addresses. For example, application components that reside in a virtual private cloud, may have both public and private IP addresses. By default, DyMonD will show these two IPs as two different components in the call graph. Nonetheless, this issue can be resolved if DyMonD can have access to IP mapping information, as also assumed in other related work [J. 17, LTRW].

On the other hand, call graphs may span more than one application. For instance, two applications could have components that call a general purpose service, e.g., a spelling service, translation service, storage service, etc. If DyMonD starts with a component of one application, it will eventually reach the common service and from there it can reach the clients of the other applications. This can lead to privacy violations. Therefore, if a user wants to get the call graph of only their application, DyMonD should stop generating the call graph as soon as it reaches components that are no more in the application's domain. The cloud provider is typically aware of the owner domain of components as it is often the owner of these multi-tenant services. Thus, DyMonD would need access to such information.

6.4 Visualization Frontend

The visualization frontend is implemented as a web-server. Users connect via a web-browser to the frontend where they can indicate the IP address of the starting component and the monitoring duration the DyMonD agent should spend for capturing messages for each application component. The latter should be carefully configured for each application. That is, the applications with a low communication rate would need longer monitoring time to get enough packets for the DyMonD agent service identification module and build a complete call graph of the application.

Once the application call graph data is available for visualization, the frontend visualizes it. Currently DyMonD uses WebVOWL [WEB] as a visualization tool. As building a graph visualization tool is out of the scope of this thesis, we have experimented with several existing tools and found WebVOWL the most appropriate. WebVOWL has been designed for the visualization of ontologies but has served our very different purposes very well after some minor adjustments to the source code[1].

6.5 Dynamic Service Identification

Our investigation in Chapter 5 has shown that using the packet payload is more beneficial than using the information in the packet header in the context of the service type classification we are aiming at. Recall that the header-based protocols do work well if the service uses one of the standard ports, and if the handshake message exchange at the connection setup is captured. However, in dynamic cloud environments and when applications need to be monitored on demand, neither of the two options can be guaranteed. Furthermore, we have seen that for the graph creation we need to classify uni-directional flows as we have to also distinguish between clients and servers. But header-based approaches work worse with uni-directional flows. Thus, we decided to employ payload-based DLM in the service identification module of the DyMonD agent, namely the CNN+BiLSTM due to its superior performance compared to the other evaluated DLMs. The CNN+BiLSTM itself is trained offline and then deployed in the service detection module of the DyMonD agent.

In the following sections, we first describe how we have adapted the payload-based CNN+BiLSTM DLM described in Section 5.4 so that we do not only receive the service type label but also whether a flow is a client flow or a server flow. Then, we will describe

[1]Adjusted WEBVOWL source code: `https://github.com/qqqqyyy/webvowl1.1.7SE`.

the details of the NLP approach we used for fine-grained service type identification that DyMonD can optionally provide for microservice-based architectures.

6.5.1 Training CNN+BiLSTM DLM for DyMonD

As just mentioned, the client/server classification of each flow is crucial for DyMonD functionality to deduce a proper application call graph. Therefore, we have to use the unidirectional variant of the payload-based dataset described in Section 5.3 that separates the client and server flows into different records. However, in the original version of that unidirectional payload-based dataset, the client and server flows are labeled with the same label, e.g. "HTTP". Thus, in order to train the DLM for the purpose of DyMonD, we modified the labeling of the unidirectional payload-based dataset by assigning different labels to the client and server flows of each service type. For instance, the HTTP requests flows are labeled with "HTTP-C"; where "C" stands for client, while the HTTP response flows are labeled as "HTTP-S"; where "S" stands for service. We have applied the same labeling strategy to all the service types DyMonD currently supports including the secured services. This increases the number of service classes in the unidirectional payload-based dataset with secured traces from 14 to 26 (each service of those 14 ones, except Spark and Cassandra management flows, has two labels, one for the client flows and the other for the server flows). Note that for training we only use the dataset with middle flow messages as DyMonD has to support service identification at any point during the run-time of the application.

In summary, one record in the extended dataset is a sequence of 100 images representing consecutive messages extracted from a single unidirectional flow, and each image is of size 36 (6x6) corresponding to the first 36 bytes extracted from a packet's payload. Note that once the pre-trained CNN+BiLSTM is used in the DyMonD agent for service identification, it needs again as input flows with 100 packets. Thus, if the agent captures less than 100 packets during the pre-defined monitoring time duration, the flow needs to be discarded and cannot be considered.

6.5.2 Service Identification for HTTP-based Microservices

Since most microservices use HTTP, our basic DyMonD service identification will classify them all as HTTP. While this might be sufficient for cloud administrators, application administrators might want to have more information about the components. Therefore, here we describe how DyMonD can optionally provide a more fine-grained classification of

HTTP-based services.

Microservice standard is to use the REST API, where the service exposes its methods via URL resources. Thus, when a client makes a request, the URL string embedded in the HTTP message contains the name of the URL resource to be called. Often, these resources have meaningful names in the context of the application, that indicate what the service/method is actually doing, such as "persistence" or "authentication". However, a microservice might offer a whole range of methods as its API where each method can have also various input parameters. Thus, there might be many different URLs associated with a microservice, although they will likely all have some common substrings. For instance, examples of the URL strings found in the HTTP requests of the persistence service in Figure 6.2 are: "/tools.descartes.teastore.persistence/rest/users/name/user82", "/tools.descartes.teastore.persistence/rest/products/110", and "/tools.descartes.teastore.persistence/rest/orderitems". Thus, an analysis of such URL strings is needed to choose the resource name that is common between them as a fine-grained service label for the microservice component.

Algorithm 3 outlines this process. First, the agent maintains a list of URLs for each detected web-service. Then, a language processing tool [NLT] is used to return the most frequent REST resource names appearing in the URLs of each HTTP-based service. In particular, the DyMonD agent iterates over the flow list, extracts the URL strings from the HTTP request messages of the flows that are labeled as "HTTP-C" (lines 1-3), extracts the web-service identifier (IP/port number) (line 4) and builds up a list of web-services with their associated URLs (lines 5-8).

Then, the DyMonD agent iterates over the constructed web-service list, uses an NLP function to extracts the resources and calculates the use frequency of each appearing resource (lines 9-10). In particular, the DyMonD agent calculates the use frequency of each resource r as the number of times r occurs in URLs of packets sent from a client to the HTTP-based service divided by the total number of packets with URLs that the service receives.

$$Freq(r) = \frac{\text{Number of occurrences of r in URLs}}{\text{Total number of URL requests received}} \tag{1}$$

Note that there are some words that should be ignored when found in a URL string such as the extension name of a file (e.g. .css, .xml, .html, etc.) and the communication protocol identifier such as "http://". The resource frequency should reach a preset threshold TH to be selected for the web service type (line 11). We set TH to be 50%, which means the chosen resource should appear in at least half of the requested URLs to be chosen for the web-service application specific type. If the frequency of more than one resource reaches that

Algorithm 3: Pseudo code for fine-grained service type identification for HTTP-based services

```
input  : List of flows $FlowsList$ and Threshold $TH$
Data   : List of stop words $StopWords$, empty list of web-services
         $WebServicesList$ and empty resource-count pair list for resource count
         $ResourceCount$
1  for $Flow$ in $FlowsList$ do
2      if $Flow$ service label is "HTTP-C" then
3          $URLs \leftarrow$ Extract URLs($Flow$);
4          $WebService \leftarrow$ Extract Web-service($Flow$);
5          $WebService.URLs \leftarrow (WebService.URLs \cup URLs)$;
6          if $WebService$ not in $WebServicesList$ then
7              $WebServicesList \leftarrow (WebServicesList \cup WebService)$;
8  end
9  for $WS$ in $WebServicesList$ do
10     $WS.ResourceCount \leftarrow$ NLP ($WS.URLs$, $StopWords$);
11     $WS.Label \leftarrow$ Get web-service label ($WS.ResourceCount$, $TH$);
12 end
```

preset threshold, we output them all for the web-service label. If none of the resources reaches the threshold, we label the web-service with the most frequent three resources along with an "Unknown" tag to acknowledge the low frequency of the produced labels. The visualized graphs of the TeaStore and the SockShop shown in Figures 6.2 and 6.3 show the labels DyMonD has produced for the detected HTTP-based services.

The complexity of the fine-grained service type identification process depends on the number of detected HTTP client flows $F(HTTP{-}C)$ and number of HTTP request messages that contain the HTTP header where the URL is located. Given that the DyMonD agent collects N packets for each flow, and in the worst case all the collected N packets for those HTTP client flows have the HTTP header (e.g. for small HTTP request messages that span only one packet), the asymptotic run-time complexity of the fine-grained service type identification process is $\mathcal{O}(N * F(HTTP - C))$. In our evaluation, determining the label for a single HTTP-based service takes a maximum of 0.3, 13 and 6 milliseconds in YCSB, TeaStore and SockShop applications (Figures 6.1-6.3), respectively.

6.6 Evaluation

In this section, we evaluate the performance of DyMonD. We first analyze how well CNN+BiLSTM performs in terms of the prediction accuracy, precision, recall and F1 score now where the flows are not only labeled with the service type but also as client or server flows. We also analyze how good our fine-grained analysis for web-services works. Then, we evaluate the performance of DyMonD in terms of: (i) the accuracy of the formed call graph, (ii) the imposed overhead and (iii) the complexity of DyMonD analysis in terms of the time duration required by DyMonD modules to infer the application call graph. Finally, we present two use cases to show how DyMonD detects application performance bottlenecks and captures changes in the application's architectural patterns during run-time.

6.6.1 Service Identification Evaluation

In this section, we first evaluate how effective our CNN+BiLSTM model is in identifying certain service types as well as the direction of the flow as either client or server flow, using the extended unidirectional payload-based dataset described in Section 6.5.1. We first evaluate the training time of CNN+BiLSTM for the modified dataset and the overall classification performance. We then report the performance for individual services. Finally, we assess the quality of our fine-grained classification for microservices.

Model training and validation We have used the same validation environment as well as the validation and parameter tuning approaches described in Section 5.5.1 for training the CNN+BiLSTM using the extended dataset. In particular, we separate our flow-based dataset into learning and test datasets as indicated in Table 5.1. Further, 20% of the learning data is separated for validation, and the verification of the model is performed on the basis of the SMBO [TP12] hyper-parameter set listed in Table 5.2 and the value of the k-fold cross-validation [TGB18]. Then, the model is trained by using the optimal hyper-parameters and the model performance is evaluated using the test data. We note that SMBO has produced the same hyper-parameters values listed in Table 5.3 for the CNN+BiLSTM DLM using the extended dataset.

Training time We measured the average training time taken by the CNN+BiLSTM for the extended dataset for five runs. It takes 392 seconds on average to train the CNN+BiLSTM on the modified dataset with standard deviation below 0.1. This represents around 5% increase in the training time required by the CNN+BiLSTM model of the

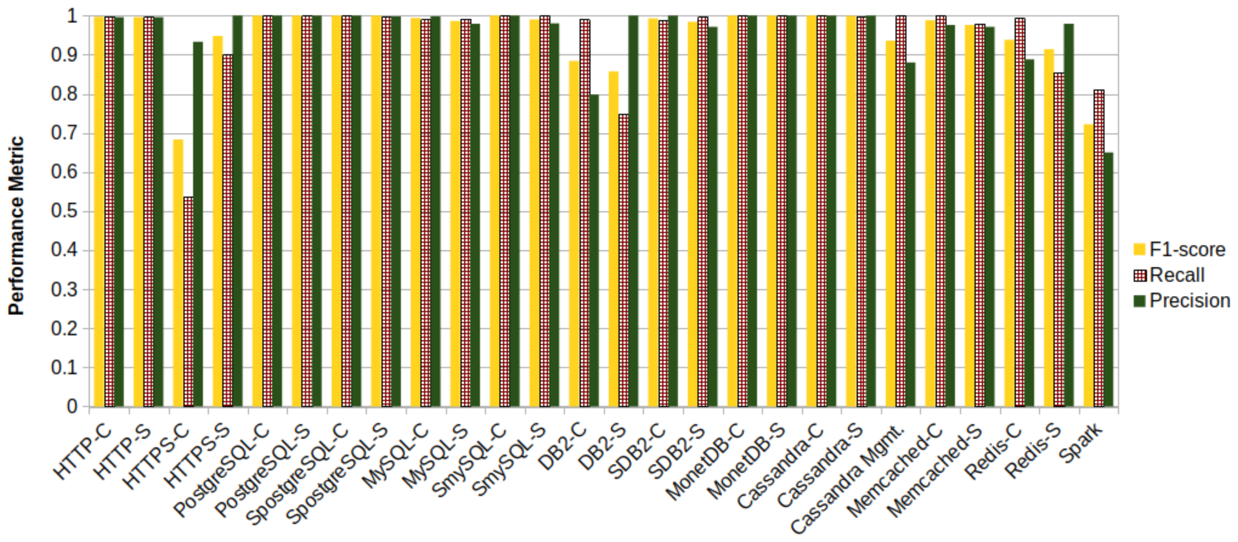


Figure 6.7: Middle-flow classification performance of the CNN+BiLSTM model for the client/server flows of encrypted and unencrypted service types in terms of Precision, Recall and F1-score.

original unidirectional payload-based dataset listed in Table 5.4. This increase is due to the increased number of classes in the extended dataset, almost the double of the number of classes in the original dataset. Note that the number of records and features have not been changed in the modified dataset, just the labels. We believe this is a reasonable overhead given the enhanced classification capability the CNN+BiLSTM now has by further distinguishing between the client and server flows for each service type. It is important to note that the DLM is trained offline as mentioned in Section 6.5, and hence, training has no impact on the execution time of DyMonD.

Classification performance We first evaluate the performance of DyMonD's DLM (i.e., CNN+BiLSTM model) in terms of the weighted average of accuracy, precision, recall and F1-Score metrics. Interestingly, DyMonD achieves a classification performance of around 89% for all the aforementioned performance metrics, which is almost the same performance as CNN+BiLSTM DLM for the original dataset. That is, increasing the number of labels, i.e. DLM classes, does not have a significant impact on the CNN+BiLSTM DLM performance.

Performance on a per-service basis We have a closer look at the performance of the CNN+BiLSTM DLM for the individual classes.

Figure 6.7 shows the F1-score, recall and precision results for all service types and flow direction. CNN+BiLSTM performs very well for most of the service labels, including distinguishing between the client and server flows of secured services such as SMySQL and SpostgreSQL, with F1-Score higher than 85%, except for the client flows of HTTPS and Spark, where the F1-score is below 80%. In other words, a fair amount of HTTPS-C and Spark traces are not identified as such. One reason could be the insufficient training data that is below 1% for each in the training dataset.

We observe that most of the misclassifications are between the server and client flows of the same service type. For instance, one third of the DB2-S traces are misclassified as DB2-C, 12.5% of Redis-S traces went to Redis-C and 2% of HTTPS-S traces are wrongly classified as HTTP-C and HTTPS-C. This kind of misclassifications can be corrected by the clean function outlined in Section 6.3 as soon as the service type is identified consistently and the position of the service in the communication flow, i.e. source or destination, is used to correct the server and client labeling.

There are also some misclassifications between the cache systems. For instance, about 12.5% of the Redis-S traces are misclassifed as Memcached-S. As we have mentioned before misclassification between different caching systems might be more acceptable in the context of DyMonD functionality than misclassifying it as something completely different, for instance, Redis as HTTP-service.

Identification for HTTP-based microservices To validate the ability of DyMonD for fine-grained classification of HTTP-based microservices, we compare the "application-specific" service labels automatically created by DyMonD with the actual ones described in the evaluated benchmark documentation. Table 6.1 shows the service labels predicted by DyMonD, as described in Section 6.5.2, against the actual ones for HTTP-based components in the YCSB, TeaStore and SockShop benchmarks. As shown, DyMonD provides very close categorization of the service types for most HTTP-based components under the three evaluated benchmarks. For "Front-end" services such as the one in the SockShop benchmark and "Web UI" in TeaStore, DyMonD provides three labels plus an "Unknown" because none of them has produced label frequency reaching the set threshold as described in Section 6.5. However, the diversity in the produced service labels can serve as an indication that the associated HTTP-component might be a load-balancer or a front-end service for the monitored application.

Summary The employed CNN+BiLSTM DLM yields excellent classification performance for server/client flows of a variety of service types, both encrypted and unencrypted, even if it does not have the handshake messages. Most of the misclassifications happen within the

Application	Deduced web-service label(s)	Actual web-service label
YCSB benchmark	YCSBWeb/ycsb	YCSB web-server
TeaStore benchmark	webui/category/cartaction "Unknown"	Web UI
	images/getwebimages	Image provider
	Auth/useractions	Authentication
	recommender/recommend	Recommender
	persistence/products	Persistence
	registry/services	Registry
SockShop benchmark	cart/category/catalogue "Unknown"	Front-end
	catalogue	catalogue
	Orders	orders
	paymentAuth	payment
	customers	users
	carts	carts
	Shipping	shipping

Table 6.1: Detected service labels for HTTP-based microservice vs. the actual ones

same service type or between the same "kind of" service type, in our case a cache system. For HTTP-base services, the proposed methodology to infer the fine-grained service type provides good predictions.

6.6.2 Validating Call Graph Accuracy

To assess the accuracy of the actual call graphs produced by DyMonD, we use logging tools that instrument the application and/or platform in order to collect all communication exchange between components and compare the results with what DyMonD produces. We denote as N_{both} the number of flows that are detected by both mechanisms, N_{ot} the number of flows detected by the other tool but not DyMonD, and by $N_{DyMondD}$ the number of flows detected by DyMonD but not the other tool. We generate call graphs for both the TeaStore and SockShop benchmark applications. For the TeaStore benchmark, we use the version that is instrumented with Kieker [HvH20], which is a software-based monitoring tool that invokes an application log-service whenever an application component makes a call. For DyMonD, we activate the monitoring after 60 seconds of warm-up time. The resulting DyMonD graph is slightly different from what is shown in Figure 6.2 where no application instrumentation was enabled. In this test scenario, we get N_{both} of 12 flows, N_{ot} of 4 flows and $N_{DyMondD}$ of 15 flows. The 4 flows that are not shown by DyMonD take only place at the application startup (with the image provider and the recommender both communicating with the persistence service). Thus, DyMonD is correct in not showing them since it produces the graph on demand when these communications are no longer active. Furthermore, DyMonD shows two additional components that are not presented by TeaStore's monitoring tool: the registry service and the log service. TeaStore does not generate logs for these two components considering them as external support services. However, all 5 microservices in fact communicate with the registry in a request/reply fashion and with the log-service in an unidirectional way (generating overall 15 unidirectional flows); thus DyMonD displays them. We believe this is the desired behaviour when monitoring an application during execution.

For the SockShop's call graph comparison (Figure 6.3), we have enabled Weavescope [Weaa] to visualize the application call graph by instrumenting the docker platform. Again, we run the application for 60s as warm-up time before enabling DyMonD. In this case study, the values of N_{both}, N_{ot} and $N_{DyMondD}$ are 29, 0, 0, respectively, which means that both systems detect exactly the same flows and yield the same call graph.

6.6.3 DyMonD Overhead

In this section we present the overhead of DyMonD during run-time and compare it with other monitoring tools. Kieker [HvH20], Weavescope [Weaa] and SysDig [sys] are chosen for such comparison as they represent examples for both software instrumentation- and network-based application monitoring that provide a similar functionality that DyMonD aims at. As described in Section 2.5, Kieker instruments the application to create logs during run-time that are sent to a logging-server for analysis, Weavescope monitors the network of the container platform to deduce dependencies between components running on the same host, while SysDig sniffs the messages among components and logs them in files; the information can then be used to deduce inter-dependencies.

Test environment Our basis for evaluation has been the YCSB benchmark on an extended architecture that is described in Section 3.3, and the TeaStore and SockShop benchmarks. For YCSB, we have used the same workload specifications mentioned in Section 3.3. For the TeaStore and SockShop benchmarks, we use a workload that sends HTTP requests emulating users browsing the store and purchasing items. For each application, we grow the workload by increasing the number of clients up until an application saturation is reached even without enabling application monitoring.

The experiments are performed using four DELL machines. The hardware and software configuration of those four machines are quite similar to the ones listed in Section 3.3. The clients of the YCSB, TeaStore or SockShop application are deployed on one machine, all server components on another machine in separate containers, and the visualization frontend

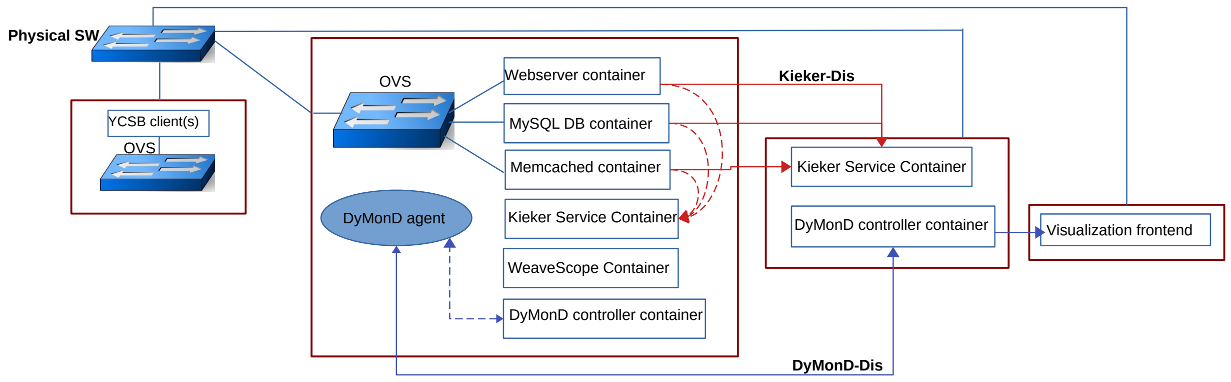


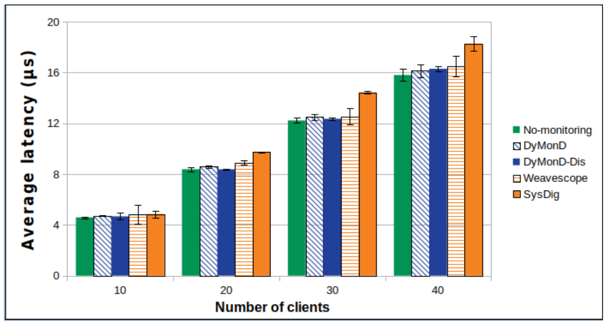
Figure 6.8: Testbed architecture for DyMonD.

on a third machine. Figure 6.8 shows an example of the setup for the YCSB benchmark. All server components are connected by 10 Gigabit Ethernet OVS ports.

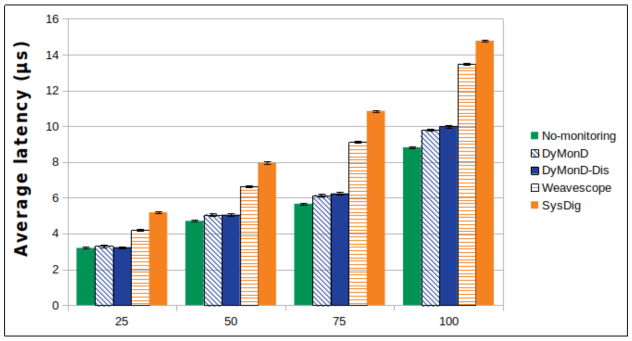
Furthermore, we have two setups for the DyMonD and Kieker monitoring tools as they both have a separate controller/ logging-server. For DyMonD the agent is either connected to a remote controller deployed on the fourth machine, referred to as "DyMondD-Dis", or to a controller that resides on the same host, referred to as "DyMonD". Similarly, we have "Kieker-Dis" where the logging messages are sent to a remote Kieker logging service on the fourth machine, and "Kieker", where the logging service is local. We run all tests for a duration that ensures that for DyMonD monitoring is active for the full test duration. This time ranges from around 1 to 2 minutes for the evaluated applications.

Impact on latency of the monitored application Figure 6.9 shows the end-to-end latency observed for the YCSB, SockShop and TeaStore applications with increasing load in terms of number of concurrent clients without monitoring and with the various monitoring options enabled. Note that Kieker was only used for the TeaStore application. For the three applications, DyMonD has the smallest impact among all evaluated monitoring approaches. Figure 6.9a shows that YCSB experiences only a small latency increase when activating DyMonD/DyMonD-Dis. Both versions have a similar impact on the monitored application because the DyMonD controller is lightweight and does not consume much resources (in our test configurations, the controller has 3% CPU utilization and almost no memory overhead). Weavescope performs similar to DyMonD; yet, DyMonD provides more context as described in Table 2.1. SysDig is significantly worse, in particular at higher loads.

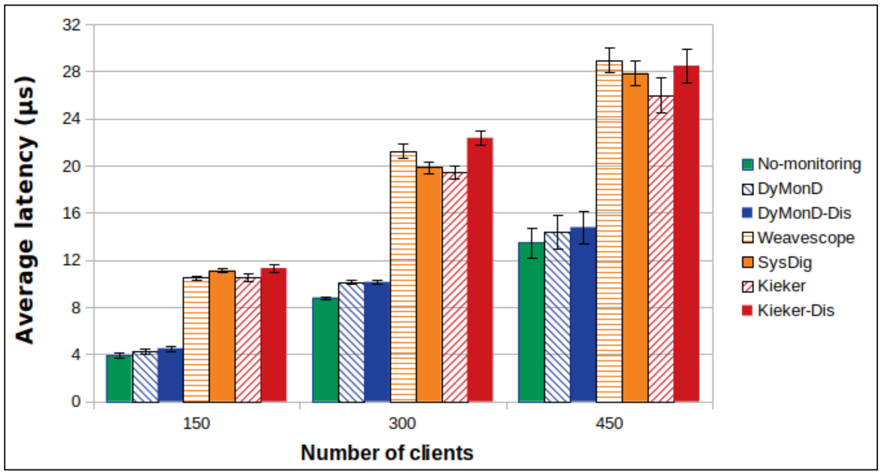
Figures 6.9b and 6.9c show that monitoring has more impact on the SockShop and TeaStore client, respectively. DyMonD performs significantly better than the other tools for



(a) Average latency reported by YCSB client.



(b) Average latency reported by SockShop client.



(c) Average latency reported by TeaStore client.

Figure 6.9: Overhead of different monitoring tools for YCSB, TeaStore and SockShop.

both applications. For the SockShop application (Figure 6.9b), the maximum overhead experienced by DyMonD is around 13% with the DyMonD-Dis setup at 100 clients, compared to around 52% and 67% overhead experienced by Weavescope and SysDig, respectively. A higher monitoring overhead is observed with the Teastore application (Figure 6.9c) where we could run a higher number of concurrent clients. For example, at 300 clients, the TeaStore application experiences a 15% latency increase with DyMonD and DyMonD-Dis, while Kieker, SysDig and Weavescope increase the latency by 120%, 125% and 140% respectively. Kieker-Dis affects the application latency more than Kieker. This is due to the high communication overhead of Kieker-Dis as we will show later in this section.

Weavescope, SysDig and DyMonD capture the messages asynchronously, that is, their actions are not within the execution path of the client request. However, they do reside on the same host as the application, thus impacting it to some degree. We assume that

DyMonD's decoupled architecture and the selection of only specific flows for monitoring at any given time make it more light-weight as we will discuss later in this section. Furthermore, DyMonD allows to dynamically switch monitoring on and off, thus providing very competitive performance.

A possible reason why Sysdig has such a high impact on response time could be because handler functions are triggered whenever certain system events such as network connections occur. They copy the event information into shared memory for further analysis, and the original system call execution is "frozen" until the handler returns, leading to delays. For Kieker, tracing and logging takes place within the application. Again it might be done synchronously, which directly impacts response time.

Computational overhead We have measured the CPU and memory utilization of the DyMonD agent in comparison to Weavescope and SysDig at their hosting node. In TeasStore with 300 clients (where the maximum monitoring overhead is observed), Weavescope and SysDig utilize 35% and 33% of the CPU and 28% and 12% of the memory capacity of the host node, respectively. Both Weavescope and SysDig sniff and log all the internal traffic on the host to infer the dependency information, which we assume contributes to this overhead. In contrast, the DyMonD agent consumes only around 16% of the host CPU and 0.1% of the host memory capacity. A large part of this consumed memory is used by the DLM to predict the service type as we will describe later in this section.

Communication overhead The communication between the DyMonD agent and controller is light-weight, as the controller only sends small request messages to the agent and the agent provides compact flow information to the controller. For instance, for our TeaStore experiment with 300 clients, only 72 TCP packets are exchanged between agent and controller. In contrast, 39,000 packets were sent from the Kieker-instrumented TeaStore application to the logging server, because Kieker traces and sends information about all individual method calls performed within Teastore. This fine-grained logging leads to a considerable communication overhead for Kieker-Dis, which is probably the reason why Kieker-Dis performs worse than Kieker in our experiment.

Location of the DLM The default DyMonD setup has the DLM deployed along with the DyMonD agent. In terms of space overhead, our DLM consumes 2.7MB disk space on the agent's host for the current 26 classes. We believe this is a reasonable footprint with the system capabilities of the host. In our experiments, we have observed that the number of classes does not significantly increase the space overhead of the model. Our deep learning

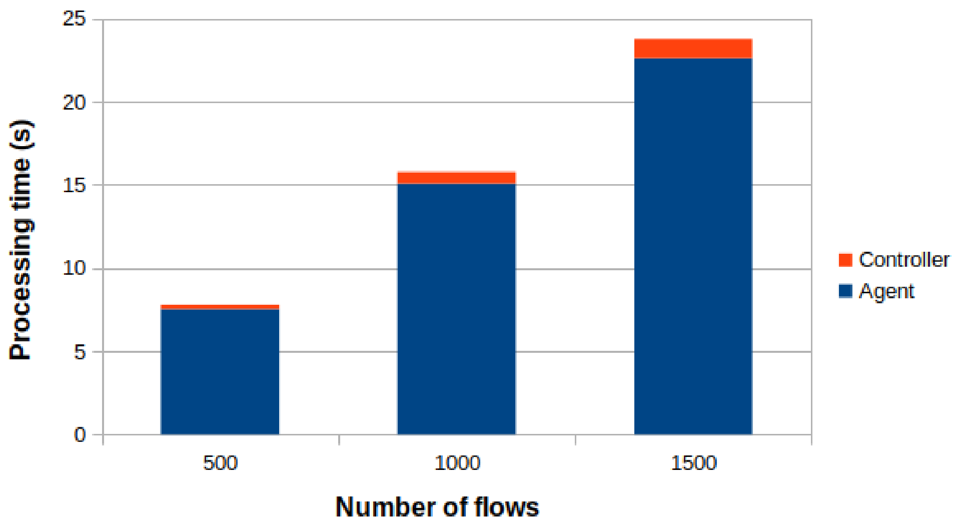


Figure 6.10: Execution time of DyMonD agent and controller analysis functions.

model has a basic size of 1.2MB even if it only classifies 2 services. Classifying 13X number of classes only doubles the basic model size. Should the DLM become significantly large, if a more complex model architecture is pursued for service classification, and storing it at every agent in the system becomes a concern, an alternative solution could be holding the DLM only at the controller. In that case, the agents send the input to the DLM (the 36 first Bytes of the 100 messages) to the controller for service identification. However, this will increase network traffic and the load on the controller and also possibly on the agent, because the model lookup might be faster than sending so many messages to the controller. To understand the trade-off, we evaluated both options with TeaStore when running 300 clients. Indeed, offloading the DLM to the remote controller in the DyMonD-Dis setup increased the observed client latency by 23.6% (which is still less than the overhead of the other monitoring tools) and increased the number of exchanged TCP packets between DyMonD agent and controller from 72 to 175. Nonetheless, this overhead increase is still much better than the other approaches (see Figure 6.9c). Thus, it might be an alternative for very large models where label lookup might be more expensive and memory might become a concern.

6.6.4 Analysis Complexity

Here we evaluate the complexity of the DyMonD in terms of the analysis time, i.e. the time spent by DyMonD agent and controller to analyze the collected network flows to infer the application call graph. In particular, we measured the time taken by the DyMonD agent

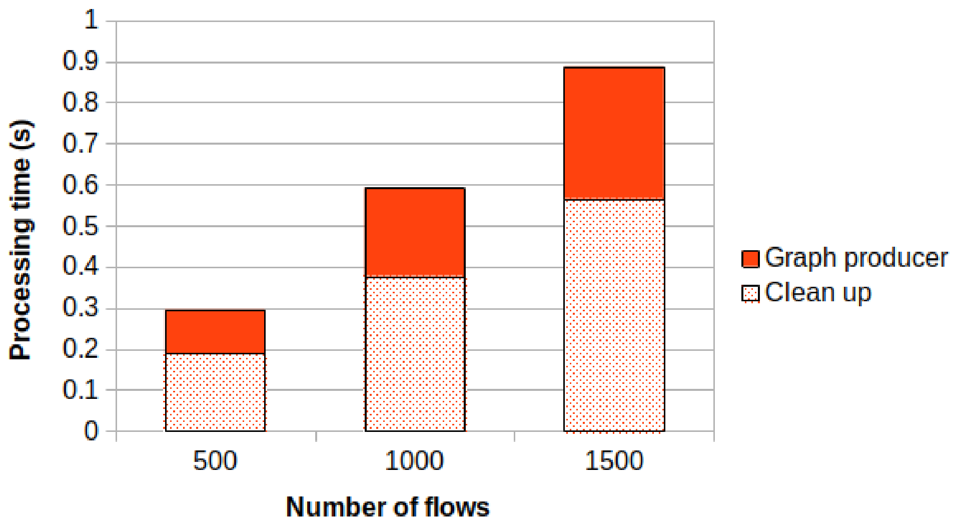


Figure 6.11: The breakdown of the execution time of DyMonD controller analysis functions.

to infer the service types and calculate the performance metrics of the collected flows, i.e. the time taken by both the service identifier and performance analyzer modules described in Section 6.2. We exclude the execution time of the flow detector and packet capture modules as they do not perform significant analysis functions and the latter's execution time mainly depends on the configured monitoring time, as described in Section 6.4. For the DyMonD controller, we measure the time taken by its functions described in Section 6.3, including extracting the call graph components, cleaning the flows and producing the call graph data. We have created a set of test flows of different sizes and contents using the YCSB, TeaStore and SockShop applications. Note that we only consider the flows with sufficient packets for the service identifier module, i.e. with 100 messages.

Figure 6.10 shows the results. Considering the test flows used, DyMonD analysis time seems to scale linearly with the number of collected flows for both the agent and controller. In general, the DyMonD agent analysis time is higher than the controller analysis time. This is expected as the agent performs more complex analysis functions than the ones executed by the controller, such as the service identification. In fact, most of the agent analysis time is consumed by the service identifier module, where on average the DLM takes around 11 milliseconds to predict the service label of one flow in our test configuration. The performance analyzer module's execution time consumes around 0.2% of the agent's execution time. At the maximum tested flow size in our configuration, which is 1.5K, the DyMonD agent and controller spent around 22 and 1 seconds, respectively to execute their analysis functions.

Figure 6.11 zooms in the execution time of the most compute intensive functions of the controller, which are the flow cleaning and the graph producer functions. As we can see, the execution time for clean up is almost double that of producing the graph. This is expected as the clean up function iterates over the flow list twice, one time for finding all the service type labels assigned for each service while the other for consolidating them over all the service's flows. On the other hand, the graph producer function iterates over the flow list only once to extract the graph nodes and edges. Consolidating the service type labels as well as the server/client labeling for 1.5k flows takes around 0.6 seconds.

6.6.5 Use-Cases

In this section, we describe two sample use cases for DyMonD: detecting and understanding performance problems and visualizing architectural changes during run-time.

Multi-Tier Performance Debugging Our first use case demonstrates how DyMonD can detect performance issues and highlight bottlenecks at run-time through the visualization of the call graph and associated application performance metrics. We have used the same YCSB application depicted in Figure 6.8 and injected a misconfiguration, by reducing the memory size for Memcached, which forced most of the web-server requests to be served by the MySQL database instead of the much faster cache. Figure 6.12 shows the call graph generated by DyMonD, where the web-server IP is given as the starting point and is activated after the application has been up and running for 5 minutes. As a result of the injected Memcached misconfiguration, the MySQL component appears in the call graph produced by DyMonD and the throughput between the web-server and MySQL is much higher than the one to Memcached (17.4Kbps vs. 5.8Kbps), leading to the substantially longer overall web-server response time compared to the one shown in Figure 6.1 ($24.2\mu s$ vs. $8.8\mu s$). Note that such cache misconfiguration causes unnecessary long delays and is not detectable by monitoring the resource utilization metrics of each component unless the database is overloaded; yet it can easily be detected when looking at each component throughput within the call graph.

Detecting application architectural patterns Here we consider a richer setting where we show how DyMonD can be used to continuously monitor the application structure which might dynamically change based on the workload data. In dynamically scalable platforms, as the application workload changes, the cloud applications are scaled up or down to meet the demands of the current workload without wasting resources. This is often done by replicating individual components and adding or removing replicas depending on need. We show here

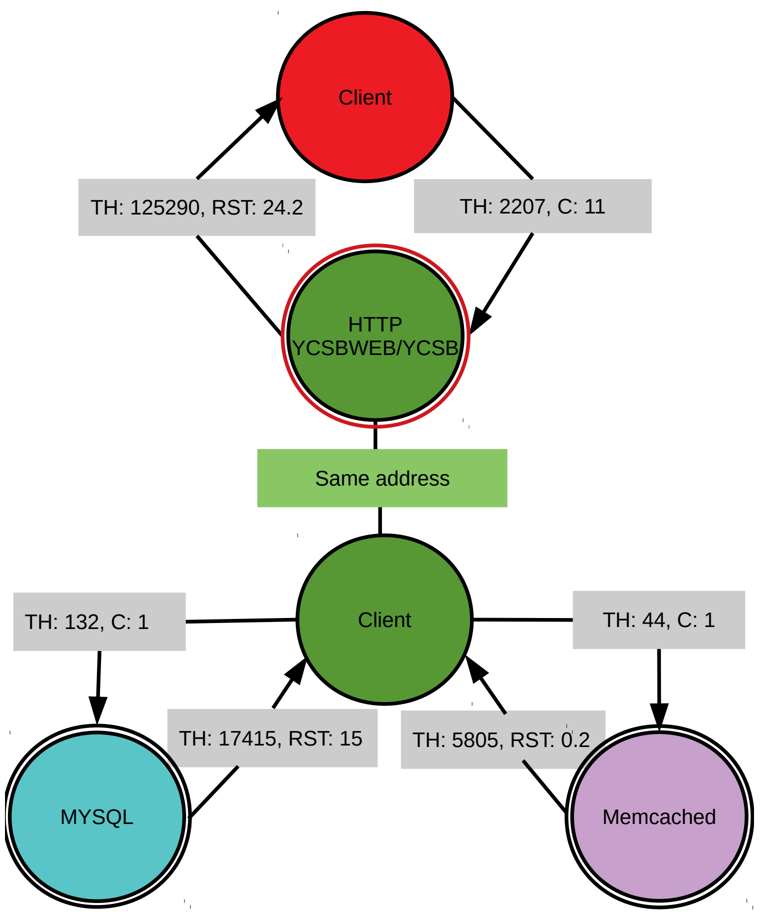


Figure 6.12: The call graph for a YCSB application with Memcached-misconfiguration, where the MySQL service (light blue) has a higher throughput towards the web-server than the Memcached service.

how DyMonD can be used to provide the administrator with an up-to-date state of the application topology at run-time. To illustrate the potential for this situation, we use the TeaStore benchmark and enable DyMonD in the online mode while giving the IP address of the Webui service as a starting point. Shortly after, we added one replica of TeaStore's authentication service. Figure 6.13 shows the part of the call graph that changed after the replica has been added. The second authentication service appeared in the call graph (the gray oval) and the Webui service is distributing its requests between the two identified authentication services (the gray and blue). We note that the TeaStore benchmark has a built-in client-side load balancer for each of its services.

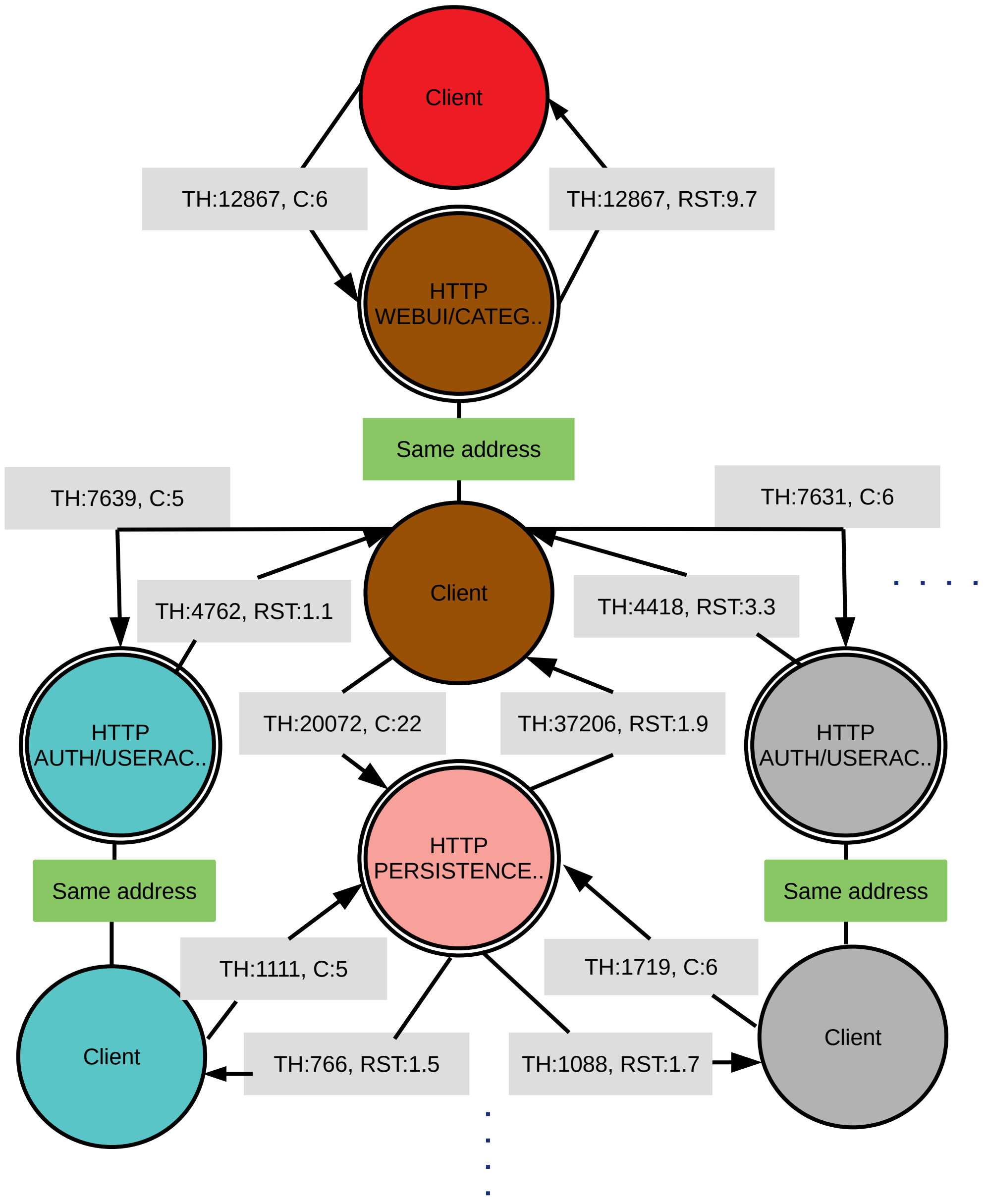


Figure 6.13: A part of TeaStore call graph with replicated authentication service.

6.7 Summary

We have presented DyMonD, a network-based framework for application call graph discovery and service identification based on the software switch-based monitoring approach proposed in Chapter 3 and the deep learning mechanisms presented in Chapter 5. We show some use cases that show DyMonD's usefulness for providing both the cloud and application administrators with the global view of the running application that is needed to diagnose and resolve performance issues.

DyMonD captures the communication flows between different application components and provides application-level performance metrics with acceptable overhead. A DLM is employed to efficiently detect the service type of each flow as well as distinguishes between the client and server flows of the service. Should some misclassifications happen, DyMonD runs a validation algorithm to best correct such misclassifications. Additionally, DyMonD uses an NLP-based approach to do a fine-grained service type classification for microservices-based architectures.

Our evaluation results confirm that DyMonD can infer the proper call graph and identify the services at run-time with acceptable overhead and good accuracy. For instance, DyMonD's overhead is 83-89% lower than that of the Weavescope and Sysdig monitoring tools, and up to 87.5-91% lower than that of the Kieker monitoring tool. In addition, it requires half of the CPU usage and around 1% of the memory usage compared to the other monitoring tools. Furthermore, DyMonD imposes less than 1% of the communication overhead caused by Kieker. Finally, DyMonD requires only a few seconds to infer the application call graph given a moderately large application traffic. We believe this is an acceptable overhead as DyMonD should be enabled for a short duration.

In contrast to the software instrumentation-based and system monitoring tools, DyMonD has some limitations. For example, DyMonD will miss the inner communications between the processes of the same application component that do not go through the network. In such case, coupling DyMonD with one of the system tracing tools [DD08, sys, ENOP21] listed in Section 2.5 would be beneficial to provide a deeper insight into the application performance.

7 Final Conclusions & Future Work

In this chapter, we will briefly recap the research motivation and contributions presented in this thesis, and outline the possible research directions for future work.

7.1 Conclusions

Many cloud applications follow a distributed service-oriented architecture, where the execution of an external client request leads to calls to various components, each of them providing a different service. A basic example is a multi-tier architecture with a web-server front-end, a Memcached-based caching service, and a MySQL database. But more and more applications are further divided into smaller components with tens of microservices. These architectures have further expanded to include replicated and/or distributed services for scalability and reliability, e.g. a multi-node Cassandra key-value store. The complexity of these cloud applications demands advanced application monitoring to detect performance bottlenecks and help troubleshoot faulty applications. In this thesis, we first reviewed the various monitoring aspects required for such distributed applications, including system, network and application layer performance metrics. From there, this thesis focused on the latter one.

7.1 Conclusions

We have identified major features to be provided by application performance monitoring tools to be able to monitor a distributed application holistically. This includes providing some component-level performance metrics, such as response time and throughput, information about the dependency of components during run-time in form of a call graph as well as identifying the service type provided by each component. To provide such monitoring functionalities, performance related data collection and analysis are needed. Common approaches to collect the relevant data are to instrument the application or the underlying platform to create log messages. An obvious disadvantage is the application and/or platform dependency. A different approach is to look at the messages exchanged between components to derive some application specific measures in non-intrusive manner that can be applied in a wide range of settings. Moreover, new networking paradigms such as SDN and NFV are exploited in some of these network-based monitoring approaches to enable dynamic instantiation of the monitoring functionalities. However, many of these network-based application monitoring approaches do not provide a holistic application monitoring functionality and/or impose a considerable overhead.

As such, we provided a holistic solution for dynamic application level monitoring, that overcomes the shortcomings of existing work and enables the application monitoring functionality to be provided as a service by the cloud infrastructure with acceptable overhead. Towards this end, we make three important contributions.

Our first contribution is the "sniffer", a switch-based application monitoring approach that does not introduce any direct delay at the switch, enhances the possibilities of conducting some of the analysis at the network components and significantly reduces the communication overhead compared to other network-based monitoring approaches.

To illustrate how the sniffer-based approach can be embedded into a MaaS platform, we designed and implemented a MaaS prototype with the sniffer as its core component. A user interface allows clients to initiate monitoring requests which are delegated to the appropriate switches which in turn extract and analyze the relevant application messages using our monitoring approach, and send the calculated measures to the client for visualization in near real-time. In addition, we have designed the MaaS software so that new types of services and new performance measures can be added to the MaaS in an incremental manner. Our performance evaluations show that MaaS has a reasonable computational and communication overheads, as well as a little impact on the monitored application.

The third contribution, DyMonD, provides a dynamic application call graph discovery framework that dynamically discovers and visualizes dependencies between application components, providing a global view of the application at run-time along with some application performance metrics. DyMonD has the same architecture as the MaaS platform, while extending the functionally of the backend monitoring agents to support DyMonD functionality. In particular, in addition to analyzing the network flows between the application components to deduce application performance metrics such as response time and throughput, DyMonD deduces the dependency information between the application components and identifies the service type of each network flow by analyzing the network messages exchanged between the application components. Moreover, DyMonD has the option to use an NLP-based approach to perform a further fine-grained service type identification by determining more application-specific service types (e.g. "authentication" or "recommender"), which is particularly useful for microservice-based architectures.

DyMonD employs a novel BiLSTM-based deep learning model (DLM) to dynamically classify the service type of network flows as well as the flow direction being a client or server flow. In this prospective, we first performed a through analysis to evaluate the use of various DLMs and design parameters in the context of service type identification of network flows. In particular, we generated a large flow-based dataset of several service types and the performance of LSTM- and CNN-based DLMs as well as the proposed BiLSTM-based DLMs are compared, while using header-based and payload-based data for training. The trade-offs and the impact of various parameters on the classification performance are also investigated. Our results show that header-based approaches do not work well if the captured flows do not contain the handshake messages at connection setup and when uni-directional flows are considered, two aspects which are important for DyMonD. Therefore, DyMonD uses unidirectional payload-based data to train a combined CNN and BiLSTM DLM, i.e. CNN+BiLSTM, that provides the best classification results for a wide range of service types, even if encryption is used. Our performance evaluation results confirm that DyMonD can infer the proper call graph and identify the services at run-time with good accuracy. DyMonD is lightweight as it requires few resources, and has little communication overhead and impact on the monitored application compared to competing approaches.

7.2 Future Work

7.2.1 Scalability evaluation

Our MaaS solutions are fully implemented, but additional work is required for using them to monitor cloud data center traffic at scale. This includes evaluating the multi-agent setup described in Section 6.3.2 and handling the lack of synchronization among the distributed agents' clocks. In addition, it would be interesting to explore how to extend MaaS performance analysis functions to support an efficient aggregation of monitored data in case the network infrastructure allows for multi-path routing. In this case, the application network flows' data might be captured from multiple monitoring agents deployed along those paths. For example, if the messages of the same network flow are distributed over multiple network paths, multiple monitoring agents will send information about that flow to the controller, and the throughput performance metric in the current implementation would update the value with the latest received one, while a summation of those results would be the correct calculation for the throughput in that case.

Moreover, the performance and overhead of our monitoring agent should be evaluated for intensive workloads, for example, when many applications are to be monitored simultaneously. In this prospective, evaluating the performance of integrating our monitoring agent with fast packet processing libraries such as [DPD] and XDP [XDP] to support such high traffic volumes would be also an interesting future direction to explore.

Furthermore, extending our MaaS solutions to services that use different communication protocols, other than the currently supported request/reply ones, such as the pipeline and publish/subscribe protocols should be considered.

7.2.2 Monitoring Using a P4-based Switch

Currently, we have the monitoring agent of the MaaS and DyMonD systems deployed as a software process on the host of the software switch. For hardware switches, one of the selective mirroring approaches presented in Section 3.1 can be used to forward the relevant packets to the monitoring agent. An alternative solution is to realize the monitoring agent functionality using a P4-based switch. As mentioned in Chapter 2, P4 [KCBH21] is a Domain Specific Language (DSL) that specifies the packet processing pipeline of a network device, such as a switch or router, designed to be protocol independent and to abstract from the variability of network hardware and interfaces. A P4 program tends to be concise, and expresses exactly the processing behaviour of a target (usually network switches). In contrast to a general purpose programming language such as C or Python, P4 is a unified abstraction that is optimized around data forwarding, with only basic computation capabilities enabled, such as arithmetic operations. P4 is receiving high attention in the network management research community as it enables the dynamic programming of the network at its lowest level of the forwarding elements. This flexibility of changing the networking behavior is cost effective and allows for more rapid response to the frequent change requirements in today's networks. The network administrators can remove any unused routing logic that is embedded in the vendor switches and/or change it according to their needs without the necessity of purchasing new hardware or being only limited by the vendor programming tools.

Given the existence of such flexibility in programming the switch internal functions, we envision that some if not all of the functionality of the sniffer and monitoring agent can be integrated with any hardware/software switch that is supporting P4. For instance, the listener and data extractor functions in Figures 3.4 and 6.8 that filter out certain packets that are relevant to the monitoring request and extract some data to construct individual flows can be implemented using P4 functions. The performance analyzer function of the monitoring agent of DyMonD can also be realized as P4 functions. For example, per flow data can be monitored such as the number of sent bytes and packets in order to deduce some performance metrics such as throughput. Moreover, the payload data required by the service identifier module of the DyMonD agent can also be extracted using P4 function. All the extracted data could then be forwarded to the controller to perform the further analysis.

However, as we mentioned in Chapter 2, P4 performance is not yet well studied. Therefore, an extensive performance evaluation of such implementation should be conducted. For example, we already saw that moving the service identification using DLM from the DyMonD agent to the controller can lead to overhead. Moreover, analysis functions that require deep packet inspection, such as the request path and methods provided by the MaaS framework or the fine-grained classification of microservices performed by the DyMonD agent, would be challenging to be performed using P4 as it mainly handles fixed-length data. Thus, functionality that requires deep packet inspection would require mirroring the packet data to a software process to perform the required analysis. As mentioned in Chapter 2 and 3, having the switch doing more processing functions on each packet, such as extracting x bytes of the first 100 packets required by the service identifier module or mirroring the packet data to a remote process, may interfere with the main switching function and hence lead to switching latency that could also impact the performance of the monitored application. A comprehensive evaluation of this approach can reveal such interesting observations and trade-offs.

7.2.3 Spicy Integration

Spicy [SAH] is a network-specific programming language that was developed for dissecting wire format data and file formats. Given a message sent using a specific communication protocol, e.g. MySQL, Spicy employs a type-based style that expresses elements at the semantic level of to dissect the message and transform it into a generic message format. Spicy has a just-in-time compiler toolchain that creates C++ code for developed Spicy modules, which in turn, can be integrated with any external tool that parse network packets. For example, Sommer et. al. [SAH] show a use case of integrating a Spicy module to parse DNS packet with Wireshark.

Similarly, Spicy modules could be developed to replace the MaaS protocol parser API described in Section 4.3. The integration of those Spicy modules with the MaaS framework could be as following: At startup, the MaaS agent compiles the user-defined Spicy modules for each protocol just-in-time. For each received protocol's packet, the spicy parsing module is executed, and needed information for performance metrics is returned back to the agent. However, as far as we are aware of, Spicy's performance has not yet been thoroughly evaluated. Therefore, a performance evaluation and comparison of such integration with the original MaaS parser API is still needed.

7.2.4 Optimization of DyMonD's Service Identifier Module

The main functionality of the DyMonD's monitoring agent is implemented in C++ because of the need for performing monitoring and capturing application data at high speed, while Python libraries are used to implement the DLM for service identification. Our evaluation results in Section 6.6 show that most of DyMonD's execution time at the agent is taken by the DLM to predict the service type of the flows (around 11 milliseconds per flow). We believe that is because interpreted languages such as Python are inefficient for computationally intensive tasks such as the ones executed by the DLM layers.

As future work, we would like to explore more efficient APIs to realize the service identifier. For instance, TensorFlow not only provides the Python API but also APIs for other languages such as C, Java, Go, etc. Although the Python API has the most complete functionality, it is still possible to load and use a pre-trained TensorFlow DLM in other languages such as C/C++. Another option is using a library that can implement the DLM layer functions in more efficient language such as C/C++. For example, the Keras2c library[1] performs automatic mapping of each layer in a DLM to C functions. These C functions can then be compiled into a static library and used by the service identifier module. However, the accuracy of such mapping and prediction of the converted DLM needs to be validated.

Furthermore, we plan to extend DyMonD to enable automatic updates to the service identification module should more services be added over time or the number of "unknown"-labeled services exceeds a certain limit [ZLWY19, RYCW21].

[1]Kera2c `https://github.com/f0uriest/keras2c`

7.2.5 Performance Monitoring for Serverless Applications

Serverless (or function as a service (FaaS)) is a cloud computing execution model that allows developers to build and run applications without having to manage servers. There are still servers in serverless, but they are abstracted away from application development. The cloud provider dynamically manages the allocation, provisioning and scaling of the servers. Serverless applications are broken up into individual functions that can be invoked and scaled individually. In particular, developers can simply package their code in containers for deployment. Once deployed, serverless functions are executed on demand and automatically scale up and down as needed. Serverless functions are usually metered on-demand through an event-driven execution model. As a result, when a serverless function is sitting idle, it does not cost anything. The concepts of serverless architecture and FaaS have grown along with the popularity of containers and on-demand cloud offerings.

Serverless application monitoring allows developers to gain important insight on what happens during each execution and event, and errors become more easily visible. Example of performance metrics of interest to be collected for serverless applications' components (i.e. functions) are: latency, cold starts (i.e. time required to start a new function instance), and invocation errors. However, increased complexity, loss of control over software layers, the large number of participating functions and backend services complicate the task of collecting such performance metrics for serverless applications. There are a lot of units to monitor, the life cycles are short, and configuring monitoring agents directly contributes to latency and cost overhead.

Many of the FaaS platforms, such as AWS Lambda[2], provide a variety of performance metrics, such as system related ones (i.e. disk space, CPU, and network I/O), as well as large log files. A tedious but widespread strategy is the manual analysis of such log data. However, finding all mandatory parameters in such large log files is challenging and some are often missing. Thus, there is a need for an automated monitoring analysis tool.

Another area that is significantly lacking in support at present is distributed monitoring that enables understanding what is happening for a business request as it is processed by a number of serverless functions. This kind of monitoring is challenging due to the heterogeneity and vendor lock-in of the FaaS platform providers. Many of the offered monitoring approaches are poorly documented, or at least poorly understood by most users. Therefore, an interesting research direction would be a unified distributed performance monitoring framework for serveless applications.

[2]`https://aws.amazon.com/lambda/`

7.2.6 Employing MaaS in Network-based Solutions for Other Domains

The flexibility and programmability of today's networks open new research directions for implementing some of the traditional application semantics into the network to achieve better and real-time performance, while having the advantage of implementation and deployment flexibility independently of the application or platform. For example, there already exists work to provide application-level load balancing as a network service by leveraging NFV technology [ZWH16]. In particular, a NFV-based middlebox can be deployed in the network to automatically redirect packets to appropriate servers and route the responses from servers directly to clients. Moving such application-level functionalities to the network should enhance throughput and latency compared to a load balancer that is completely implemented as an application or platform component. Another example is building the caching layer into the network providing something like a cache-as-a-service (CaaS), where the caching service can be added to any application, as well as enhancing the caching service performance. This can be again achieved by leveraging the power and flexibility of programmable networks to cache data in the network. For instance, a network function is implemented in [JLZ+17] to detect, index, store, and serve key-value items. This leverages the fact that switches are already placed on the data path and have access to all cache queries through the system. Furthermore, if a ToR switch is employed as the cache for a key-value storage rack, this would incur little to no overhead in terms of latency and cost.

The integration of the MaaS functionality into such systems could further enhance its performance and capability. For instance, collecting performance metrics such as the frequency of each request and the estimated number of total requests for the in-network load balancer can be used to periodically adjust the replication factor for popular contents based on changing workloads on predefined intervals, adding on-demand elasticity for load balancing services. Similarly, performance metrics such as request rate for each cached item can help identifying which contents are hot at a particular time optimizing the in-network caching service.

Finally, our MaaS architecture can also be employed to reduce the cost of real-time monitoring of data centers. Data center's monitoring is a broad process that focuses on monitoring the entire data center infrastructure. This includes providing end-to-end visibility across all data center components: computers, storage, network, power, heating and security. As data centers can host over a hundred thousand servers, the potential number of data points is in the order of hundreds of millions. The communication overhead induced by such real-time monitoring is huge enough to influence the performance of a data center. One method to reduce this overhead is bringing the computations closer to the source of the data. This reduces the amount of hops required for the data to reach their destination, and in turn limits the communication overhead. ToR switches are excellent candidates for that purpose due to their proximity to the servers that are being monitored. Deploying the monitoring agents of the MaaS framework at each ToR switch, while extending the monitoring agent functionality to support the monitoring demands in that context, will reduce the spectrum of the monitoring data transfer to the range of one rack. In particular, every monitoring agent would be responsible for processing and analysing the data of their rack only. Therefore, the monitoring agent would only have to handle the network traffic of a limited amount of servers. We envision such implementation as an interesting area of future work as well as conducting a quantitative performance evaluation of the MaaS architecture, compared to monitoring based on a traditional cloud computing approach.

Bibliography

[ACA$^+$15] Sara Ayoubi, Yiheng Chen, Chadi Assi, Tarek Khalifa, and Khaled Bashir Shaban. Multicast tree repair and maintenance in the cloud. In *Proceedings of IEEE International Conference on Cloud Computing, (CLOUD'15), USA*, pages 829–835, 2015.

[ADP$^+$] Pedro Amaral, João Dinis, Paulo Pinto, Luís Bernardo, João Tavares, and Henrique S. Mamede. Machine learning in software defined networks: Data collection and traffic classification. In *Proceedings of 24th IEEE International Conference on Network Protocols (ICNP'16), Singapore, 2016*, pages 1–5.

[ama] Cloudwatch. `https://aws.amazon.com/de/cloudwatch/`, [March 2022].

[App] Appdynamics. `https://www.appdynamics.com`, [March 2022].

[ASRC14] Diego Adrada, Esteban Salazar, Julián Rojas, and Juan Carlos Corrales. Automatic code instrumentation for converged service monitoring and fault detection. In *Proceedings of 2014 28th International Conference on Advanced Information Networking and Applications Workshops*, pages 708–713, 2014.

[azu] Microsoft azure monitor. `https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview`, [March 2022].

[CAR] Ivano Cerrato, Mauro Annarumma, and Fulvio Risso. Supporting fine-grained network functions through intel DPDK. In *Proceedings of Third European Workshop on Software Defined Networks, (EWSDN'14), Hungary, 2014*, pages 1–6.

[Cas] Apache cassandra. `https://cassandra.apache.org`, [March 2022].

[CBB] Valentín Carela-Español, Tomasz Bujlow, and Pere Barlet-Ros. Is our ground-truth for traffic classification reliable? In *Proceedings of 15th International Passive and Active Measurement Conference, (PAM'14), USA, 2014*, pages 98–108.

[CC16] Jiajia Chen and Weiqing Cheng. Analysis of web traffic based on HTTP protocol. In *Proceedings of 24th International Conference on Software, Telecommunications and Computer Networks, (SoftCOM'16), Croatia*, pages 1–5, 2016.

[CFAI17] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In *Proceedings of Second International Workshop on Pre- and Post-Deployment Verification Techniques, (PrePost@iFM'17), Torino, Italy*, volume 254, pages 15–28, 2017.

[CLKdR16] ChoongHee Cho, JungBok Lee, Eun-Do Kim, and Jeong dong Ryoo. A sophisticated packet forwarding scheme with deep packet inspection in an openflow switch. In *Proceedings of IEEE International Conference on Software Networking, (ICSN), South Korea*, pages 1–5, 2016.

[CMF$^+$14] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14), USA*, pages 217–231, 2014.

[CMLX15] Tommy Chin, Xenia Mountrouidou, Xiangyang Li, and Kaiqi Xiong. Selective packet inspection to detect dos flooding using software defined networking (SDN). In *Proceedings of IEEE 35th International Conference on Distributed Computing Systems Workshops, (ICDCS) Workshops, USA*, pages 95–99, 2015.

[CST$^+$] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of 1st ACM Symposium on Cloud Computing, (SoCC'10), USA, 2010*, pages 143–154.

[CVW$^+$21] Carlos Garcia Cordero, Emmanouil Vasilomanolakis, Aidmar Wainakh, Max Mühlhäuser, and Simin Nadjm-Tehrani. On generating network traffic datasets with synthetic attacks for intrusion detection. *ACM Transactions on Privacy and Security*, 24(2):8:1–8:39, 2021.

[Dat] Datadog. `https://www.datadoghq.com`, [March 2022].

[DB2] IBM. `https://www.ibm.com/ca-en/products/db2-database`, [March 2022].

[dCRCG$^+$16] Guilherme da Cunha Rodrigues, Rodrigo N. Calheiros, Vinicius Tavares Guimaraes, Glederson Lessa dos Santos, Márcio Barbosa de Carvalho, Lisandro Zambenedetti Granville, Liane Margarida Rockenbach Tarouco, and Rajkumar Buyya. Monitoring of cloud computing environments: concepts, solutions, trends, and future directions. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Italy*, pages 378–383, 2016.

[DD08] M. Desnoyers and M. Dagenais. Lttng: Tracing across execution layers, from the hypervisor to user-space. In *Linux Symposium*, page 101, 2008.

[DFC$^+$16] Lautaro Dolberg, Jérôme François, Shihabur Rahman Chowdhury, Reaz Ahmed, Raouf Boutaba, and Thomas Engel. A generic framework to support application-level flow management in software-defined networks. In *Proceedings of IEEE NetSoft Conference and Workshops, (NetSoft'16), South Korea*, pages 121–125, 2016.

[DLMG] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of encrypted and VPN traffic using time-related features. In *Proceedings of International Conference on Information Systems Security and Privacy, (ICISSP'16), Italy, 2016*, pages 407–414.

[DP11] Vladimir Deart and Alexander Pilugin. HTTP traffic model for web2.0 and future webx.0. *International Journal of Wireless Networks and Broadband Technologies*, 1(1):50–55, 2011.

[DPD] Data plane development kit (DPDK). `http://www.dpdk.org`, [March 2022].

[Dyn] Dynatrace. `https://dynatrace.status.io`, [March 2022].

[EFK21] Mona Elsaadawy, Laetitia Fesselier, and Bettina Kemme. Demo: Application monitoring as a network service. In *Proceedings of 41st IEEE International Conference on Distributed Computing Systems, (ICDCS'21), USA,*, pages 1091–1094, 2021.

[Els19] Mona Elsaadawy. Monitoring as a service for SDN based cloud data centers. In *Proceedings of the 20th International Middleware Conference Doctoral Symposium, (Middleware'19), USA*, pages 36–40, 2019.

[ELW$^+$21] Mona Elsaadawy, Aaron Lohner, Ruoyu Wang, Jifeng Wang, and Bettina Kemme. Dymond: dynamic application monitoring and service detection framework. In *Proceedings of the 22nd International Middleware Conference: Demos and Posters, (Middleware'21), Virtual Event / Canada*, pages 8–9, 2021.

[ENOP21] Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. CAT: content-aware tracing and analysis for distributed systems. In *Middleware '21: 22nd*

International Middleware Conference, Québec City, Canada, December 6 - 10, 2021, pages 223–235, 2021.

[ERP+21] Ismael Essop, José Carlos Ribeiro, Maria Papaioannou, Georgios Zachos, Georgios Mantas, and Jonathan Rodriguez. Generating datasets for anomaly-based intrusion detection systems in iot and industrial iot networks. *Sensors*, 21(4):1528, 2021.

[ERR18] Silvia Esparrachiari, Tanya Reilly, and Ashleigh Rentz. Tracking and controlling microservice dependencies. *ACM Queue*, 16(4):98–104, 2018.

[ESEFAM21] Ahmed M. El-Shamy, Nawal A. El-Fishawy, Gamal Attiya, and Mokhtar A. A. Mohamed. Anomaly detection and bottleneck identification of the distributed application in cloud data center using software–defined networking. *Egyptian Informatics Journal*, 22(4):417–432, 2021.

[ESn16] ESnet. iperf - the ultimate speed test tool for tcp, udp and sctp, 2016.

[FDN14] Hamid Farhadi, Ping Du, and Akihiro Nakao. User-defined actions for SDN. In *Proceedings of International Conference of Future Internet 2014, (CFI'14), Japan*, pages 3:1–3:6, 2014.

[FEAD] Quentin Fournier, Naser Ezzati-Jivan, Daniel Aloise, and Michel R. Dagenais. Automatic cause detection of performance problems in web applications. In *Proceedings of IEEE International Symposium on Software Reliability Engineering Workshops, (ISSRE'19), Berlin, Germany, 2019.*

[Fes] Laetitia Fesselier. Application monitoring as a network service. McGill COMP400 report: `https://www.cs.mcgill.ca/~lfesse/doc/Application%20Monitoring%20As%20A%20Network%20Service-v2.pdf`, [May-2019].

[FLH+00] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. Generic routing encapsulation (GRE). *RFC*, 2784:1–9, 2000.

[FRR+14] Michael Finsterbusch, Chris Richter, Eduardo Rocha, Jean-Alexander Muller, and Klaus Hanssgen. A survey of payload-based traffic classification approaches. *IEEE Communications Surveys & Tutorials*, 16(2):1135–1156, 2014.

[GAM15] Brian R. Granby, Bob Askwith, and Angelos K. Marnerides. SDN-PANDA: software-defined network platform for anomaly detection applications. In *Proceedings of 23rd IEEE International Conference on Network Protocols, (ICNP'15), USA*, pages 463–466, 2015.

[GEJD21] Loïc Gelle, Naser Ezzati-Jivan, and Michel R. Dagenais. Combining distributed and kernel tracing for performance analysis of cloud applications. *Electronics*, 10(21), 2021.

[GFS05] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional LSTM networks for improved phoneme classification and recognition. In *International Conference on Artificial Neural Networks*, ICANN'05, pages 799–804. Springer, 2005.

[GKK$^+$19] Sahil Garg, Kuljeet Kaur, Neeraj Kumar, Georges Kaddoum, Albert Y. Zomaya, and Rajiv Ranjan. A hybrid deep learning-based model for anomaly detection in cloud datacenter networks. *IEEE Transactions on Network and Service Management*, 16(3):924–935, 2019.

[Goo19] Google. Stackdriver monitoring for applications running on google cloud platform and amazon web services., 2019.

[Goo22] Google. Google cloud's operations suite, Last accessed: March 2022.

[GS20] Marcello Guarro and Ricardo G. Sanfelice. Hyntp: An adaptive hybrid network time protocol for clock synchronization in heterogeneous distributed systems. In *2020 American Control Conference, ACC 2020, Denver, CO, USA, July 1-3, 2020*, pages 1025–1030, 2020.

[Ham14] Hamid Farhadi and Ping Du and Akihiro Nakao. Enhancing openflow actions to offload packet-in processing. In *Proceedings of the 16th Asia-Pacific Network Operations and Management Symposium, (APNOMS'14), Taiwan*, pages 1–6, 2014.

[HBHH] Jonatan Heyman, Carl Byström, Joakim Hamrén, and Hugo Heyman. Locust: An open source load testing tool.

[HLZ$^+$] J. Hwang, G. Liu, S. Zeng, F. Y. Wu, and T. Wood. Topology discovery and service classification for distributed-aware clouds. In *Proceedings of IEEE International Conference on Cloud Engineering, USA, 2014*, pages 385–390.

[HRW15] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, 2015.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[HSG$^+$] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang. Net-cohort: detecting and managing VM ensembles in virtualized data centers. In *Proceedings of 9th International Conference on Autonomic Computing, (ICAC'12), USA, 2012*, pages 3–12.

[HvH20] Wilhelm Hasselbring and André van Hoorn. Kieker: A monitoring framework for software engineering research. *Software Impacts*, 5:5, 2020.

[J. 17] J. Thalheim et al. Sieve: actionable insights from monitored metrics in distributed systems. In *Proceedings of 18th ACM/IFIP/USENIX Middleware Conference, USA*, pages 14–27, 2017.

[Jae] Jaeger. `https://www.jaegertracing.io/`, [Feb. 2022].

[JG17] Ravi Shankar Jha and Punit Gupta. Clock synchronization in iot network using cloud computing. *Wirel. Pers. Commun.*, 97(4):6469–6481, 2017.

[JKW17] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Performance monitoring and root cause analysis for cloud-hosted web applications. In *Proceedings of the 26th International Conference on World Wide Web, (WWW'17), Australia*, pages 469–478, 2017.

[JLZ$^+$17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 121–136. ACM, 2017.

[JMD14] Yosr Jarraya, Taous Madi, and Mourad Debbabi. A survey and a layered taxonomy of software-defined networking. *IEEE Communications Surveys and Tutorials*, 16(4):1955–1980, 2014.

[KAB$^+$14] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan J. Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, (NSDI'14), Seattle, WA, USA*, pages 203–216, 2014.

[KCBH21] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155, 2021.

[KRV+15] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), 2015.

[KSK21] Ouassim Karrakchou, Nancy Samaan, and Ahmed Karmouch. Ep4: An application-aware network architecture with a customizable data plane. In *Proceedings of IEEE 22nd International Conference on High Performance Switching and Routing, (HPSR)*, pages 1–6, 2021.

[Lam14] M. Lamourine. Openstack. *login Usenix Mag.*, 39(3), 2014.

[Lin19] Linux. Linux programmer's manual: Bpf, 2019.

[LKC+16] F. Li, A. M. Kakhki, D. R. Choffnes, P. Gill, and A. Mislove. Classifiers unclassified: An efficient approach to revealing IP traffic classification rules. In *Proceedings of ACM on Internet Measurement Conference, (IMC'16), USA*, pages 239–245, 2016.

[LKH+] Hyun-Kyo Lim, Ju-Bong Kim, Joo-Seong Heo, Kwihoon Kim, Yong-Geun Hong, and Youn-Hee Han. Packet-based network traffic classification using deep learning. In *Proceedings of International Conference on Artificial Intelligence in Information and Communication, (ICAIIC'19), Japan, 2019*, pages 46–51.

[LKK+19] Hyun-Kyo Lim, Ju-Bong Kim, Kwihoon Kim, Yong-Geun Hong, and Youn-Hee Han. Payload-based traffic classification using multi-layer lstm in software defined networks. *Applied Sciences*, 9(12):16, 2019.

[LLL+12] George Lee, Jimmy J. Lin, Chuang Liu, Andrew Lorek, and Dmitriy V. Ryaboy. The unified logging infrastructure for data analytics at twitter. *PVLDB*, 5(12):1771–1780, 2012.

[Llo] I. Llorente. Opennebula - latest innovations in private cloud computing. In *Proceedings of 4th International Conference on Cloud Comp. & Serv. Sci., Spain, 2014.*

[LTRW] Guyue Liu, Michael Trotter, Yuxin Ren, and Timothy Wood. Netalytics: Cloud-scale application performance monitoring with SDN and NFV. In *Proceedings of 17th International Middleware Conference, Italy, 2016*, pages 1–14.

[LTW+17] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, 20(3):2575–2589, 2017.

[LW15] G. Liu and T. Wood. Cloud-scale application performance monitoring with SDN and NFV. In *IEEE International Conference on Cloud Engineering, (IC2E'15), USA*, pages 440–445, 2015.

[MAB+08] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. Openflow: enabling innovation in campus networks. *Computer Communication Review*, 38(2):69–74, 2008.

[MAR+14] João Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Andrei Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, (NSDI'14), USA*, pages 459–473, 2014.

[MCSL17] Manuel López Martín, Belén Carro, Antonio Sánchez-Esguevillas, and Jaime Lloret. Network traffic classifier with convolutional and recurrent neural networks for internet of things. *IEEE Access*, 5:18042–18050, 2017.

[MDD+14] Mallik Mahalingam, Dinesh G. Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. *RFC*, 7348:1–22, 2014.

[Mem] Memcached. `https://memcached.org`, [March 2022].

[MFP+22] Francesco Musumeci, Ali Can Fidanci, Francesco Paolucci, Filippo Cugini, and Massimo Tornatore. Machine-learning-enabled ddos attacks detection in P4 programmable networks. *Journal of Network and Systems Management*, 30(1):21, 2022.

[MHM+14] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, and T. V. Lakshman. Application-aware data plane processing in SDN. In *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking, (HotSDN'14), USA*, pages 13–18, 2014.

[MKK17] Mehrnoosh Monshizadeh, Vikramajeet Khatri, and Raimo Kantola. An adaptive detection and prevention architecture for unsafe traffic in SDN enabled mobile networks. In *Proceedings of IFIP/IEEE Symposium on*

Integrated Network and Service Management (IM), Portugal, pages 883–884, 2017.

[MN15] Ibrahim Ben Mustafa and Tamer Nadeem. Dynamic traffic shaping technique for HTTP adaptive video streaming using software defined networks. In *Proceedings of 12th Annual IEEE International Conference on Sensing, Communication, and Networking, (SECON'15), USA*, pages 178–180, 2015.

[Mon] Monetdb. `https://www.monetdb.org`, [March 2022].

[MSG$^+$16] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys and Tutorials*, 18(1):236–262, 2016.

[MyS] Mysql. `https://www.mysql.com/`, [March 2022].

[Net] Building netflix's distributed tracing infrastructure. `https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304`, [March 2022].

[Net12] Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers Security*, 31(3):357–374, 2012.

[New] Newrelic. `https://newrelic.com/`, [March 2022].

[NLT] Natural language toolkit. `https://www.nltk.org`, [March 2022].

[Ope] openQRM. `http://www.openqrm-enterprise.com`, [Sept-2021].

[Ovi] ovirt. `https://www.ovirt.org`, [March 2022].

[PHW11] Byungchul Park, James W. Hong, and Young J. Won. Toward fine-grained traffic classification. *IEEE Communications Magazine*, 49(7):104–111, 2011.

[Pos] Postgresql. `https://www.postgresql.org/about/`, [March 2022].

[PPN$^+$20] Yashwant Singh Patel, Aditi Page, Manvi Nagdev, Anurag Choubey, Rajiv Misra, and Sajal K. Das. On demand clock synchronization for live VM migration in distributed cloud data centers. *J. Parallel Distributed Comput.*, 138:15–31, 2020.

[PSkJ17] M. R. Parsaei, M. J. Sobouti, S. R. khayami, and R. Javidan. Network traffic classification using machine learning techniques over software defined networks. *International Journal of Advanced Computer Science and Applications, (IJACSA)*, 8(7), 2017.

[ptr] ptrace. `https://man7.org/linux/man-pages/man2/ptrace.2.html`, [June. 2022].

[Red] Redis. `https://redis.io`, [March 2022].

[Riz12] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *Proceedings of USENIX Annual Technical Conference, USA*, pages 101–112, 2012.

[RKL20] Shahbaz Rezaei, Bryce Kroencke, and Xin Liu. Large-scale mobile app identification using deep learning. *IEEE Access*, 8:348–362, 2020.

[RYCW21] Wei Rang, Donglin Yang, Dazhao Cheng, and Yu Wang. Data life aware model updating strategy for stream-based online deep learning. *IEEE Transactions on Parallel Distributed Systems*, 32(10):2571–2581, 2021.

[SAH] R. Sommer, J. Amann, and S. Hall. Spicy: a unified deep packet inspection framework for safely dissecting all your data. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, (ACSAC'16), USA, 2016*, pages 558–569.

[SAR14] M. Omair Shafiq, Reda Alhajj, and Jon G. Rokne. Handling incomplete data using semantic logging based social network analysis hexagon for effective application monitoring and management. In *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM'14, China*, pages 634–641, 2014.

[SBB$^+$] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc. `https://research.google.com/archive/papers/dapper-2010-1.pdf`, [March 2022].

[SBS] Liran Sidki, Yehuda Ben-Shimol, and Akiva Sadovski. Fault tolerant mechanisms for SDN controllers. In *Proceedings of IEEE Conference on Network Function Virtualization and Software Defined Networks, (NFV-SDN), USA, 2016*, pages 173–178.

[SBZ+] Mona El Saadawy, Petar Basta, Yunjia Zheng, Bettina Kemme, and Mohamed Younis. Flow-based service type identification using deep learning. In *Proceedings of IEEE International Conference on Network Softwarization, (Netsoft'21), Japan, 2021.*

[SCP14] JaeSeung Song, Cristian Cadar, and Peter R. Pietzuch. Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, 2014.

[SDGK21] Sogand SadrHaghighi, Mahdi Dolati, Majid Ghaderi, and Ahmad Khonsari. Softtap: A software-defined TAP via switch-based traffic mirroring. In *Proceedings of 7th IEEE International Conference on Network Softwarization, (NetSoft'21), Japan*, pages 303–311, 2021.

[Sin] H Singh. Performance analysis of unsupervised machine learning techniques for network traffic classification. In *Proceedings of International Conference on Advanced Computing & Communication Technologies, USA, 2015*, pages 401–404.

[SKY] Mona El Saadawy, Bettina Kemme, and Mohamed Younis. Enabling efficient application monitoring in cloud data centers using SDN. In *Proceedings of IEEE International Conference on Communications, (ICC'20), Ireland, 2020*, pages 1–6.

[SMX+15] Meral Shirazipour, Heikki Mahkonen, Ming Xia, Ravi Manghirmalani, Attila Takács, and Veronica Sanchez Vega. A monitoring framework at layer4-7 granularity using network service headers. In *Proceedings of IEEE Conference on Network Function Virtualization and Software Defined Networks, (NFV-SDN'15), USA*, pages 54–60, 2015.

[Sol] Solarwinds. `https://www.solarwinds.com`, [March 2022].

[Spa] Apache spark. `https://cassandra.apache.org`, [March 2022].

[sys] Sysdig. `http://www.sysdig.org`, [March 2022].

[SYW16] Muhammad Shafiq, Xiangzhan Yu, and Dawei Wang. Network traffic classification techniques and comparative analysis using machine learning algorithms. In *Proceedings of 2nd IEEE International Conference on Computer and Communications, (ICCC)*, pages 2451–2455, 2016.

[SYWK22] M. El Saadawy, M. Younis, J. Wang, and B. Kemme. Dynamic application call graph formation and service identification in cloud data centers. *IEEE Transactions on Network and Service Management*, 2022.

[SZL$^+$12] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, Z. Zhang, and Z. Jia. Precise, scalable, and online request tracing for multitier services of black boxes. *IEEE Transactions on Parallel and Distributed Systems*, 23(6):1159–1167, 2012.

[TGB18] Ioannis Tsamardinos, Elissavet Greasidou, and Giorgos Borboudakis. Bootstrapping the out-of-sample predictions for efficient and accurate cross-validation. *Machine Learning*, 107(12):1895–1922, 2018.

[TP12] Arit Thammano and Patcharawadee Poolsamran. SMBO: A self-organizing model of marriage in honey-bee optimization. *Expert Systems with Applications*, 39(5):5576–5583, 2012.

[Tsh] Tshark. `www.wireshark.org/docs/man-pages/tshark.html`, [March 2022].

[vADK14] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *Proceedings of IEEE Network Operations and Management Symposium, (NOMS'14) , Poland*, pages 1–8, 2014.

[vK] Jóakim v. Kistowski. Http load generator for variable load intensities. `https://github.com/joakimkistowski/HTTP-Load-Generator`, [Sept-2021].

[vKES$^+$19] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *Proceedings of Software Engineering and Software Management, SE/SWM'19, Germany*, pages 99–100, 2019.

[Weaa] Weaveworks. Introducing weave scope. `https://www.weave.works/docs/scope/latest/introducing/`, [March 2022].

[Weab] Weaveworks. Sock shop: A microservices demo application. `https://microservices-demo.github.io`, [March 2022].

[WEB] Webvowl. `http://vowl.visualdataweb.org/webvowl.html`, [March 2022].

[WEG] Coburn Watson, Scott Emmons, and Brendan Gregg. A microscope on microservices. `https://netflixtechblog.com/a-microscope-on-microservices-923b906103f4`, [March 2022].

[WGH+15] A. Wang, Y. Guo, F. Hao, T. V. Lakshman, and S. Chen. UMON: flexible and fine grained traffic monitoring in open vSwitch. In *Proceedings of 11th ACM Conference on Emerging Networking Experiments and Technologies, (CoNEXT'15), Germany*, pages 15:1–15:7, 2015.

[WYKH15] Q. Wang, A. Yahyavi, B. Kemme, and W. He. I know what you did on your smartphone: Inferring app usage over encrypted data traffic. In *Proceedings of IEEE Conference on Communications and Network Security, (CNS)*, 2015.

[WZXG20] Tao Wang, Wenbo Zhang, Jiwei Xu, and Zeyu Gu. Workflow-aware automatic fault diagnosis for microservice-based applications with statistics. *IEEE Transactions on Network and Service Management*, 17(4):2350–2363, 2020.

[WZZ+17] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. Malware traffic classification using convolutional neural network for representation learning. In *Proceedings of International Conference on Information Networking, (ICOIN'17), Vietnam*, pages 712–717, 2017.

[XDP] Xdp. `https://developers.redhat.com/blog/2021/04/01/get-started-with-xdp`, [June 2022].

[Zip] Openzipkin. `https://zipkin.io`, [March 2022].

[ZJYP21] Jingjing Zhao, Xuyang Jing, Zheng Yan, and Witold Pedrycz. Network traffic classification for data fusion: A survey. *Information Fusion*, 72:22–47, 2021.

[ZKC+15] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert G. Greenberg, Guohan Lu, Ratul Mahajan, David A. Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. *Computer Communication Review*, 45(5):479–491, 2015.

[ZLWY19] Jielun Zhang, Fuhao Li, Hongyu Wu, and Feng Ye. Autonomous model update scheme for deep learning based network traffic classifiers. In *IEEE Global Communications Conference, (GLOBECOM)*, pages 1–6, 2019.

[ZWG+18] Zili Zha, An Wang, Yang Guo, Doug Montgomery, and Songqing Chen. Instrumenting open vSwitch with monitoring capabilities: Designs and challenges. In *Proceedings of the Symposium on SDN Research, (SOSR'18), Los Angeles, USA*, pages 16:1–16:7, 2018.

[ZWH16] Wei Zhang, Timothy Wood, and Jinho Hwang. Netkv: Scalable, self-managing, load balancing as a network function. In *Proceedings of IEEE International Conference on Autonomic Computing, (ICAC'16), Germany*, pages 5–14, 2016.

[ZXW+13] Jun Zhang, Yang Xiang, Yu Wang, Wanlei Zhou, Yong Xiang, and Yong Guan. Network traffic classification using correlation information. *IEEE Transactions on Parallel Distributed Systems*, 24(1):104–117, 2013.

[ZZZ+] X. Zhang, Y. Zhang, X. Zhao, G. Huang, and Q. Lin. Smartrelationship: a VM relationship detection framework for cloud management. In *Proceedings of 6th Asia-Pacific Symposium on Internetware, China, 2014*, pages 72–75.

Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
DB	Database
DLM	Deep Learning Model
DPDK	Data Plane Development Kit
DSL	Domain-specific Language
GRE	Generic Routing Encapsulation
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
JSP	Java Server Pages
JVM	Java Virtual Machine
MAC	Media Access Control
ML	Machine Learning
NFV	Network Function Virtualization
NLP	Natural Language Processing
OVS	Open vSwitch
P4	Programming Protocol-independent Packet Processors
PCAP	Packet CAPture
QoS	Quality of Service
SDN	Software Defined Networking
SLA	Service Level Agreement
TCP	Transmission Control Protocol
ToR	Top of Rack
UDP	User Datagram Protocol
UI	User Interface
URL	Uniform Resource Locator
VM	Virtual Machine
VXLAN	Virtual Extensible Local Area Network
YCSB	Yahoo! Cloud Serving Benchmark