Learning from Network Device Statistics

Rolf Stadler^{1,2} · Rafael Pasquini³ · Viktoria Fodor¹

Received: 22 July 2017 / Revised: 26 August 2017 / Accepted: 2 September 2017 /
Published online: 26 September 2017
© The Author(s) 2017. This article is an open access publication

Abstract We estimate end-to-end service metrics from network device statistics. Our approach is based upon statistical, supervised learning, whereby the mapping from device-level to service-level metrics is learned from observations, i.e., through monitoring the system. The approach enables end-to-end performance prediction without requiring an explicit model of the system, which is different from traditional engineering techniques that use stochastic modeling and simulation. The fact that end-to-end service metrics can be estimated from local network statistics with good accuracy in the scenarios we consider suggests that service-level properties are “encoded” in network-level statistics. We show that the set of network statistics needed for estimation can be reduced to a set of measurements along the network path between client and service backend, with little loss in estimation accuracy. The reported work is largely experimental and its results have been obtained through testbed measurements from a video streaming service and a KV store over an OpenFlow network .

Keywords Network management · Machine learning · Statistical learning · Feature selection · End-to-end performance Prediction · Network analytics · OpenFlow

✉ Rolf Stadler
stadler@kth.se

Rafael Pasquini
rafael.pasquini@ufu.br

Viktoria Fodor
vjfodor@kth.se

¹ ACCESS Linnaeus Center, KTH Royal Institute of Technology, Stockholm, Sweden

² Swedish Institute of Computer Science (RISE SICS), Kista, Sweden

³ Federal University of Uberlândia (FACOM/UFU), Uberlândia, Brazil

1 Introduction

Network statistics and measurements have been at the core of network management since its inception. How to model and retrieve this data from network devices in an open way has been a driving force behind the evolution of network management standards.

Network statistics have traditionally been used for managing the network layer and have driven tasks like network provisioning, routing, and fault detection. In this work, we use network-level, local statistics to estimate service-level, end-to-end metrics. We give examples and show scenarios where this can be achieved with an accuracy that matches methods which rely on both network and backend statistics.

We believe that our results are significant, for two reasons. First, they include mappings from low-level device statistics, such as packet rates, onto higher-level application statistics like frame rates or response times, which is hard to do with traditional network engineering techniques. Second, while application statistics highly depend on the configuration and the state of the backend system that runs the service [1, 2], we compute mappings without considering backend statistics. We thus conclude that end-to-end application statistics must be “encoded” in network device statistics in the scenarios we investigated.

Our approach to mapping network device statistics to service level-metrics is based upon statistical, supervised learning, whereby the mapping function is learned from observations, i.e., through monitoring the system.

Our approach enables end-to-end performance prediction without requiring an explicit model of the system. The method is different from the traditional engineering approach to performance prediction, which is based on stochastic modeling and simulation. Traditional engineering requires detailed knowledge of the architecture of the system and of its components, as well as their functionality and interactions. The reason we advocate statistical learning for end-to-end performance estimation is that a stochastic model would become too complex for many application scenarios and thus would be infeasible to develop and apply in practice.

The work reported in this paper is largely experimental, and its findings have been obtained from testbed measurements. We have deployed two services namely—an HTTP *Video-on-Demand* (VoD) service with VLC [3] clients, and Voldemort [4], a *Key-Value store* (KV)—on a cluster and measured service-level QoS metrics on clients which access these services over an OpenFlow network. We have built generators that load the testbed with service requests and cross traffic, and we have devised a framework for collecting device statistics from servers and switches during test runs. The collected traces are then used as input to train models that estimate service-level QoS metrics from infrastructure statistics.

The main contributions of this paper are:

- We demonstrate through experimentation that service-level metrics can be learned from network device statistics using standard machine-learning methods. In our case, the metrics are KPIs from two services, streaming video and KV store. The device statistics are low-level, course-grain metrics from

OpenFlow switches of the network that enables communication between a server cluster and a client;

- We show that the set of network statistics needed for estimation can be reduced using feature reduction techniques, which decreases the overhead both for data collection and model computation. It turns out that features (i.e., metrics) which score high with univariate feature reduction tend to lie on the network path between the server cluster that runs the service and the client for which the prediction is made.

Preliminary results of this work appeared in [5]. This paper includes a revised presentation of the problem and firmer results. It includes additional experiments involving cross-traffic. All model computation and evaluations reported in this paper have been performed with Python libraries instead of R, which was used in [5]. Further, we include a naïve learning method as a baseline to assess the effectiveness of the approach. Also, we apply an additional feature reduction method, which is computationally efficient and allows us to rank features.

A word on terms we will be using throughout the paper. We apply terminology from network management and machine learning. The expression “we estimate service metrics from device measurements” translates into a phrase like “we predict target variables from features” in machine learning. Therefore, with “we estimate a metric” we mean the same as “we predict a metric” in the machine-learning sense. (Note also that, in machine learning terminology, “prediction” normally does not refer to a future time.) Furthermore, we use the terms “application” and “service” in the same sense.

The remainder of this paper is organized as follows. Section 2 formulates the learning problem and discusses the machine learning methods used in this work. Section 3 describes the infrastructure statistics and the service-level metrics collected for QoS estimation, and it explains how traces are generated. Section 4 details the testbed and load generation. Section 5 describes the experiments, the model computation process, and the evaluation results. Section 6 provides an assessment of the experimental results. Section 7 surveys related work. Finally, Sect. 8 presents conclusions and future work.

2 End-to-End Estimation of Service Metrics as a Learning Problem

Figure 1 outlines the system under investigation. It is composed of a backend in form of a server cluster, a network, and a client population. Clients access the services running on the cluster through the network. The services we are considering in this work are streaming video and KV store. Statistics collected from the infrastructure are used to train models for end-to-end metrics estimation.

We are interested in how the infrastructure statistics X relate to the service-level metrics Y on the client side. The infrastructure statistics X include measurements from the network and from the server cluster. The performance indicators Y on the client side refer to service-level metrics, for example, frame rate and response time. Details regarding the composition of X and Y are given in Sect. 3.

Fig. 1 Learning service-level metrics Y from network and cluster statistics X

The metrics X and Y evolve over time, influenced, e.g., by the load on the servers, operating system dynamics, network traffic, number of active clients. Assuming a global clock that can be read on the machines in the server cluster, in the OpenFlow network controller, and in the clients, we model the evolution of the metrics X and Y as time series $\{X_t\}$, $\{Y_t\}$, and $\{(X_t, Y_t)\}$.

Our objective is to estimate the service-level metric Y_t at time t on a client, based on knowing the infrastructure statistics X_t . Using the framework of statistical learning, the problem is finding a model $M: X_t \rightarrow \hat{Y}_t$, such that $\hat{Y}_t$ closely approximates Y_t for a given X_t . This is a regression problem, which we solve through supervised learning [6].

We apply two machine learning methods in this work—regression tree and random forest. The regression tree method computes region boundaries with the objective of minimizing the residual sum of squares (RSS). It recursively partitions the space of the input statistics (the feature space) into regions $R_1, R_2, \dots, R_M$. For a given X , the metric Y is estimated as $\hat{Y} = \sum_{i \in R_k} \frac{Y_i}{|R_k|}$, where R_k is the region that X falls into, $|R_k|$ is the number of training samples in R_k , and i is the index of the samples in R_k . The regions are constructed using a greedy algorithm, whereby during each construction step of a selected region, a feature and a threshold are identified that fulfill the optimization criterion [7]. This method has a computational complexity of $O(m^2n)$, whereby m is the number of samples and n is the number of features.

Random forest is an ensemble method. Each estimated value of Y is an average of estimations from several regression trees [8]. Each of these trees is constructed using a fraction of the input statistics, and each construction step uses a randomized reduced feature set [7]. This method has a computational complexity of $O(Tm^2n)$, whereby T is the number of trees in the adopted forest.

As a baseline to the machine learning methods, we use a naïve method which relies on Y values only. For each $x \in X$ it predicts a constant value $\bar{y}$ which is simply the average of the samples y_t .

To investigate to which extent a reduced set of input statistics, automatically selected, can achieve accurate estimations, we apply feature selection techniques from machine learning. Computing a subset of features that minimizes the

estimation error requires the evaluation of 2^n subsets for n features and is thus infeasible for large n . For this reason, heuristic selection methods have been developed. One such method we use in this work is *forward-stepwise-selection* [7]. Starting from an empty feature set, the method incrementally grows the feature set by including, in each iteration, a new feature that minimizes the estimation error. The process stops whenever including an additional feature does not further decrease the estimation error. The method requires the evaluation of $O(n^2)$ subsets of the full feature set. The second selection method we use is *univariate-feature-selection*, which relies on computing the cross correlation between regressor and target for each feature, sorting the features according to the evaluation values, and selecting the top k features to obtain the reduced feature set. This method requires the evaluation of n subsets, each containing a single feature. It has lower computational complexity than *forward-stepwise-selection*. In this work, we refer to a resulting feature set from *forward-stepwise-selection* as the ‘minimal’ feature set.

3 Infrastructure Statistics and Service-Level Metrics

This section describes the statistics of the input feature set $X = X_{cluster} \cup X_{port}$ and its subsets. We refer to X also as the *full feature set*. Further, the section explains the specific service-level metrics Y_{VoD} and Y_{KV} and how traces for model computation are generated.

The $X_{cluster}$ feature set is extracted from the kernel of the Linux operating system that runs on the servers executing the applications. The Linux kernel is the core of the Linux operating system. It gives applications access to resources, such as CPU, memory, and network, and it schedules requests to those resources. To access the kernel data structures, we use System Activity Report (SAR), a popular open source Linux library [9]. Accessing kernel data through *procfs* [10], SAR computes various system statistics over a configurable interval. Examples of such statistics are CPU core utilization, memory utilization, and disk I/O. $X_{cluster}$ includes only numeric features from SAR, about 1700 statistics per server.

The X_{port} feature set is extracted from the OpenFlow switches at *per port* granularity level. It includes statistics from all switches in the network. We implemented a monitoring module in an OpenFlow controller, using standard OpenFlow *statistic request* and *statistic reply* messages [11] to periodically collect statistics regarding: (1) Total number of Bytes Transmitted per port; (2) Total number of Bytes Received per port; (3) Total number of Packets Transmitted per port; and (4) Total number of Packets Received per port.

The X_{path} feature set is a subset of X_{port} containing only statistics from ports on the path between the server cluster and the client. During the experiments, the path is composed of 12 ports, which results in a feature set of 48 statistics.

The Y_{VoD} service-level metrics: for the VoD application, we chose the VLC media player software [3], which provides single-representation streaming with varying frame rate. The service-level metrics we are considering are measured on the client device. During an experiment, we capture the following metrics: (1)

Display Frame Rate (frames/sec), i.e., the number of displayed video frames per second; (2) *Audio Buffer Rate (buffers/sec)*, i.e., the number of played audio buffers per second. These metrics are not directly measured, but computed from VLC events like the display of a video frame at the client's display unit. We have instrumented the VLC software to capture these events and log the metrics every second.

The Y_{KV} service-level metrics: for the KV store, we chose the Voldemort software [4]. The service-level metrics we are considering are measured on the client device. During an experiment, we capture the following metrics: (1) *Read Response Time* as the average read latency for obtaining responses over a set of operations performed per second; (2) *Write Response Time* as the average write latency for obtaining responses over a set of operations performed per second. These metrics are computed using a benchmark tool of Voldemort. The read and write operations follow the *request-and-reply* paradigm, which allows for tracking the latency of individual operations. We have instrumented the benchmark tool to log the metrics every second.

Generating the traces: the collected statistics for $X_{cluster}$ and X_{port} are stored in csv files. A csv file contains m rows of n features. Each row represents one observation and has a timestamp t indicating when the statistics were measured. Collected service-level metrics for Y_{VoD} and Y_{KV} are also stored in csv files together with the observation time t . During experiments, X and Y statistics are collected every second on the testbed. For each application running on the testbed, the data collection framework produces a time series $\{(X_t, Y_t)\}$. We interpret this time series as a set of samples $\{(X_1, Y_1), \dots, (X_m, Y_m)\}$. Assuming that each sample (X_t, Y_t) in the set is drawn uniformly at random from a joint distribution (X, Y) , we obtain models using methods from statistical learning.

4 The Testbed

The testbed is deployed on a server rack in our laboratory at KTH. It includes ten high-performance machines interconnected by Gigabit Ethernet. Nine of them are Dell PowerEdge R715 2U servers, each with 64 GB RAM, two 12-core AMD Opteron processors, a 500 GB hard disk, and four 1 GB network interfaces. The tenth machine is a Dell PowerEdge R630 2U machine with 256 GB RAM, two 12-core Intel Xeon E5-2680 processors, two 1.2 TB hard disks, and twelve 1 GB network interfaces. All machines run Ubuntu Server 14.04 64 bits, and their clocks are synchronized through NTP [12].

The VoD application is deployed on six PowerEdge R715 machines: one HTTP load balancer, three web server and transcoding machines, and two network file storage machines. The load balancer runs HAProxy version 1.4.24 [13]. Each web server and transcoding machine runs Apache version 2.4.7 [14] and ffmpeg version 0.8.16 [15]. The network file storage machines run GlusterFS version 3.5.2 [16] and are populated with the ten most-viewed YouTube videos in 2013, which have a

length of between 33 s and 5 min. The VoD client is deployed on another PowerEdge R715 machine and runs VLC [3] version 2.1.6 over HTTP.

The Voldemort KV store runs on the same machines as the VoD application. Six of them act as KV store nodes in a peer-to-peer fashion, running Voldemort version 1.10.22 [4]. The store is first populated with 10 million unique keys, selected uniformly at random from a 32-bit key-space. The size of the stored values is 1024 bytes, the default for Voldemort. Each key-value pair is stored on three machines in the cluster. Consistent hashing is used to identify these machines. The KV client runs the Voldemort benchmark tool and uses the same machine as the VoD client.

By deploying VoD and KV store on the same machines, the testbed allows for experiments with either a single application running or with both applications running concurrently. Additional details about the VoD and KV application setup and configuration can be found in [2, 17, 18].

4.1 Emulated OpenFlow Network

The OpenFlow network, including switches and controller, is virtualized on the PowerEdge R630 machine described above. We use Virtual Box as hypervisor. Each OpenFlow switch and the OpenFlow controller runs in a virtual machine (VM). VMs run Ubuntu 14.04 and have 10 GB disk space. The switch VMs have 1 core and 4 GB RAM; the controller VM has 4 cores and 8 GB RAM. We use 18 cores to run the VMs, out of 24 physical cores available on this machine. We monitor the *cpu steal time* in all the virtual machines to monitor the competition among the virtual machines for access to physical cores. In all experiments we conducted, the observed *cpu steal time* was zero, which means that each VM had access to an entire physical core during an experiment.

Figure 2 shows the configuration of the OpenFlow network deployed on the PowerEdge R630 machine. It represents a three-tier network with border switches (SWB1...SWB4), aggregation switches (SWA1...SWA6), and core switches (SWC1...SWC4).

An OpenFlow switch is emulated using *Open vSwitch 2.3.2* (OVS), an open-source software switch for virtualized server environments [19]. Such a switch can forward traffic between different VMs on the same physical machine over a physical network. *Open vSwitch* supports standard management interfaces (e.g. sFlow, NetFlow, CLI) and is open to programmatic extension and control through the OpenFlow protocol.

The links between OpenFlow switches in Fig. 2 are layer-2 ethernet links, emulated through the Virtual Box hypervisor and configured for 1 GB per second. We use the *netem* package to control the communication delay between the switches [20]. The links between the physical machine emulating the OpenFlow network on the one side and the server cluster, client machine, load generator machines, and cross traffic machines on the other side are physical.

The OpenFlow controller is implemented using the Floodlight 1.0 package [21], extended with the monitoring module we developed for collecting the X_{port} feature

Fig. 2 Connectivity of the OpenFlow network testbed and the location of client, load generators, cross-traffic generators, and application servers

set. The network topology has 14 switches with a total of 44 ports, which are periodically polled. The OpenFlow controller maintains a connection to each OpenFlow switch using the layer-2 ethernet links of Virtual Box, some of which appear in Fig. 2 as dashed lines.

Beside the OpenFlow network, Fig. 2 shows four other components of our testbed. First, the server cluster, which runs the VoD and KV applications. Second, the client component, which issues service requests, and for which the service-level metrics are estimated. Third, three load generators, which emulate client communities and generate requests from different network locations. Load generator 1 runs on one PowerEdge R715 machine, load generators 2 and 3 run in VMs and share another PowerEdge R715. These two VMs are each configured with 4 cores, 8 GB RAM, and 10 GB disk space. Both applications have their own client and the load generator as described in Sect. 4.2. Fourth, two VMs are dedicated to cross traffic generation. Both run on the same machine as load generators 2 and 3 and share their configuration.

The routing topology is set up in such a way that different levels of traffic aggregation occur during experiments. All routes are bidirectional. The traffic between client and load generator 1 on the one side and the server cluster on the other side follows the path SWB3, SWA5, SWC4, SWC1, SWA1, and SWB1; the traffic between load generator 2 and the server cluster follows the path SWB2, SWA3, SWC2, SWA2, and SWB1; the traffic between the load generator 3 and the server cluster follows the path SWB4, SWA6, SWC1, SWA1, and SWB1; and the traffic between the cross traffic load generator and the cross traffic KV server (see

below) follows the path SWB4, SWA6, SWC1, SWA1, SWB1, SWA2, SWC2, SWA3, and SWB2.

4.2 Generating Load Patterns

We built two types of load generators, one for the VoD application and another for the KV store. The VoD load generator dynamically controls the number of active VoD sessions, spawning and terminating VLC clients. The KV load generator controls the rate of KV operations issued per second. Both generators produce load according to two distinct load patterns.

The Periodic-load pattern: The load generator produces requests following a Poisson process whose arrival rate is modulated by a sinusoidal function, with start value P_S , amplitude P_A , and a period of 60 min.

Flashcrowd-load pattern: The load generator produces requests following a Poisson process whose arrival rate is controlled by the flashcrowd model described in [22]. The arrival rate starts at value F_S and peaks at flash events. F_E such events occur within an hour, distributed uniformly at random over this time period. At each flash event, the arrival rate increases within a minute to a peak value of F_R , stays at this level for one minute, and then decreases to the initial rate within four minutes.

Table 1 gives the configurations of the load generators for the experiments reported in Sect. 5.

To produce cross traffic, we use an instance of the KV load generator, together with a single KV server which runs in a VM outside the sever cluster (see Fig. 2). This KV server is populated with 100K unique keys, selected uniformly at random from a 32-bit key space. The size of the stored values is 1024 bytes, the default for Voldemort. We use the flashcrowd-load pattern for generating cross traffic.

Table 2 shows the configurations of the cross-traffic load generator for the experiments reported in Sect. 5. The cross traffic is produced by a load generator attached to SWB4 (see Fig. 2). As explained above, it is realized as a KV request generator that sends a request stream towards a KV server that is attached to SWB2. The responses of the KV server are routed on the same path as the request stream, but in reverse direction. The KV load generator with parameters (200, 10, 500) produces a highly dynamic traffic pattern on the network. The same generator with parameters (500, 10, 1000) creates a low-varying, almost constant traffic pattern. The reason for that is that the higher level of request load saturates the KV server.

Table 1 Configuration parameters of VoD and KV load generators

Application	Load generator	Periodic load		Flashcrowd load		
		P_S	P_A	F_S	F_E	F_R
VoD	1	100	50	50	10	150
	2, 3	80	40	20	3	120
KV	1	1000	800	200	10	1800
	2, 3	350	150	200	3	500

Table 2 Configuration parameters of KV cross-traffic load generator

Cross traffic type	Flashcrowd load		
	F_S	F_E	F_R
Dynamic	200	10	500
Constant	500	10	1000

Note that the two components, KV request generator and KV server, run on machines outside the server cluster and are independent of the services running on the cluster. The sole purpose of these two components is to produce interfering network traffic.

Figure 3 shows the utilization of the link from switch SWC1 towards switch SWA1 with and without cross traffic. The link carries VoD requests from the client and from the load generators 1 and 3 towards the server cluster. Figure 3a shows link utilization under dynamic cross traffic, Fig. 3b shows link utilization under constant cross traffic.

5 Experiments, Model Computation and Evaluation Results

The experiments on the platform produce trace files with structure $\{(X_t, Y_t)\}$ (see Sect. 3). Using concepts and methods from statistical learning, we train and evaluate the model M that fits a particular trace. We apply the well-known validation-set approach: we (1) randomly assign each sample (X_t, Y_t) of a trace to either a training set or a test set; (2) compute the model from the training set; and (3) evaluate it using the test set [7]. Following standard practice, the training set contains 70% of the samples, and the test set 30%.

We compute two metrics to evaluate the learned models. The first metric is the *Normalized Mean Absolute Error (NMAE)*, which is computed as $\frac{1}{\bar{y}}(\frac{1}{m} \sum_{i=1}^m |y_i - \hat{y}|)$, where $\hat{y}$ is the model estimation for the measured service-level metric y , and $\bar{y}$ is the mean of the samples y_i . m is the size of the test set. The second

Fig. 3 Traffic through output port of SWC1 towards SWA1 (see Fig. 2); **a** VoD traffic without cross traffic and VoD traffic with dynamic cross traffic; **b** VoD traffic without cross traffic and VoD traffic with constant cross traffic

metric is the *training time*, which measures the time it takes to train a model on the training set. The models are computed on a PowerEdge R630 machine, using Python 2.7.6 and scikit-learn version 0.18.1, specifically the libraries RandomForestRegressor (with 120 trees) and DecisionTreeRegressor.

All results bellow are from experiments with a running time of 12 h, which resulted in 12×3600 samples per run. Figure 4 shows two time windows of 4000 s each from such runs. The blue points indicate measurements, the red points model estimates. (Some points have been omitted for better visibility.)

In the following, we explain how an experimental run using the VoD application is performed on the testbed. At the beginning of a VoD experiment, the VLC client sends a *VoD session request* for playing a video to the HTTP load balancer machine. After the video has played, the VLC client sends a new request for another video to the HTTP load balancer. Also, at the beginning of the run, the load generators start sending VoD session requests to the HTTP load balancer, according to the specified load pattern. Receiving a VoD session request from a client or a load generator, the HTTP load balancer forwards the request to the backend web server that has the least number of pending connections with the HTTP load balancer. After receiving a request, the web server spawns a transcoding instance for a selected video, whereby the raw video content is retrieved over the network from one of the network file storage machines.

At the beginning of an experiment with the KV application, the KV client starts sending requests, at the rate of 100 per second, to the server cluster, using randomly selected keys from the set of 10 million keys (see Sect. 4). At the same time, the KV load generators start sending requests to the server cluster at a varying rate, according to the specified load pattern. The client and load generators produce a ratio of 80% read to 20% write requests following a Bernoulli process.

For all the experiments, the communication delays in the emulated OpenFlow network are controlled as follows. The *netem* package is configured to introduce a delay per interface that is normally distributed with an average of 4 ms and a variance of 0.1 ms. Therefore, the end-to-end round-trip delay between client and server cluster is 50 ms on average. The end-to-end round-trip delay between the cross-traffic load generator and the cross-traffic KV server is 75 ms on average.

Fig. 4 Measurements and estimates from testbed experiments. **a** Display Frame Rate for the VoD service under periodic load pattern. **b** Read Response Time for the KV service under periodic load pattern

From the extensive set of experiments we performed on the testbed, we select eight which illustrate our key findings. The results of these experiments are described in the following subsections. Two experimental runs were performed using only the VoD application on the testbed, driven by the periodic and flashcrowd load patterns; two runs were performed using only the KV application, for both load patterns; two runs were performed using both the VoD and the KV applications running concurrently and independently on the testbed, for both load patterns; and, finally, two runs were performed using the VoD application under periodic load, together with cross traffic with both dynamic and constant load conditions. The traces of these eight experiments are available at a public data repository [23].

5.1 Estimating Service-Level Metrics Using the Full Feature Set X

We perform model computations using the traces collected during the eight experimental runs described above. All computations in this section are based on the full feature set $X = X_{cluster} \cup X_{port}$ (see Sect. 3). Table 3 shows the evaluation results for the VoD application, Table 4 shows the evaluation results for the KV application, and Table 5 shows the evaluation results for the VoD application under cross traffic. The results displayed in these tables are consistent with measurement results from our earlier work that is based exclusively on $X_{cluster}$ statistics [1, 2, 18].

Based on the results in Tables 3, 4 and 5, we make the following observations. First, as expected, the *random forest method* consistently outperforms regression tree, in terms of estimation accuracy for service-level metrics, across both load patterns and both applications. This is because random forest is an ensemble method that averages over a large set of regression trees. The *regression tree method* thus allows for faster model computation than random forest. Even though the computation for random forest includes 120 computations of single trees in our configuration, the RandomForestRegressor library from scikit-learn takes advantage

Table 3 Estimation error and training time for VoD application using the full feature set X for model computation

Load pattern	Exp. type	Regression method	Display frame rate		Audio buffer rate	
			NMAE (%)	Training time (s)	NMAE (%)	Training time (s)
Periodic	Single App.	Regr. Tree	9.30	78	27.6	108
		Rand. Forest	9.17	281	20.6	334
	Both Apps.	Regr. Tree	12.5	57	37.4	75
		Rand. Forest	12.2	178	29.8	211
Flash crowd	Single App.	Regr. Tree	8.63	81	24.8	83
		Rand. Forest	8.74	279	20.4	285
	Both Apps.	Regr. Tree	7.56	69	23.0	95
		Rand. Forest	7.34	214	18.3	297

Table 4 Estimation error and training time for KV application using the full feature set X for model computation

Load pattern	Exp. type	Regression method	Read Resp. time		Write Resp. Time	
			NMAE (%)	Training time (s)	NMAE (%)	Training time (s)
Periodic	Single App.	Reg. Tree	2.99	30	3.09	31
		Rand. Forest	2.07	89	2.19	96
	Both Apps.	Regr. Tree	5.23	39	5.33	44
		Rand. Forest	3.76	113	3.73	120
Flash crowd	Single App.	Regr. Tree	2.57	25	2.92	25
		Rand. Forest	1.78	70	2.00	66
	Both Apps.	Regr. Tree	3.94	33	3.93	34
		Rand. Forest	2.75	94	2.69	98

Table 5 Estimation error and training time for VoD application under periodic-load and cross traffic using the full feature set X for model computation

Cross traffic condition	Regression method	Display frame rate		Audio buffer rate	
		NMAE (%)	Training time (s)	NMAE (%)	Training time (s)
Dynamic	Regr. Tree	10.6	71	31.9	96
	Rand. Forest	10.6	224	25.0	271
Constant	Regr. Tree	7.59	88	22.4	111
	Rand. Forest	7.74	288	17.3	341

of the multi-core platform of the PowerEdge R630 machine and allows for an almost ideal speedup. This explains why the model computation time for random forest is only three times longer than the time for regression tree.

Second, running a *single application* allows for more accurate estimation, across regression methods and load patterns, compared with running both applications concurrently on the testbed. We explain this by the fact that the models for both applications share many aggregate features, for instance, aggregate CPU utilization, aggregate memory utilization, and aggregate OpenFlow statistics. The implication is that the dynamics of one application influences many features used for the model computation of the second application, which we can interpret as a source of noise, since both applications run independently. We observe generally an increase in estimation error when running both applications, compared to running a single application. This increase depends on the service-level metric and the load pattern, and it can reach 10% NMAE.

Third, service-level metric estimations for the KV application tend to be significantly more accurate than for the VoD application. Two factors seem to cause this difference. Regarding architecture and functionality, the KV application is much less complex than the VoD application, and estimation models for KV

application should thus be easier to learn. Furthermore, the Display Frame Rate and Audio Buffer Rate follow multi-modal distributions, while the Read and Write Response Times exhibit unimodal distributions. Figure 5 shows the distribution of the service-level metrics for runs with single applications on the testbed, driven by a periodic load pattern. Multi-modal distributions tend to have larger NMAE values than unimodal distributions.

Fourth, running VoD under the periodic-load pattern and cross traffic shows that the error increases slightly under dynamic cross traffic (1.3% NMAE), while it decreases somewhat under constant cross traffic (1.5% NMAE).

Overall, these experiments give evidence that it is possible, in many cases, to accurately estimate service-level metrics end-to-end, over a networking infrastructure, even under cross traffic.

5.2 Estimating Service-Level Metrics Using the Network Feature Set X_{port}

The purpose of this evaluation is to determine the estimation accuracy of the model computed from the X_{port} feature set (see Sect. 3), which includes only network statistics and has 176 features from 44 ports. The network enables the communication between the server cluster on the one side and the client and load generators on the other side, for the VoD and KV applications. In some experiments, it also carries cross traffic.

Tables 6 and 7 show the evaluation results for the scenarios described at the beginning of this section. To highlight the main results, we include in the tables only figures for the Display Frame Rate and Read Response Time, and for the random

Fig. 5 Histograms of service-level metrics for runs of VoD application (a,b) and KV application (c,d) under the periodic load pattern

Table 6 Estimation error and training time for display frame rate and read response time using the X_{port} feature set, which includes all network statistics

Load pattern	Experiment type	Display frame rate		Read Resp. time	
		NMAE (%)	Training time (s)	NMAE (%)	Training time (s)
Periodic	Single App.	10.1	11	2.40	11
	Both Apps.	13.5	8	4.46	10
Flashcrowd	Single App.	9.31	10	2.09	8
	Both Apps.	8.66	10	2.98	8

Experiments are without cross traffic; model computation is performed using random forest

forest method. The other measurements are consistent with those reported in Sect. 5.1.

Compared to the results from the full feature set X (see Tables 3, 4 and 5), the estimation error increases slightly, between 0.2 and 2% NMAE. The training time is much shorter and reduced by an order of magnitude. Under cross traffic, the error for the X_{port} feature set is slightly higher than the error for the full feature set X , the increase is similar to the figures without cross traffic. We observe the same qualitative behavior as for the full feature set X : the estimation accuracy gets slightly worse under dynamic cross-traffic load and slightly better under constant load.

We draw the following conclusions from these experiments. First, we expect that the estimation accuracy decreases when we compare the results from using the full feature set X , including cluster and network features, with the results from using the network features X_{port} only. What is surprising to us is that the accuracy from using the network features is very close to that of the full feature set X . In other words, it seems possible to learn end-to-end service-level metrics from network features alone.

Second, we see a significant reduction of model computation time by factor of more than 20 when comparing the time for the full feature set X to that of the network feature set X_{port} . This can be expected since X has more than 10K features, X_{port} less than 200. Therefore, not only can we limit the features we need to network statistics, we also achieve a significant reduction in computational overhead.

5.3 Comparing Feature Reduction Techniques to Reduce the Network Feature Set X_{port}

The purpose of this evaluation is to determine whether we can further reduce the size of the network feature set X_{port} while maintaining similar levels of estimation accuracy. We apply two known feature selection methods on the traces from all eight reported experiments. The two methods are *forward-stepwise-selection*, which considers subsets of the network feature set, and *univariate-feature-selection*, which considers single features of the network feature set (see Sect. 2). In the following,

Table 7 Estimation error and training time for display frame rate of VoD application under cross traffic using the X_{port} feature set

Cross traffic condition	Display frame rate	
	NMAE (%)	Training time (s)
Dynamic	11.2	11
Constant	8.52	11

The model computation is performed using random forest

we first discuss results obtained with *forward-stepwise-selection*, followed by results with *univariate-feature-selection*.

Forward-stepwise-selection takes as input a feature set and produces a ‘minimal’ subset. For our traces these subsets have less than ten features, while the network feature set has 176 features. It turns out that these ‘minimal’ subsets are specific to the scenario and to the service metric.

Tables 8 and 9 show the evaluation results of the models computed with the subsets from *forward-stepwise-selection*. When comparing the results with those in Tables 6 and 7, we find that the difference in estimation error varies between 1.6 and 3.2% NMAE for Display Frame Rate and the difference is almost zero for Read Response Time. As expected, the training times for the ‘minimal’ feature sets are significantly smaller than those for the network feature set X_{port} . This time reduction has to be weighted against the cost of computing the ‘minimal’ subsets. For random forest, a ‘minimal’ subset computation takes between 900 and 1800 s.

The second method of feature selection is *univariate-feature-selection*. We apply the method to all network features, which gives a score to each feature and produces a ranked list of features. The top k features form the subset for model computation.

In order to compare the effectiveness of the two feature reduction methods, we compute the accuracy of model estimation for subsets of the same size. This means that k is below ten. Tables 10 and 11 show the results for the *univariate-feature-selection*.

Table 8 Estimation error and training time for display frame rate and read response time using the ‘minimal’ subsets created by *forward-stepwise-selection*

Load pattern	Experiment type	Display frame rate		Read Resp. time	
		NMAE (%)	Training time (s)	NMAE (%)	Training time (s)
Periodic	Single App.	12.7	1.2e-1	2.47	2.4e-2
	Both Apps.	16.7	1.6e-1	4.99	4.8e-2
Flashcrowd	Single App.	10.9	1.1e-1	2.18	2.1e-2
	Both Apps.	12.3	1.2e-1	3.56	3.5e-2

Experiments are without cross traffic; model computation method is random forest

Table 9 Estimation error and training time for display frame rate using the ‘minimal’ subsets created by *forward-stepwise-selection*

Cross traffic condition	Display frame rate	
	NMAE (%)	Training time (s)
Dynamic	13.4	1.3e-1
Constant	10.6	1.0e-1

Experiments under cross traffic; model computation method is random forest

Table 10 Estimation error and training time for display frame rate and read response time using the ‘minimal’ subsets created by *univariate-feature-selection*

Load pattern	Experiment type	Display frame rate		Read Resp. time	
		NMAE (%)	Training time (s)	NMAE (%)	Training time (s)
Periodic	Single App.	11.1	1.0e-1	2.62	2.0e-2
	Both Apps.	15.6	1.7e-1	5.03	5.0e-2
Flashcrowd	Single App.	11.0	1.1e-1	2.14	2.3e-2
	Both Apps.	12.9	1.4e-1	3.77	3.1e-2

Experiments are without cross traffic; model computation method is random forest

Comparing the two feature reduction techniques, we observe the following. Regarding estimation error, both methods produce similar accuracy levels, *univariate-feature-selection* performs slightly better. When it comes to model training time, both techniques use feature sets of the same size and the same number of measurements; hence the training time is practically the same. However, the cost of computing the ‘minimal’ subsets differs widely: *univariate-feature-selection* takes around 10 s per subset, while *forward-stepwise-selection* takes 900–1800 s, two orders of magnitude longer.

From these experiments we conclude that *univariate-feature-selection* outperforms *forward-stepwise-selection* for our purpose, and therefore *univariate-feature-selection* is our method of choice.

5.4 Reducing the Network Feature Set X_{port} Using *Univariate-Feature-Selection*

The purpose of this evaluation is to assess the performance of *univariate-feature-selection* for subsets of different sizes of the network feature set X_{port} .

Figures 6 and 7 show the evaluation results of the models computed with *univariate-feature-selection* for different subsets of size k . A subset of size k includes the top k features. The application running for the experiments is VoD, the service-level metric is Display Frame Rate, and the model computation method is random forest.

Figure 6 shows the assessment for experiments without cross traffic. As expected, the estimation error decreases with increasing size of the subset k . For

Table 11 Estimation error and training time for display frame rate using the ‘minimal’ subsets created by *univariate-feature-selection*

Cross traffic condition	Display frame rate	
	NMAE (%)	Training time (s)
Dynamic	14.4	1.5e-1
Constant	10.0	1.1e-1

Experiments under cross traffic; model computation method is random forest

Fig. 6 Estimation Error versus Feature Rank k . k indicates the size of the feature set, which comprises the top k features. Application is VoD; service metric is Display Frame Rate; experiments are without cross traffic; model computation uses random forest

$k = 176$ the result is that of the network feature set X_{port} given in Table 6. For k larger than 100, the gain in accuracy becomes minimal. Consistent with earlier observations, single application scenarios allow for more accurate estimation than two application scenarios, and scenarios driven by flashcrowd load give better estimations than those driven by periodic load.

Figure 7 shows the effect of cross traffic on VoD under periodic load pattern for Display Frame Rate. Again, the minimal k for the best NMAE values are around 100. Consistent with other results reported before, dynamic cross traffic seems to increase the estimation error, while constant cross traffic seems to decrease the error.

The curves in Figs. 6 and 7 show steps with sizes as large as 2% NMAE. We speculate that this is due to high correlation among features with similar rank, a hypothesis needs further investigation. In any case, the curves fall monotonically with increasing k , as expected.

The curves reveal the minimum number of features needed for the most accurate estimation in a particular scenario, for a given service-level metric and learning method. As a consequence, we conclude that the network feature set X_{port} can be further reduced without losing accuracy. Unfortunately, as pointed out earlier, the minimum number is scenario and metric dependent.

We did the same evaluation as above for the KV application, with very similar results. One difference is that the step sizes of the curves tend to be smaller, another is that the minimal number of features k needed for accurate estimation is smaller as well.

Fig. 7 Estimation Error versus Feature Rank k . k indicates the size of the feature set, which comprises the top k features. Application is VoD; service metric is Display Frame Rate; experiments are conducted under cross traffic; model computation uses random forest

In the context of feature selection, it is important to understand at which part of the network topology the top ranked features can be found. A close inspection shows that in scenarios where the response times for KV is estimated the top 48 features all belong to switch ports along the path between the cluster and the client for which the estimation is performed. (Recall from Fig. 2 that the traffic between the client and the cluster traverses 12 ports, each of them having four features.) In the case of VoD, averaged over all scenarios we investigated, more than 50% of the top-ranked features lie on the path.

This observation motivates our approach to consider the features along the path for model computation. In other words, we reduce the feature set X_{port} with 176 features to the feature set X_{path} with 48 features, a reduction by 75%.

5.5 Estimating Service-Level Metrics Using the X_{path} Feature Set

The purpose of this evaluation is to determine the estimation accuracy of the model using the network features along the path between the server cluster and the client (see Fig. 2).

Table 12 shows the evaluations results of VoD and KV scenarios without cross traffic, Table 13 shows the results with cross traffic. Compared to the findings with the network feature set X_{port} (Tables 6 and 7), the estimation accuracy is similar and the increase in the NMAE is around 1% (at most 2%) for the Display Frame Rate. For the Read Response Time the increase is much smaller. In addition, the training time is cut in half.

The question is whether the set of features along the path can be further reduced without sacrificing accuracy. Figure 8 provides the answer. We apply *univariate-feature-selection* on the feature set X_{path} and create 48 subsets containing between 1 and 48 features. The figure shows the estimation accuracy versus the feature rank k for features along the path with respect to the VoD service and Display Frame Rate. The experiments are without cross traffic and the evaluation method used is random forest.

The figure suggests that the feature set X_{path} can be cut in half without losing accuracy and that *univariate-feature-selection* is an effective method to do that. However, as pointed out before, the minimal rank k as well as the identity of the

Table 12 Estimation error and training time for display frame rate and read response time using the 48 features along the path between server cluster and client

Load pattern	Experiment type	Display frame rate		Read Resp. time	
		NMAE (%)	Training time (s)	NMAE (%)	Training time (s)
Periodic	Single App.	10.4	6	2.45	6
	Both Apps.	14.5	4	4.46	5
Flashcrowd	Single App.	10.0	5	2.04	4
	Both Apps.	10.3	4	3.19	4

Experiment are conducted without cross traffic

Table 13 Estimation error and training time for display frame rate of VoD application under cross traffic, using the 48 features along the path between server cluster and client

Cross traffic condition	Display frame rate	
	NMAE (%)	Training time (s)
Dynamic	13.1	5
Constant	9.81	5

features are dependent on the scenario, the service-level metric, and the model computation method.

Figure 8 also shows the curve for network feature set X_{port} . This curve corresponds to the segment $k = 1.48$ of the corresponding curve in Fig. 6 and confirms that selecting features along the path is an important step in reducing the feature set.

6 Assessment of Evaluation Results

In this section, we review some key measurement results and draw conclusions. We first assess the effectiveness of the applied learning methods against a naïve method

Fig. 8 Estimation error versus feature rank k . k indicates the size of the feature set, which comprises the top k features on the path between server cluster and client. Application is VoD, service metric is Display Frame Rate, experiments without cross traffic, the model computation method is random forest

Table 14 Estimation error using the naïve method-VoD without cross traffic

Load pattern	Experiment type	Display frame rate NMAE (%)	Audio buffer rate NMAE (%)
Periodic	Single App.	13.2	29.2
	Both Apps.	18.4	43.0
Flashcrowd	Single App.	14.4	32.8
	Both Apps.	14.1	32.1

Table 15 Estimation error using the naïve method-KV without cross traffic

Load pattern	Experiment type	Read resp. time NMAE (%)	Write resp. time NMAE (%)
Periodic	Single App.	4.34	4.51
	Both Apps.	6.39	6.30
Flashcrowd	Single App.	3.95	4.15
	Both Apps.	7.14	6.88

Table 16 Estimation error using the naïve method-VoD under cross traffic

Cross traffic Condition	Display frame rate NMAE (%)	Audio buffer rate NMAE (%)
Dynamic	16.1	36.1
Constant	11.0	23.0

that predicts $\hat{Y}$ values as the sample mean of the Y values in the training set (see Sect. 2). This method thus predicts the same $\hat{Y}$ value for all possible X values.

Tables 14, 15 and 16 show the results of the naïve method for all experiments reported in Sect. 5. As expected, the naïve method produces estimations with a larger error than what we achieve with the decision tree or random forest methods.

In order to focus the discussion, we restrict ourselves to the case of the VoD application, the estimation of Display Frame Rate, and random forest for model computation. Table 17 provides a summary of the measurement results presented in Sect. 5 and allows the comparison with the naïve method.

Comparing random forest with the naïve method gives a difference of 3.2–6.8% NMAE for the full feature set X . For the feature set X_{path} , the difference varies from 1.2 to 4.4% NMAE. As mentioned above, random forest estimations are always better than estimation with the naïve method. The relative superiority of the random forest method varies though. In the case of running both applications simultaneously under the flashcrowd-load pattern, the difference is 6.8% NMAE (for the X feature set), which is half of the error of the naïve method. On the other hand, when running VoD under constant cross traffic, the difference is 1.2% NMAE (for the X_{path} feature set), which reduces the error of the naïve method by a factor of 0.1 only. We conclude that our method significantly outperforms the naïve method in most cases.

Table 17 Summary results: estimation error of display frame rate for VoD application; learning methods are random forest and naïve estimator

Load pattern	ClusterApp	Cross traffic	X NMAE (%)	X_{port} NMAE (%)	X_{path} NMAE (%)	Naïve NMAE (%)
Periodic	VoD	no	9.17	10.1	10.4	13.2
	VoD	Dynamic	10.6	11.2	13.1	16.1
	VoD	Constant	7.74	8.52	9.81	11.0
	VoD, KV	no	12.2	13.5	14.5	18.4
Flash crowd	VoD	no	8.74	9.31	10.0	14.4
	VoD, KV	no	7.34	8.66	10.3	14.1

Fig. 9 Difference in accuracy between naïve estimation method and random forest for different feature sets. The figure uses data from Table 17

Figure 9 uses data from Table 17 and illustrates the difference in NMAE between the learning method we used and the naïve method for all the reported scenarios running the VoD application on our testbed. We observe that the difference in accuracy monotonically decreases when the models used for estimation are based on X , X_{port} , and X_{path} , respectively. One can see that, in case of cross traffic, the estimation based on path features only is clearly less accurate than that relying on features from the entire network.

The results in Table 17 show that reducing a feature set comes at the expense of estimation accuracy. However, the increase in estimation error is small and still supports our claim that service-level metrics can be estimated from network statistics alone. As demonstrated in Sect. 5.5, the feature set X_{path} can be further reduced using *univariate-feature-selection* without losing accuracy.

7 Related Work

Research close to this paper is presented in [24], where the authors discuss an approach for learning from a set of network-level metrics, including delay, loss, and jitter measurements, in order to estimate QoS metrics for IPTV streaming clients.

The authors conclude that their estimation method is accurate, as long as the packet loss ratio is low. In order to produce the statistics, they instrument their application on both sides and inject probes in the network, which add extra traffic and are affected by packet loss. Relying on local (network) statistics only, our method is service independent and does not rely on active probing. If packet loss statistics were available at OpenFlow switches, our method would use them as input features for model computation.

While the features we adopt for model computation are service independent, other works propose methods that are engineered for a specific service and metric. Examples in the context of cloud and statistical learning are [25–28]. Other works like [29–32] use statistical learning models to estimate quality-of-experience metrics of a multimedia service. The authors in [33] describe a method that dynamically allocates run-time resources for MapReduce tasks under unbalanced data distribution. In particular, the work applies linear regression to predict partition sizes for reduce tasks and uses the predictions to guide run-time resource allocation.

Research that applies machine learning concepts in the context of OpenFlow networks is limited and often focuses on traffic classification. In [34] the authors present an architecture to collect traffic data from OpenFlow switches and use the collected data to classify such traffic as belonging to certain well-known applications. In [35] the author uses machine learning and OpenFlow in the design and implementation of a traffic classification system that accurately classifies traffic without affecting the latency or bandwidth in the data plane. In [36] the authors use machine learning algorithms to predict potential target host attacks based on historical network attack data for SDNs. In [37] the authors describe machine learning techniques to handle intrusions and Distributed Denial of Service (DDoS) attacks in SDNs. In [38] the author discusses different anomaly detection mechanisms for SDNs.

In the context of sensor networks, [39] predicts network-level metrics for network paths and performs feature selection in order to reduce the communication overhead when collecting node and link statistics.

8 Conclusions and Future Work

A key aspect of this work has been to reduce the feature set of infrastructure statistics that we use to estimate service-level metrics. We started out with the full feature set X of infrastructure statistics, which include device statistics from the server cluster and the network. We then reduced X to the set of network features X_{port} and, finally, to the set of network features along the path between server cluster and client, namely X_{path} . From the point of view of machine learning, the reason for feature reduction is to shorten the model computation time and to decrease the number of observations needed for a statistically accurate model. From the point of view of network management, reducing the feature set as described above means (a) estimating service-level metrics based on network device statistics only and

(b) further reducing the monitoring overhead by collecting measurements only along a network path.

We demonstrated through experimentation that our method of estimating service-level metrics using statistical learning is effective for two different applications, across two different load patterns and for the case of interfering cross traffic. The achieved accuracy differs between the scenarios, as Table 17 shows. However, across all scenarios we investigated, a clear pattern has emerged. For instance, predicting Read Response Times consistently achieves higher accuracy than predicting Display Frame Rates. The same is true for predicting metrics for a scenario involving a flashcrowd-load pattern versus one involving a periodic-load pattern, for a scenario with a single application versus one with two concurrent applications, for a prediction using random forest versus one using decision tree, and, finally, for a prediction using either of those two learning methods versus one using the naïve method.

The network in this work has been emulated, and its results must thus be confirmed in a real network environment with OpenFlow hardware switches. In addition, the question arises whether our approach is suitable for prediction in a virtualized network or an environment built with NVF (Network Function Virtualization) technology. We believe this to be the case if streams of statistics from lower layers of such systems can be extracted and processed. Our believe is based partly on the results from a study that investigated the influence of a virtualization layer between operating system and application on the mapping from operating-systems statistics to application-layer metrics [40]. The study used Docker containers and measured metrics prediction on a server cluster. It found a minor degradation in prediction accuracy compared to a bare-metal environment without a virtualization layer. Interestingly, it showed that if generic container metrics are included in the feature set, the degradation becomes negligible.

For our study, we selected on purpose two applications with very different characteristics regarding service logic, system architecture, and resource requirements for serving requests. Since the features (i.e., device statistics) we are using for prediction are not application-specific, our method is, in principle, applicable to a wide range of applications and network services. For instance, we chose for our experiments the VLC media player software, which provides single-representation streaming with varying frame rate. More advanced, rate-adaptive streaming solutions (e.g., DASH [41]) can be considered similarly, in which case predicted metrics will relate to the level of video quality, such as frame resolution or bitrate.

Statistical learning has shown to be an effective approach to estimating quality-of-experience metrics of multimedia services (see e.g., [29, 30, 42]). We anticipate that the path taken in this paper is applicable to quality of experience and that such metrics can be learned from network device statistics. So far, we have restricted ourselves to application-level QoS metrics that can be measured in a straightforward manner. Extending this work towards quality of experience is an option worth pursuing, given the emphasis of many technologies on improving user satisfaction and experience.

We foresee our future work along several directions. We plan to develop adaptive learning methods that allow for model updates as new measurements become

available and thus are suited for real-time, adaptive estimation. One of the challenges will be to dynamically adapt the feature set, i.e., the set of monitored network ports, to achieve a target level of accuracy with minimal overhead. Further, we see the need for a more fundamental understanding of the best methods and the achievable accuracy for predicting high-level end-to-end metrics from low-level local statistics. Questions that arise in the context of this paper relate to the type of service-level metrics that can be accurately predicted; the location of the data that must be collected and the collection rate; the strategies for mitigating the influence of cross traffic and other services that share the infrastructure on the prediction process; and the opportunities provided by new networking paradigms like softwarization and in-network computation for building a real-time learning infrastructure.

Acknowledgements This research has been partially supported by the Brazilian National Council for Scientific and Technological Development, CNPq, through the Science Without Borders Program, under the Grant 232586/2014-4; by the Swedish Governmental Agency for Innovation Systems, VINNOVA, through project SENDATE-EXTEND; and by the Swedish Research Council through the ACCESS Linnaeus Centre.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Yanggratoke, R., Ahmed, J., Ardelius, J., Flinta, C., Johnsson, A., Gillblad, D., Stadler, R.: Predicting real-time service-level metrics from device statistics. In: IFIP/IEEE International Symposium on Integrated Network Management (IM). pp. 414–422. (2015)
2. Yanggratoke, R., Ahmed, J., Ardelius, J., Flinta, C., Johnsson, A., Gillblad, D., Stadler, R.: Predicting service metrics for cluster-based services using real-time analytics. In: 11th International Conference on, IEEE Network and Service Management (CNSM). pp. 135–143. (2015)
3. VLC: <http://www.videolan.org/vlc/> (2016)
4. Voldemort: <http://www.project-voldemort.com/voldemort/> (2016)
5. Pasquini, R., Stadler, R.: Learning end-to-end application QoS from OpenFlow switch statistics. In: NetSoft Conference on IEEE Network Softwarization. (2017)
6. James, G., Witten, D., Hastie, T., Tibshirani, R.: An Introduction to Statistical Learning, vol. 6. Springer, Berlin (2013)
7. James, G., Witten, D., Hastie, T., Tibshirani, R.: An Introduction to Statistical Learning with Applications. Springer, Berlin (2014)
8. Breiman, L.: Random forests. *Mach. Learn.* **45**(1), 5–32 (2001)
9. SAR: <http://linux.die.net/man/1/sar> (2016)
10. Bowden, T., Bauer, B., Nerin, J., Feng, S., Seibold, S.: The /proc Filesystem. In: Linux Kernel Documentation. (2000)
11. ONF: OpenFlow Switch Specification-Version 1.5.0. (2014)
12. NTP: <http://www.ntp.org/> (2016)
13. HAProxy: <http://www.haproxy.org/> (2016)
14. Apache HTTP Server: <http://httpd.apache.org/> (2016)
15. FFmpeg: <https://www.ffmpeg.org/> (2016)
16. Gluster FS: <http://www.gluster.org/> (2016)

17. Yanggratoke, R.: Data-driven performance prediction and resource allocation for cloud services. In: PhD thesis, KTH Royal Institute of Technology (2016)
18. Yanggratoke, R., Ahmed, J., Ardelius, J., Flinta, C., Johnsson, A., Gillblad, D., Stadler, R.: A service-agnostic method for predicting service metrics in real-time. *Int. J. Netw. Manag.* (2017). doi:[10.1002/nem.1991](https://doi.org/10.1002/nem.1991)
19. Open vSwitch: <http://openvswitch.org/> (2016)
20. Hemminger, S.: Netem-Emulating real networks in the lab. In: *Proceedings of Linux Conference Australia*. (2005)
21. Floodlight: <http://www.projectfloodlight.org> (2016)
22. Ari, I., Hong, B., Miller, E.L., Brandt, S.A., Long, D.D.: Managing flash crowds on the internet. In: 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS). pp. 246–249. (2003)
23. Pasquini, R., Stadler, R.: JNSM 2017 Traces: infrastructure statistics and service-level metrics. <https://github.com/rafaelpasquini/traces-jnsm-2017> (2017)
24. Handurukande, S., Fedor, S., Wallin, S., Zach, M.: Magneto approach to QoS monitoring. In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. pp. 209–216. (2011)
25. Bodík, P., Griffith, R., Sutton, C.A., Fox, A., Jordan, M.I., Patterson, D.A.: Statistical machine learning makes automatic control practical for internet datacenters. *HotCloud* **9**, 12–12 (2009)
26. Matsunaga, A., Fortes, J.A.: On the use of machine learning to predict the time and resources consumed by applications. In: *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society. pp. 495–504. (2010)
27. Kundu, S., Rangaswami, R., Gulati, A., Zhao, M., Dutta, K.: Modeling virtualized applications using machine learning techniques. In: *ACM SIGPLAN Notices*. Vol. 47, pp. 3–14. (2012)
28. Hlavacs, H., Treutner, T.: Predicting web service levels during VM live migrations. In: *IEEE 5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management (SVM)*. pp. 1–10. (2011)
29. Menkovski, V., Oredope, A., Liotta, A., Sánchez, A.C.: Predicting quality of experience in multi-media streaming. In: *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (ACM)*. pp. 52–59. (2009)
30. Menkovski, V., Exarchakos, G., Liotta, A.: Online QoE prediction. In: *IEEE 2010 Second International Workshop on Quality of Multimedia Experience (QoMEX)*. pp. 118–123. (2010)
31. Song, H.H., Ge, Z., Mahimkar, A., Wang, J., Yates, J., Zhang, Y., Basso, A., Chen, M.: Q-score: proactive service quality assessment in a large IPTV system. In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference, ACM*. pp. 195–208. (2011)
32. Khan, A., Sun, L., Ifeachor, E.: Learning models for video quality prediction over wireless local area network and universal mobile telecommunication system networks. *IET commun.* **4**(12), 1389–1403 (2010)
33. Liu, Z., Zhang, Q., Zhani, M.F., Boutaba, R., Liu, Y., Gong, Z.: Dreams: dynamic RE source allocation for Mapreduce with data skew. In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. pp. 18–26. (2015)
34. Amaral, P., Dinis, J., Pinto, P., Bernardo, L., Tavares, J., Mamede, H.S.: Machine learning in software defined networks: data collection and traffic classification. In: *IEEE 24th International Conference on Network Protocols (ICNP)*. pp. 1–5. (2016)
35. Li, S.: Towards ultra-high speed online network traffic classification enhanced with machine learning algorithms and OpenFlow accelerators. University of Massachusetts Lowell (2012)
36. Nanda, S., Zafari, F., DeCusatis, C., Wedaa, E., Yang, B.: Predicting network attack patterns in SDN using machine learning approach. In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (IEEE NFV-SDN)*. (2016)
37. Ashraf, J., Latif, S.: Handling intrusion and DDoS attacks in software defined networks using machine learning techniques. In: *IEEE National Software Engineering Conference (NSEC)*. pp. 55–60. (2014)
38. Sommer, V.: Anomaly detection in the SDN control plane. Master's thesis, Technische Universität Munchen (2014)
39. Panousopoulou, A., Azkune, M., Tsakalides, P.: Feature selection for performance characterization in multi-hop wireless sensor networks. *Ad Hoc Netw.* **49**, 70–89 (2016)
40. Jiang, Z.: Predicting service metrics from device statistics in a container-based environment. In: Master's thesis, KTH, Communication Networks. (2015)
41. ISO: Iso/iec 23009-1:2014 dynamic adaptive streaming over http (dash) (2014)

42. Balachandran, A., Sekar, V., Akella, A., Seshan, S., Stoica, I., Zhang, H.: Developing a predictive model of quality of experience for internet video. In: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, ACM. pp. 339–350. (2013)

Rolf Stadler is a Professor with the Department of Network and Systems Engineering at KTH Royal Institute of Technology in Stockholm, Sweden. He holds an M.Sc. degree in mathematics and a Ph.D. in computer science from the University of Zurich. Before joining KTH in 2001, he held positions at the IBM Zurich Research Laboratory, Columbia University, and ETH Zurich. His group made contributions to real-time monitoring, resource management, and self-management for large-scale networks and clouds. His current interests include advanced monitoring techniques and data-driven methods for network engineering and management. Rolf Stadler is Editor-in-Chief of IEEE Transactions on Network and Service Management (TNSM).

Rafael Pasquini received his M.Sc. and Ph.D. degrees in computer engineering from the State University of Campinas in 2006 and 2011, respectively. In 2011 he joined Federal University of Uberlândia, where he now acts as assistant professor in the Faculty of Computer Science. His research interests include network management, machine learning, cloud computing, and software defined networks.

Viktoria Fodor received her M.Sc. and Ph.D. degrees in computer engineering from the Budapest University of Technology and Economics in 1992 and 1999, respectively. In 1999 she joined KTH Royal Institute of Technology, where she now acts as professor in the Department of Network and Systems Engineering. Her research interests include network performance evaluation, protocol design, wireless and multimedia networking, and distributed processing.