EarlyBird: Automating Application Signalling for Network Application Integration in Datacenters

Seyed Hossein Mortazavi*
Datacenter Networking Lab, Huawei Technologies Canada

Ali Munir
Datacenter Networking Lab, Huawei Technologies Canada

Mahmoud Mohamed Bahnasy
Datacenter Networking Lab, Huawei Technologies Canada

Haiwei Dong
Datacenter Networking Lab, Huawei Technologies Canada

Shimiao Wang
Datacenter Networking Lab, Huawei Technologies Canada

Yashar Ganjali†
Datacenter Networking Lab, Huawei Technologies Canada

ABSTRACT

Many recent studies in datacenter networking have proposed the idea of using information from applications for optimizing and resource planning. These *Application-Aware Networks* generally assume that applications can provide an accurate view about their requirements from the network and their traffic characteristics in real time. However, relying on the applications and developers to convey the traffic information is not realistic. We believe that automating the process of information extraction from applications is a crucial step towards realizing the idea of Network-Application Integration (NAI). In this paper, we investigate whether we can automatically identify places in the application code that, when executed, lead to predictable changes in the host's network output. By augmenting the application code at these execution places, we can generate explicit signals that can be used to predict local network events (such as changes in rate, bursts, etc.). This creates a mechanism for automatic adjustment of the network based on application signals.

We propose a combination of simple heuristics and learning methods to minimize the burden of the application developer for NAI. To the best of our knowledge, this is the first study that attempts to automatically realize NAI on the application. Our experimental evaluation shows a high accuracy (over 87%) of our prototype in predicting local network events such as micro-bursts.

CCS CONCEPTS

• **Networks** → **Network design principles**; **Data center networks**; *Cross-layer protocols.*

KEYWORDS

Network Tags, Datacenter Networks, Network Application Integration

*Corresponding author: seyed.hossein.mortazavi@huawei.com
†Also with University of Toronto.

NAI '22, August 22, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9395-9/22/08…$15.00
https://doi.org/10.1145/3538401.3546599

ACM Reference Format:
Seyed Hossein Mortazavi, Ali Munir, Mahmoud Mohamed Bahnasy, Haiwei Dong, Shimiao Wang, and Yashar Ganjali. 2022. EarlyBird: Automating Application Signalling for Network Application Integration in Datacenters. In *ACM SIGCOMM 2022 Workshop on Network-Application Integration (NAI '22), August 22, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3538401.3546599

1 INTRODUCTION

Advance knowledge of network events and their causes in a datacenter can often be leveraged to improve resource utilization, efficiency and performance of datacenter network and its applications [11, 18]. Examples of these network events include microbursts, rate changes on flows, establishing/terminating flows and changes in network topology or status of the network devices, etc. While some of these events occur due to changes in the network, the vast majority are originated because of changes in the application's communication patterns.

Legacy datacenters are not aware of application's communication patterns or requirements and do not utilize this information. They generally only rely on traditional traffic management techniques (such as shortest path routing and QoS priority queues) that are application agnostic because accurately calculating, analyzing and obtaining such information is challenging.

Recently, there has been a new wave of studies proposing the idea of Network-Application Integration (NAI) to share information between the network and applications [8–10, 15, 17]. The collaboration between the network and applications increases the overall efficiency of both the network and the application. On the network side for example, the earlier the network knows about the traffic and its properties, the more time is available to plan and mitigate network events. NAI, enables technologies such as clairvoyant schedulers that leverage the flow size information from the application to significantly improve network performance [5, 16].

However, there is a plausibility and feasibility question in how network related information can be obtained from the application [3, 18, 20, 21]. In many instances, this information may not be available to the application developer [1] or acquiring and transferring it to the network may require significant changes to the application. In other cases, the application developer may not know about the details of underlying communication structures. For example, a machine learning developer that is using Pytorch or TensorFlow in a distributed setting is usually only interested in the performance of the machine learning model. For many sophisticated applications

that are constantly evolving, extracting network related information becomes particularly difficult.

In this paper, we investigate how the information about the application state, that can help predict its communication patterns and their relation with the network events, can be extracted automatically and seamlessly without involving the application developer. This aligns with the assessment that NAI should not be left to the application developer [18].

To this end, we present *EarlyBird* a novel system that automates the application information extraction process on the host. The key idea of EarlyBird is to identify which parts of the application code, when executed, lead to network events. Using this information, EarlyBird then predicts local network events. EarlyBird works in two phases. First, it profiles an application in an offline process, and uses the timings between execution points in different parts of the code as an indicator for application state. It uses simple heuristics and correlation methods [2] to identify which of these application states correlate with the network events. Next, it generates a learning model based on these timings to predict network events.

The main contribution of this work is automating the process of application information extraction without the involvement or assistance of the application developer. EarlyBird is not limited to certain applications and can be generalized for different types of network events. This is a major benefit compared to other approaches that only focus on predicting specific metrics such as flow sizes. To the best of our knowledge, this is the first study to assess the feasibility of automating NAI on the applications.

Our initial evaluation with Distributed Machine Learning (DML) applications shows that EarlyBird achieves over 90% accuracy in predicting micro-bursts on the end-host. Our current prototype predicts local network events on end-hosts which is limited to the scope of an end-host but is lightweight and requires very little processing. Furthermore, network events that are locally observable (e.g. changes in rate, bursts, etc.) usually correlate with global network events. For example, by predicting local rate changes, we are able to predict in-cast happening remotely in the network. Similarly, a micro-burst happening at a switch, is correlated with several smaller bursts coming from various sources.

EarlyBird opens many venues for future research. By augmenting the application code at important execution places, we can generate explicit signals that can be used to predict network events. This creates a mechanism for automatic adjustment of the network based on application signals that can help predict network events several time units (order of ms) in advance, which can be employed by the network to improve scheduling, routing and rate limiting techniques to mitigate network events. Note that, as the network latencies are getting smaller (order of μs), legacy in-network scheduling and routing algorithms have very limited time to detect and react to network events. EarlyBird provides an automated way to extend the capabilities of such solutions.

The rest of the paper is organized as follows; Section 2 presents the motivation behind this work. The design of our proposed method is discussed in Section 3. In Section 4, we present our experimental results. We discuss related works in Section 5 followed by the conclusion in Section 6.

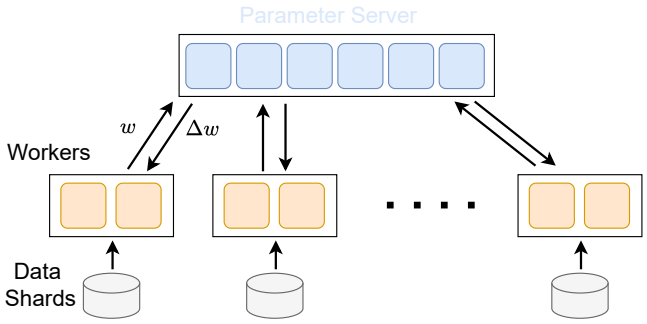


Figure 1: The parameter server architecture

2 CASE STUDY: DISTRIBUTED MACHINE LEARNING

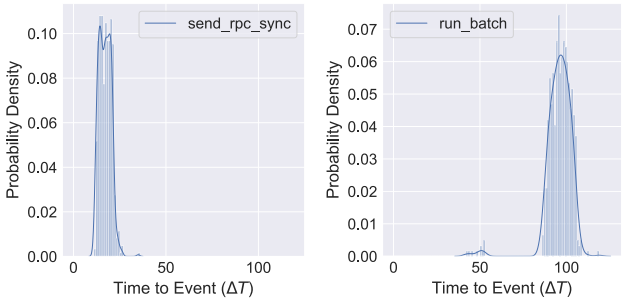
Integrating modern applications that commonly run in the datacenter such as DML systems, databases and big data processing systems with the networks is a tedious task. Integrating these application with the network as NAI suggests [8], requires the developers to have advanced knowledge about the expected network behavior [18] which for many applications can be an enormous task. This is due to the reason that such applications are composed of hundreds of thousands lines of code and can be immensely sophisticated in terms of logic.

In this section we use the example of Pytorch [14], a DML system to illustrate how network related information about the application state can be extracted automatically without overburdening the application developer.

DML across multiple machines is commonly used to leverage computational power over a cluster of machines in a datacenter. We specifically focus on the RPC-Based distributed training model that supports general training structures and assume a *Parameter Server* data-parallel method. A parameter server training cluster consists of *workers* and *parameter servers*. Each trains a local model based on a subset of the training data. In each iteration of training, model variables are pulled from the parameter servers, the gradients are then computed by the workers and pushed back to the parameter server. Workers read and update the variables independently without synchronizing with each other and updates are sent to the parameter server. At the end of each iteration the parameter server aggregates all updated gradients. This process is depicted in Figure 1.

Pytorch has different communication back-ends each with different capabilities. So the question is how the system can be integrated with the NAI frameworks without the developer having advanced knowledge about PyTorch or the underlying systems. We argue that if the application state and changes are known at anytime, the application's network behavior in the future can be predicted. An inexpensive way to capture the application state, is by going directly to the application code and identifying and monitoring parts of the application code that when executed can signal imminent network events.

In Figure 2, we illustrate the distribution of time between the start of execution of two functions and micro-burst events on a worker machine in a DML application. As shown in Figure 2, network



(a) *send_rpc_sync* **(b) *run_batch***

Figure 2: The distribution of time to event when the two different functions are called in a DML application

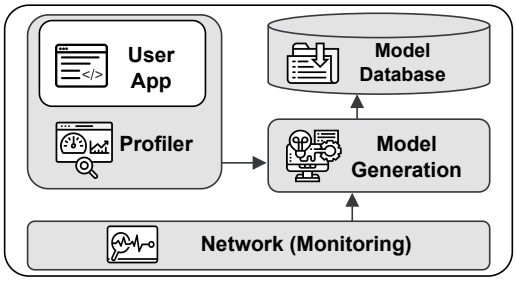


Figure 3: EarlyBird's offline modules in grey. Logs from the network module and profiler are passed for model generation. The result is saved in a local database.

events generally occur around 100*ms* after the *run_batch* function execution this time was around 20*ms* for the *send_rpc_sync* function. If the network has a good estimation of when micro-bursts will happen in the future, it can use various scheduling and rate control techniques to mitigate the effects.

3 DESIGN AND IMPLEMENTATION

One of EarlyBird's main goals is to identify places in the application code that when executed, result in a local network event. We refer to these places as *interest points*. Potentially each line of the application code can be an interest point. Our objective is to find these interest points for different types of network events. Signals from these interest points will then be used to predict whether a local network event will occur in a fixed time window or not.

The EarlyBird agent runs on each host and consists of an offline profiling and online monitoring module and a driver module containing the prediction model that is also responsible for communication with a centralized controller.

3.1 Offline Profiler

The offline profiling module is responsible for gathering data and correlating the local network events with function calls. This module profiles and logs code placeholders such as function calls with a timestamp while running the code in an offline setting. EarlyBird can profile the application code line-by-line but because profiling can be expensive in terms of resource usage, we provide filtering tools to narrow down the scope of potential interest points. This includes only profiling the start of functions, limiting the stack depth and limiting the profiling to certain classes or folders of the code. The logs gathered from the profiling step is then correlated to local network events. The overall architecture of the offline processing is depicted in Figure 3

To identify the *interest points* we first construct the temporal sequence of place holder calls (in our case function calls) $F_i(t)$, and the temporal sequence of local network events E_t.

To find the correlations between the function calls and the local network events, we use the Pearson correlation coefficient [2]:

$$\rho(F_i(t), E(t)) = \frac{cov(F_i(t), E(t))}{\sigma(F_i(t))\sigma(E(t))}$$
$$= \frac{\mathbb{E}[F_i(t)E(t)] - \mathbb{E}[F_i(t)]\mathbb{E}[F_i(t)]}{\sqrt{\mathbb{E}[F_i(t)^2] - (\mathbb{E}[F_i(t)])^2}\sqrt{\mathbb{E}[E(t)^2] - (\mathbb{E}[E(t)])^2}}$$

As the function executions and the local network events have a time lag between them, we shift F_t over time ($\Delta \in T$) and find Pearson's correlation coefficient on different time lags to establish a score vector for each F_i over time:

$$S_i(t)) = \rho(F_i(t + \Delta), E(t))$$

The higher the score in each window, the more likely that function calls will result in respective events. Using this score, we filter out functions that don't have a score point above a given threshold for different time windows. We use the rest of the functions as interest points. In our experiments, this filters out over 85% of functions for the DML application.

At this stage, we also extract statistical information about the intensity and duration of each local network event. With the interest points identified, EarlyBird uses two mechanisms to create signatures for live event detection.

3.2 Learning-based Signature Generation

Once the interest points for each local network event type are identified, we create a model that predicts these events based on interest point execution. This model outputs a signature that represent the event properties (such as size and duration) and when the event is going to happen.

In the simplest setting, one can simply use the direct mapping between a sole interest point to a local network event. Whenever that certain interest point is executed, signal the network about the probability of a network event occurrence based on the probability distribution of the interest point causing that event (such as data from Figure 2). However, this method will not capture many of the events (low recall) as it does not capture the relations between the execution of multiple interest points that lead to a network event.

Instead, we build a machine learning model that predicts event occurrence in a fixed time period (ΔT) seconds. As we want to find the relationship between interest point executions and network events, we use the correlation scores of function occurrences as features to train our model. In this way, we enrich the probability calculation by training the prediction module to identify the

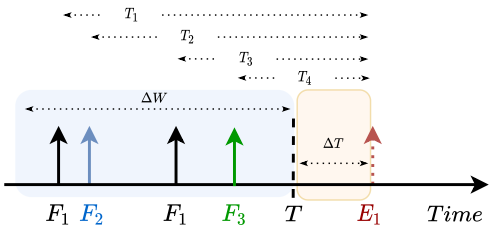


Figure 4: Predicting events ΔT seconds away using information from interest point execution in a ΔW time window

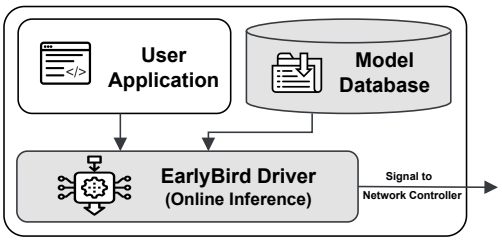


Figure 5: EarlyBird's online modules shown in grey. The driver loads appropriate models from the database and receives function signals from the application. It then runs the inference model to predict events and signals the network controller

relation, not one execution point to one event, but a sequence of execution points and their corresponding network event. Data points in our training data set contains the execution history of interest points within a certain time window (ΔW).

For example in Figure 4, F_1, F_2, F_3 have been executed during the blue time window with a length of ΔW. We lookup the interest point's correlation score vectors and calculate $S_1(T_1) + S_1(T_3)$ for F_1, $S_2(T_2)$ for F_2 and $S_3(T_4)$ for F_4. The corresponding datapoint would be a vector:

$$[S_1(T_1) + S_1(T_3) \quad , S_2(T_2) \quad , S_3(T_4) \quad , 0 \quad \dots]$$

These points are then fed to a learning algorithm that generates the learning model. In our experiments we use a simple 3 layer neural network and an Adaboost ensemble with 5000 estimators and compare their performances.

3.3 Online Inference and Live Signalling

Once the classification model is generated, we save the model along with the correlation score vectors on a local database on the host. We also augment the application code by adding function calls to a local driver module at the interest points.

During application execution, whenever a interest point is executed, a function call is made to EarlyBird's driver. The driver records interest point executions for the past ΔW seconds. The driver then looks up up correlation scores and generates data points with the same method explained above. Note that without identifying interest points, logging all possible point executions in the code is impractical and has a huge overhead.

The data points are then given as an input to the classifier for inference. The classifier along with the correlation scores is loaded from a local database on EarlyBird's driver at application startup. If an event is predicted, the EarlyBird driver signals the network about the imminent network event along with the network event's probability of occurrence, it's intensity and duration.

This signal will then be used to inform the network about a possible change in the application's network behavior. Routing, scheduling and rate adjustment techniques can then be used to react to these signals. The sooner the controller receives these signals, the more time it has to process and mitigate.

4 RESULTS

We evaluate EarlyBird on a DML (Pytorch) running on the parameter server setting. The goal of this experiment is to see whether EarlyBird and its learning models can correctly predict network events just from application logs gathered by the profiler. We chose the DML application as its one of the most common workloads in the datacenter. We extract information by profiling one of the worker machines and filter interest points by only profiling function calls with a stack depth of less than 15.

For network events, we chose to identify micro-bursts by analyzing Tcpdump logs. Micro-bursts are difficult to handle in datacenter network and an early signalling mechanism can assist the network with mitigation.

In our experiments, we assume that the list of potential interest points has been limited by the application developer to the first line of every function and we limit logging functions with a depth of less than 15 on the stack. We predict network events on two different time resolutions (1ms, 10ms in our experiments). We run our experiments for 10 minutes and gather over 900, 000 function calls from 526 different functions.

4.1 Correlation Scores

We obtain the correlation score vectors for each function by calculating Pearson's correlation for different time lags. In Figure 6, the heat map of the correlation scores of different functions for different time windows is depicted. We only select 12 functions out of a total 564 due to space limitation. The x-axis depicts the time lag and the y-axis contains the function calls. This Figure also shows the underlying relation between function calls and micro-bursts. For example calls to the *send_data_rpc* method, generally result in micro-bursts after 20 milliseconds. From the Figure, one can also deduce the computation time between learning iterations.

From this Figure, any application developer can understand the relation between interest point execution and local network event occurrence and even without building a learning model for event prediction, the application developer is able to identify important placeholders.

4.2 Learning Model Accuracy

Using the correlation scores from the previous section, and using the logs of over 900, 000 function calls, we train two machine learning models, a 3 layer neural network and an AdaBoost Ensemble. As

		10 ms window resolution		1ms window resolution	
		Neural Network	AdaBoost Ensemble	Neural Network	Adaboost Ensamble
Event	Precision	**88%**	**87%**	89%	87%
	Recall	**87%**	**87%**	82%	79%
Non Event	Precision	99%	98%	99%	99%
	Recall	98%	98%	98%	99%
ROC AUC score		**0.94**	**0.9**	0.9	0 .87
Accuracy		98%	97%	96%	95%

Table 1: The performance of EarlyBird's learning algorithm.

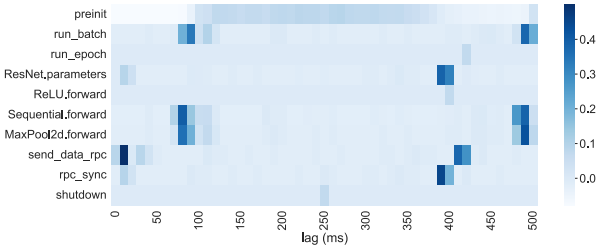


Figure 6: The correlation between selected function calls and micro-bursts for a DML app.

the ratio of non-events to events is small in our data set we use machine learning techniques such as undersampling used in anomaly detection to increase the precision and recall of our models.

The performance results of our model in Table 1. We run two sets of experiments, one with a time window resolution of 1ms where we keep a history time window (ΔW) of 40ms one with a time window resolution of 10ms where we keep a history time window of 100ms. For both experiments we predict the time window 20ms in the future (ΔT).

As shown in the Figure EarlyBird achieves high precision & recall for both time resolutions using both learning models. While the neural network performs slightly better, we suggest using the AdaBoost Ensemble as it does not require much hyper-parameter tuning and is ideal for EarlyBird that targets to automate the whole process of signal generation for the application with minimal developer involvement. While the training time depends on the amount of input data, the whole classification time only takes a few microseconds. Our predictions come with a confidence score that can be sent as part of the signal to the network controller.

4.3 Discussion

Network input and output is generated as a result of changes in the application state. Approaches that use periodic passive monitoring techniques (CPU, memory, system calls, etc) to predict flow sizes, are by some means indirectly inferring the application state. However, the live monitoring process can be expensive on the host as it requires frequent operations to gather time sensitive information from multiple components. Placing function calls in the application code as trigger points to EarlyBird's driver is much faster compared to other systems that predict events based on more complicated machine learning models such as in [18, 20] as there is less computing required in the live phase. The few hundred millisecond processing requirements on these systems also impact the time resolution of predictions. While EarlyBird operates in a different domain than these approaches, it can also complement them by (i) Improving the prediction accuracy by providing information from the application and (ii) reducing the online inference overhead by only triggering monitoring when certain functions are called.

Predicting network wide events (such as detecting congestion on switches) requires centralized monitoring. EarlyBird will be a signalling mechanism for the centralized control module of those systems, warning about incoming traffic from the endhosts while a centralized correlation system such as SmartTags [12] can identify the causal relationship between signals and network wide events.

EarlyBird currently targets open source applications (such as DML or distributed streaming platforms) where the code can be shared with the network. However, EarlyBird's profiler as a tool can assist application developers with finding interest points in the application code and providing signals manually.

While we expect the interest points to be the same for different application workloads, the timing between execution points can differ meaning that in our current implementation, a different learning model should be generated for each application workload, however we plan to extend EarlyBird to automatically adjust the learning models in real-time using online learning techniques.

5 RELATED WORK

With the evolution of network controllers with more processing capabilities, several scheduling techniques have been proposed that rely on a priori information about flow sizes [4, 5, 16]. Some methods propose passive monitoring or indirect anticipation of application behavior without the applications direct involvement, these include (i) flow size prediction based on machine learning techniques using data from various system components such as CPU, memory, etc. [18, 20], (ii) Monitoring TCP send buffer occupancy for estimations on flow sizes [13], (iii) Estimating remaining flow sizes based on data already sent [1], (iv) Predicting flow sizes based on previous traces [6]. These approaches, however, have their own shortcomings. First, they do not exploit information on the application level. Second, they require more processing power effecting the application's runtime and third, they are limited to certain applications and workloads. Until recently, researchers have mainly overlooked Network Application Integration in datacenters [8].

Multiple studies [7, 11, 19], demonstrate how application-aware networks in the context of NAI can potentially benefit from direct communication between the application and the network. These

approaches propose new communication interfaces and suggest Directed acyclic graph (DAG) like graphs for application tasks to be transferred to the network. However, modern applications that commonly run in the datacenter such as distributed machine learning systems, databases and big data processing systems are composed of hundreds of thousands lines of code and can be immensely sophisticated in terms of logic. Integrating these applications with the network is an arduous task and requires the application developers to have advanced knowledge about the expected network behavior [18]. A system like EarlyBird that automatically enables NAI for these systems is required.

6 CONCLUSION

In this paper, we presented EarlyBird, a novel automated system for finding points in the application code that lead to changes in the application's network output and predicting network events on the end-host. Our argument was based on the observation that changes in the application state can lead to network events. EarlyBird provides the tools and means for the developer to integrate its application with the network. Our prototype achieves high accuracy in predicting micro-bursts on end-hosts for a distributed learning application without any assistance or intervention from the application developer.

While in this work we focused on using the information from the application to predict local network events, this information can also be used to predict network wide events [12]. Furthermore, we can envision this information being used for processing and storage management in addition to the network within a datacenter.

As for future work, we plan to test EarlyBird on more generic applications such as databases and also expand our implementation to gather information about method parameters and application variables to make our predictions more accurate. We will also test our system with other type of network events such as changes in rates or the initiation of flows. We will integrate passive monitoring techniques on the host to complement EarlyBird's prediction methods. Finally, we plan to extend the profiling platforms to include other programming languages such as Java and C++.

REFERENCES

[1] Wei Bai, Li Chen, Kai Chen, et al. 2015. {Information-Agnostic} Flow Scheduling for Commodity Data Centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 455–468.

[2] Jacob Benesty, Jingdong Chen, Yiteng Huang, et al. 2009. Pearson correlation coefficient. In *Noise reduction in speech processing*. Springer.

[3] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 393–406.

[4] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 443–454.

[5] Peter X Gao, Akshay Narayan, Gautam Kumar, et al. 2015. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 1–12.

[6] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, et al. 2016. Morpheus: Towards Automated {SLOs} for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 117–134.

[7] Sangeetha Abdu Jyothi, Sayed Hadi Hashemi, Roy Campbell, et al. 2020. Towards an application objective-aware network interface. In *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*. 16–16.

[8] Danny Lachos, Qiao Xiang, Christian Rothenberg, et al. 2020. Towards deep network & application integration: Possibilities, challenges, and research directions. In *Proceedings of the Workshop on Network Application Integration/CoDesign*. 1–7.

[9] HE Lin, Peng KUANG, WANG Shicheng, et al. 2019. Application-aware IPv6 networking. *Telecommunications Science* 36, 8 (2019), 36.

[10] Takuya Miyasaka, Yuichiro Hei, and Takeshi Kitahara. 2020. NetworkAPI: An Inband Signalling Application-aware Traffic Engineering using SRv6 and IP anycast. In *Proceedings of the Workshop on Network Application Integration/CoDesign*. 8–13.

[11] Seyed Hossein Mortazavi, Hossein Shafieirad, Mahmoud Bahnasy, et al. 2021. Accord: Application-Driven Networking in the Datacenter. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing (UCC '21)*. Article 13, 10 pages.

[12] Ali Munir, Seyed Hossein Mortazavi, Mahmoud Mohamed Bahnasy, et al. 2022. Toward Stable Interdomain Network-Application Integration and. In *Proceedings of the ACM SIGCOMM 2022 Workshop on Network-Application Integration*.

[13] Aisha Mushtaq, Radhika Mittal, James McCauley, et al. 2019. Datacenter congestion control: Identifying what is essential and making it practical. *ACM SIGCOMM Computer Communication Review* 49, 3 (2019), 32–38.

[14] Adam Paszke, Sam Gross, Francisco Massa, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[15] Danny Alex Lachos Perez, Christian Esteve Rothenberg, Mateus Santos, et al. 2020. Ani: Abstracted network inventory for streamlined service placement in distributed clouds. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 319–325.

[16] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, et al. 2014. Fastpass: A centralized" zero-queue" datacenter network. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 307–318.

[17] Philipp S Schmidt, Theresa Enghardt, Ramin Khalili, et al. 2013. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. 295–300.

[18] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, et al. 2019. Is advance knowledge of flow sizes a plausible assumption?. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 565–580.

[19] Weitao Wang, Sushovan Das, Xinyu Crystal Wu, et al. 2021. MXDAG: A Hybrid Abstraction for Emerging Applications. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 221–228.

[20] Renhai Xu, Wenxin Li, Keqiu Li, et al. 2021. DarkTE: Towards Dark Traffic Engineering in Data Center Networks with Ensemble Learning. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 1–10.

[21] Hong Zhang, Li Chen, Bairen Yi, et al. 2016. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 160–173.