Poster: ReactNet: Self-Adjusting Architecture for Networked Systems

Habib Mostafaei
TU Berlin

Seyed Milad Miri
TU Berlin

Stefan Schmid
TU Berlin, University of Vienna & Fraunhofer SIT

ABSTRACT

Providers today run numerous applications on their networks with diverse quality of service requirements. An appealing vision to deal with the resulting complexity of network operation, is to give more control to the network, allowing it to become more autonomous and to dynamically "self-adjust", to meet its requirements. This paper presents an architecture, ReactNet, to realize this vision, by leveraging two enabling technologies. First, we use programmable dataplanes and P4 to get accurate information about the traffic patterns the network currently serves. Second, we leverage Machine Learning (ML) techniques to process this information and react to the network changes dynamically.

CCS CONCEPTS

• **Networks** → **Network manageability**; Programmable networks.

KEYWORDS

Self-adjusting networks, programmable dataplane, machine learning

ACM Reference Format:
Habib Mostafaei, Seyed Milad Miri, and Stefan Schmid. 2021. Poster: ReactNet: Self-Adjusting Architecture for Networked Systems. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3485983.3493348

1 INTRODUCTION

The complexity of cloud provider networks as well as the amount of traffic served by these networks is growing rapidly. To account for the diverse mix of application needs, network operators often develop customized scripts to tailor the network to specific workloads [3]. Besides the complexity of providing the different desired quality-of-service requirements, the efficient operation of these networks is also complicated by the limited insights operators typically have into the traffic currently served by their networks.

This paper is motivated by the vision of self-adjusting networks, that is, more autonomous networks that unburden human operators from many manual tasks (a.k.a. self-driving networks [3]). In particular, self-adjusting networks should be able to automatically observe their current state, and dynamically react accordingly, to optimize for specific performance goals.

A first key enabler is the increasing programmability and softwarization of networks: this not only allows collecting additional information about the network traffic and load (from the packet processing switches), but also to flexibly adjust network configurations. For example, network operators could place programmable switches where their access networks connect to the Internet [4], to obtain fine-grained information using either active or passive measurements [3, 7].

A second key enabler is Machine Learning (ML), which based on measurements earlier on allows deriving patterns and optimize for oncoming traffic flows. However, employing such approaches is still challenging, as they need to apply to various use cases and apply also to dynamic traffic patterns, requiring minimal parameter tuning. Thus, a self-adjusting system needs to update its training data with oncoming traffic flows to better drive the network.

Contributions: The ReactNet allows the admins to collect the data stream for the ML-based classification whenever needed. It can adapt itself to the traffic changes and update the forwarding rules according to the network demands. We implement our system using the programmable data plane, e.g., P4 [2], to test its feasibility.

2 SYSTEM DESIGN

To design the envisioned self-adjusting network, we need to account for interactions of many components, collect the information regarding the network status and apply optimized mechanisms to better handle the current situations of the network. We design and develop ReactNet in P4 that has five main components: *P4 switch*, *collector*, *ML classifier*, *optimizer* and *agent*. The P4 switch mirrors the traffic according to desired header fields. The controller applies ML classifiers to classify the packets based on the user-defined classes. The P4 runtime agent updates the rules. We now explain each component of the system architecture in detail (Fig. 1).

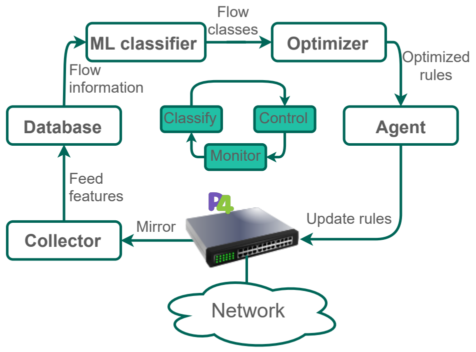

Figure 1: The architecture of our system.

ReactNet allows the network operators to collect a training data stream whenever needed using a flag. If the flag is set, the system mirrors the traffic to the designated egress port. We use a P4 register to store the flag value. ReactNet leverages the In-band Network Telemetry (INT) provided by P4 [8] to feed the data stream. We collect the following information: source and destination IP addresses, source and destination ports, protocol number, enqueue and dequeue sizes, packet size, ingress_port, packet inter-arrival time, and the packet processing time. We leverage the `egress_clone` when mirroring the traffic. Currently, the system mirrors all the traffic to the traffic collector, but the network operator can tune this parameter to the desired intervals [6].

The *collector* component uses Logstash [1] to log the telemetry packets mirrored from the P4 switch to the traffic collector. This component also formats the data and inserts them into the dataset implemented using *influxDB*. ReactNet stores the existing records into an appropriate file. Since each packet has a different processing time at the switch, we use four labels when collecting the data. More specifically, we use a range of integer numbers for each group of packet processing time.

After collecting the dataset, the ReactNet applies the *ML classifiers* to classify the data. We use the widely used classifiers in networking [5]: K-Nearest Neighbor, AdaBoost, Decision Tree, Support-Vector Machine (SVM), Random Forest, and Logistic Regression. The results of the classification are fed into *Optimizer* component of the system to make the corresponding decision for each class of the traffic. The *Optimizer* component also generates the required forwarding rules to be placed on top of the switches.

The *Agent* component of the system writes the forwarding rules to handle the upcoming traffic based on the detected traffic pattern. Also, it sets the flag whenever the dataset needs updating with the new telemetry information.

3 PRELIMINARY RESULTS

We set up a dumbbell topology with two switches, namely, s1 and s2, connected with a 10Mbps link to test the performance of the system and the ML-based classification techniques. We connect two hosts to each switch and use Mininet with BMv2 for the emulation. We use *iPerf* to generate the traffic from the two hosts connected to switch s1, and the two hosts connected to switch s2 are the traffic sink. One of the source hosts generates TCP traffic while the other one generates UDP traffic. We run experiments for 20 minutes to collect enough data for our dataset.

Table 1: Classifying our architecture's traffic.

Model	A	FPR	TPR	P	FS
AdaBoost	0.69	0.09	0.42	0.78	0.55
Decision Tree	0.89	0.1	0.89	0.87	0.89
K-Nearest Neighbors	0.86	0.14	0.88	0.83	0.86
Logistic Regression	0.85	0.16	0.87	0.81	0.85
Naïve Bayes	0.67	0.45	0.94	0.50	0.65
Random Forest	0.90	0.1	0.90	0.88	0.89
SVM	0.86	0.15	0.87	0.85	0.85

To test the dataset, we attach the traffic collector to switch s1 that mirrors the traffic. The ReactNet extracts the headers of the collected mirrored packets and applies each of the ML-based techniques to classify them. We use four different traffic classes to differentiate the packets using the packet processing time. Table 1 shows that *Random Forest* has the highest accuracy (90%) among the other classifiers.

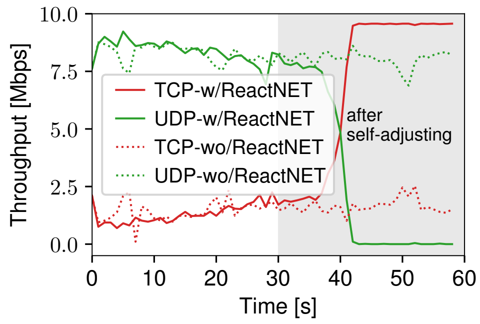

Figure 2: Impact of changing the priority after ML classification.

The optimizer of ReactNet assigns the highest priority using DiffServ header fields in the IPv4 header to the class of packets that the system needs to serve earlier. The agent updates the corresponding P4 register on top of switch s1 afterward. We assess the impact of the optimizer on applying the higher priority to TCP flows when competing with UDP ones. Fig. 2 depicts the throughput of the different flows by classifying them with and without using ReactNet after 30 seconds– for a 60-second experiment with the egress queue size of 50 packets.

Future work. We plan to apply our system to autonomously run a cluster of distributed stream processing systems such as Apache Flink and Apache Storm. Many organizations deploy these systems to get insights from their customers. Thus, depending on the running application, the network operators apply different traffic policies. We also plan to empower our optimizer with more sophisticated approaches to handle the network traffic flows.

ACKNOWLEDGMENT

This work was partially funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A), as well as by the Austrian Science Fund (FWF) project I 4800-N (ADVISE), lead agency: German Research Foundation (DFG), 2020-2023.

REFERENCES

[1] 2021. Logstash: Collect, Parse, Transform Logs. https://www.elastic.co/logstash/.
[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (2014), 87–95.
[3] Nick Feamster and Jennifer Rexford. 2017. Why (and How) Networks Should Run Themselves. *CoRR* (2017). http://arxiv.org/abs/1710.11583
[4] Sharat Chandra Madanapalli, Hassan Habibi Gharakheili, and Vijay Sivaraman. 2019. Assisting Delay and Bandwidth Sensitive Applications in a Self-Driving Network *(NetAI'19)*. 64–69.
[5] Donald McGaughey, Trevor Semeniuk, Ron Smith, and Scott Knight. 2018. A systematic approach of feature selection for encrypted network traffic classification. In *2018 Annual IEEE International Systems Conference (SysCon)*. 1–8.
[6] Habib Mostafaei and Shafi Afridi. 2021. P4Flow: Monitoring Traffic Flows with Programmable Networks. *IEEE Communications Letters* (2021), 1–1.
[7] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *NSDI 21*. 785–808.

[8] The P4.org Applications Working Group. 2020. In-band network telemetry (INT) dataplane specification v2.1. https://github.com/p4lang/p4-applications/tree/master/docs.