Failure Mitigation in Software Defined Networking Employing Load Type Prediction

Nader Bouacida, Amer Alghadhban, Shiyam Alalmaei, Haneen Mohammed and Basem Shihada

King Abdullah University of Science and Technology (KAUST)
Computer, Electrical and Mathematical Science and Engineering Division (CEMSE)
Thuwal 23955-6900, Saudi Arabia
Emails: {nader.bouacida, amer.alghadhban, shiyam.alalmaei, haneen.mohammed, basem.shihada}@kaust.edu.sa

***Abstract*—The controller is a critical piece of the SDN architecture, where it is considered as the mastermind of SDN networks. Thus, its failure will cause a significant portion of the network to fail. Overload is one of the common causes of failure since the controller is frequently invoked by new mice flows. Even through SDN controllers are often replicated, the significant recovery time can be an overkill for the availability of the entire network. In order to overcome the problem of the overloaded controller failure in SDN, this paper proposes a novel controller offload solution for failure mitigation based on a prediction module that anticipates the presence of a harmful long-term load. In fact, the long-standing load would eventually overwhelm the controller leading to a possible failure. To predict whether the load in the controller is short-term or long-term load, we used three different classification algorithms: Support Vector Machine, k-Nearest Neighbors, and Naive Bayes. Our evaluation results demonstrate that Support Vector Machine algorithm is applicable for detecting the type of load with an accuracy of 97.93% in a real-time scenario. Besides, our scheme succeeded to offload the controller by switching between the reactive and proactive mode in response to the prediction module output.**

***Index Terms*—Software Defined Networking, load type prediction, Support Vector Machine, controller offloading, failure mitigation, overload**

I. Introduction

Recently, Software-defined networking (SDN) has been growing in popularity because it addresses the lack of programmability in existing networking architectures. Moreover, it enables easier and faster network innovations by providing a more structured software environment for developing network-wide abstractions while potentially simplifying the data plane. SDN separates the data plane from the control plane and facilitates software implementations of complex networking applications.

Researchers proposed OpenFlow [1], which is the predominant control protocol in SDN. OpenFlow switches are the core network devices in the data plane which implement data transmission functions. The SDN controller provides a global network view for the OpenFlow switches by acquiring network topology information. In this way, it can implement flexible network management. Indeed, it plays the role of network brain, which has total control over dummy data-plane devices and mainly responsible for flow tables installation and maintenance.

However, such fine-grained control does not come without costs. Any failure in the central controller will cause an important part of the network to be functional only for a limited period of time before the flow entries expire. In this situation, the data-plane devices are left paralyzed in front of the incoming flows and the network will fail totally because the switches are unable to take forwarding decisions on new arriving flows. SDN networks are often designed to cope with such failures. However, failure recovery mechanisms need a significant delay to achieve the urgent recovery, especially with control-plane devices. Recovering from network failures in the data plane, such as link failure, can be performed shortly in order of milliseconds by rerouting the affected traffic to backup paths [2]. However, the failure of the control-plane devices will have a catastrophic impact on the whole network and takes a large time in order of seconds to recover and restore the network connections. Initial SDNs used to deploy a single controller which makes the resiliency issue even a bigger deal.

The straightforward solution is installing proactive flow entries in data-plane devices [3]. Although this solution will handle incoming flows in case of failure, it has two main drawbacks. First, it converts SDN structure from smart "reactive" paradigm into static dummy paradigm, where new flows are handled by the same static proactively installed flow entries of previous flows, similar to source routing. Second, the controller will not be involved in the flow setup, which prohibits the latter to react in response to network changes.

In this paper, we aim at addressing the SDN controller failure caused by overload. We propose a practical controller failure mitigation mechanism based on techniques from data mining and machine learning. Our scheme detects the long-term traffic load in early stage via an SVM classifier. Then, based on the load prediction output, the overloaded controller will decide if it is necessary to offload the traffic by switching to the proactive mode. In this case, OpenFlow switches will forward the incoming packets based on proactive flow tables, which have been pre-installed by the controller. In fact, heavy traffic loads on the controller are likely to cause a failure if they last for a long time. Thus, detecting long-term loads early enough will alert us to take action and offload the controller before falling into a potential failure.

To summarize, the main contributions of this paper are as

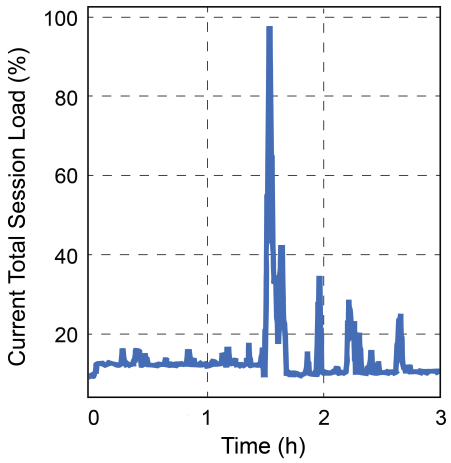


Fig. 1: Short-term load faced by an in-production network server in a cloud service provider.

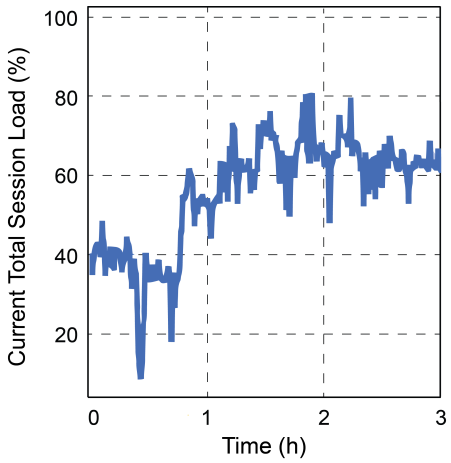


Fig. 2: Long-term load faced by an in-production network server in a cloud service provider.

follow:

- We design a preventive offloading approach for controller failure mitigation while preserving enough visibility of the network.
- We demonstrate that classification algorithms can be used effectively to detect long-term load. We used different classification methods to compare the performance and to understand the strengths and limitations of each method. We have been able to correctly predict the type of controller load in 97.93% of the cases.
- We conduct a real-time evaluation to confirm previous results and show that our framework offloads the controller without introducing extra latency.

The rest of this paper is organized as follows. First, we highlight the challenges and explain the system design. Second, we introduce our prediction framework. Next, we provide the results and discuss the real-time evaluation setup. Finally, we close with the description of related approaches and the conclusion.

II. Challenges

Before introducing the system design, we highlight the challenges faced by this work, which are mainly: (1) Load type differentiation, (2) Flow-entries offloading delay.

Load type differentiation: In a real network, there exist two main classes of traffic load: (1) A persistent long-term traffic load, such as database queries or new connections, (2) Short-term traffic pulses, such as spikes of load resulting from a storm of connections in a backup server after a failure on the primary server. Our idea is to switch from proactive to the reactive mode when a failure is predicted. The execution of this procedure has its cost on the network performance. For instance, some services that are provided by the controller in the reactive mode will not be available during the proactive mode. Thereby, the offloading procedure needs to be executed for a valid reason, i.e., true prediction of a long-term load. Thus, the proposed solution should demonstrate high accuracy in predicting the load type. The Fig. 1 and Fig. 2 give an example of short-term and long-term loads. The illustrated data was derived from in-production network server running in a cloud service provider datacenter.

Flow-entries offloading delay: In order to switch from reactive to proactive mode, several flow-entries need to be installed into the data-plane devices to handle flows routing during the failure of the controller. The problem exists when the offloading of such flow-entries exceeds the look-ahead interval of the implemented solution. Thereby, the proposed scheme needs to minimize the offloading time to an acceptable value considering different scenarios.

III. System Design

To prevent the controller from falling into a possible failure due to overload, we propose a novel schema that leverages a machine learning algorithm for load type prediction. As illustrated in Fig. 3, the proposed design is composed of three main modules: Statistics Collector, Load Type Predictor, and Traffic Manager. The Statistics Collector keeps track of different traffic and controller statistics, such as average inter-arrival time, CPU load, buffers usage, etc. When the controller becomes overloaded, we trigger the prediction process and the information gathered by the Statistics Collector are fed into the Load Predictor module that will be responsible for differentiating long-term load from a short-term load. Indeed, this component, based on a learned decision function, will predict if the network traffic load will last for a considerable time, where it can be considered as long-term load, or it is just dealing with temporary spikes of load. In case the controller predicts that it is facing long-term load, this situation may overwhelm it with time and here comes the role of Traffic Manager. It had to quickly react by offloading the traffic management to the switches, which contain static proactive flow tables for the purpose of forwarding the packets without involving the controller. Once the controller is relaxed, it can

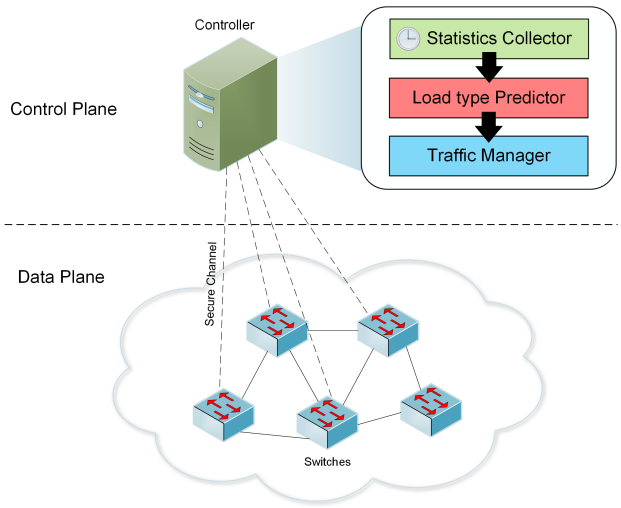


Fig. 3: System Design

switch back to reactive mode. Each module functioning is explained with further details in the following subsections.

Statistics Collector: The Statistics Collector module monitors various controller statistics related to load, where all experiments were carried out using POX controller [4] software written in Python language. For this purpose, we used psutil tool [5] to track, within a time interval set to one second, the CPU load, sent and received data in bytes, RAM usage, and buffers occupancy. Additionally, we keep track of average inter-arrival time, average processing time for incoming packets, the number of flows within the time interval and the total flows count. In total, we collected nine features. All those statistics are relevant for predicting the traffic behavior in the controller.

Load Type Predictor: Classification is one of the most efficient machine learning methods, which can be utilized to predict if the load will last for a long time or not. After going through a learning phase, the classifier will generate a decision function that will decide on future observations from the testing dataset. The corresponding load will be classified as short-term or long-term. We defer the explanation of the learning phase and the generation of decision function to section IV. To conclude the classification performance, both online and offline evaluation are examined. For offline evaluation, we tested three well-known classification algorithms: Support Vector Machine, k-Nearest Neighbors, and Naive Bayes. Meanwhile, the online evaluation is confined to the SVM algorithm, which achieves the best performance among the three methods.

Traffic Manager: Our switches use two flow tables, namely *Table 0* and *Table 1*. *Table 0* will be considered as the primary table, which contains the reactive flow entries dynamically learned by the controller. These reactive flow entries are temporary and will expire after a timeout period. The ability to dynamically reconfigure the network behavior allows a fine-grained traffic engineering, which leads to a faster response to emerging applications requirements that the network has to support [6]. However, as mentioned previously, the reactive flow installation may cause the controller to get overloaded with bursty and highly loaded traffic, hence, it cannot serve new incoming packets anymore and risk falling into failure. Therefore, once the controller predicts that it is overloaded with a long-term load, the Traffic Manager module will relay the traffic management to the switches. Therefore, switches must hold some proactively installed (static) flow entries stored in *Table 1*. In this case, data-plane devices will forward the packets based on the proactive rules installed in *Table 1*. They will keep forwarding the traffic for some time until the controller becomes offloaded. Then, they can shift the traffic management back to the controller to manage it dynamically. In summary, the controller will order the data-plane devices to switch from *Table 0* to *Table 1* after getting a warning, coming from Load Type Predictor module, predicting that the current load will last for a long time. The offloading process is initiated by broadcasting a single control packet by the controller, thereby avoiding extra overhead and delay.

IV. Prediction Model Framework

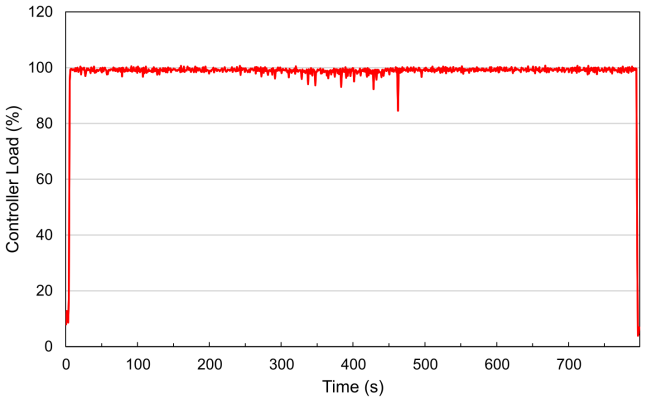
This section is devoted to describe the prediction process and its parameters tuning. Before that, we highlight the dataset generation approach for decision function learning and offline evaluation. In this part, the terms "classification" and "prediction" are used interchangeably.

A. Traffic Generation

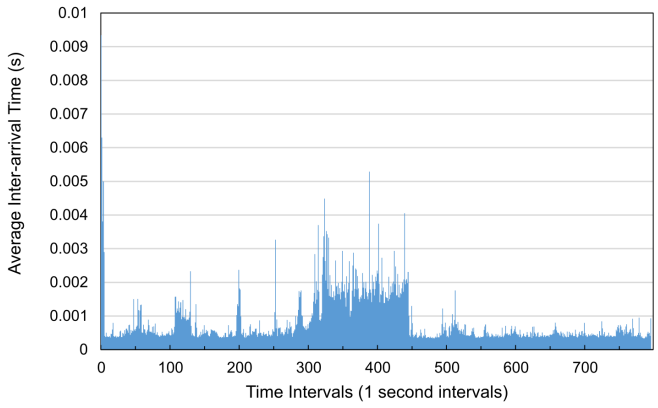
We have evaluated our solution with POX controller [4] over the Mininet network emulator [7]. We introduced traffic with 64 emulated hosts in Mininet arranged in a tree topology with fanout 8 and depth 2. We draw the attention of the reader that the controller is emulated in a separate virtual machine to ensure that the CPU load generated by emulating the topology does not interfere with the controller load. Since we are primarily concerned with the control-plane traffic load, even a relatively small data center topology can easily stress the limits of simulation [8]. The controller load mainly depends on the traffic patterns.

To generate the long-term and short-term traffic load patterns, we consider injecting *Packet-In* messages to the controller. In fact, the *Packet-In* message is a way for the switch to send a captured packet to the controller via the control channel when this packet does not match any flow entry in its flow table. Each host will generate random traffic with packet inter-arrival time following the exponential distribution [9]. When the controller is under-utilized, the number of incoming packets within a second is around 100-300 packets. Meanwhile, when the controller is highly utilized, the number of incoming packets is around 2000-4000 packets.

To vary the traffic behavior between long-term load and load spikes, we set the mean of the exponential distribution of packet inter-arrival time in each host to a small value, equal to 0.001s, over an extended period. As a result, this traffic pattern will generate long-term load which makes the controller highly utilized for a long duration. On the contrary, for the sake of

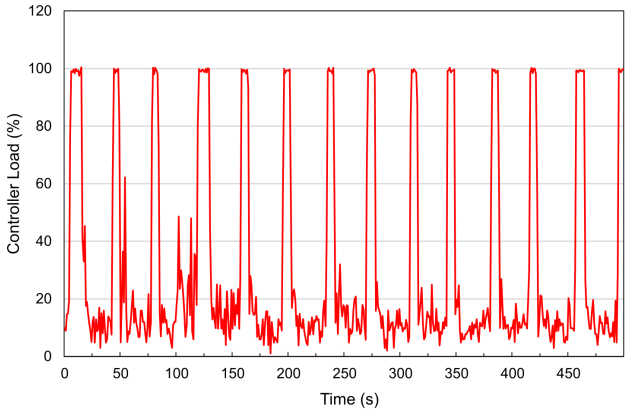


(a) Controller load in percentage

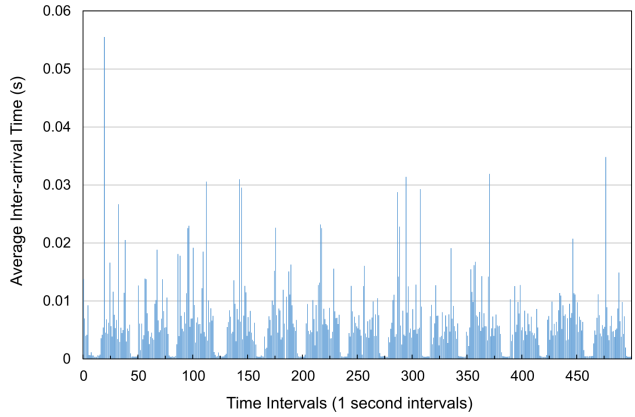


(b) Average inter-arrival time for *Packet-In* messages over intervals of one second

Fig. 4: Traffic generation for long-term load



(a) Controller load in percentage



(b) Average inter-arrival time for *Packet-In* messages over intervals of one second

Fig. 5: Traffic generation for short-term load

producing traffic with spikes-behavior, the inter-arrival time distribution of generated packets will alternate between small mean, equal to 0.001s, and large mean, equal to 0.2s. Those parameters are experimentally tuned to generate the intended load behaviors shown in Fig. 4a and Fig. 5a. In Fig. 5a, we clearly see fast spikes of the load. However, the heavy load situation persists in Fig. 4a representing the long-term load behavior.

The resulting average inter-arrival time for *Packet-In* messages is shown in Fig. 4b and Fig. 5b. For long-term load generation, the inter-arrival time of *Packet-In* messages does not vary frequently. Its mean value over the simulation time is equal to 828μs, close to the small distribution mean (0.001s) used to generate the traffic in each host. Those small values of inter-arrival times guarantee that the overload will last for a prolonged time. For instance, when the inter-arrival time of incoming packets increases between 300 and 450 seconds in the time axis, the controller load experienced some drops in response, reaching 84.6% in time interval 462s. Considering the short-term load case, the average inter-arrival time of *Packet-In* messages vary between 327μs and 55ms. This fast and periodic variation will generate the spikes of load depicted in Fig. 5a. When the average inter-arrival time is in order of microseconds, the load increases drastically. Then, it drops quickly with the rise of inter-arrival time of incoming packets. This makes sense because higher inter-arrival time means fewer incoming packets per time unit.

B. Dataset Description

The generated dataset contains nine features and 2344 instances (799 instances labeled as "long-term load", 1545 instances labeled as "short-term load"). The unbalanced nature of data reflects the reality, where the controller is often dealing with load spikes more than long-term load. Since the scales of different features differ, the data was normalized prior to the learning process. Besides, the data instances are shuffled randomly. We divided the dataset randomly into a training set and testing set. 70% of the data is selected for training, and the remaining is reserved for testing purposes.

C. Prediction Models and Learning Parameters Tuning

1) Support Vector Machine Model: Support Vector Machine (SVM) [10] has become popular in recent years for solving problems in classification, large-scale regression, and

novelty detection. SVM determines the decision boundary in high dimensional space that maximizes the margin between data points belonging to different classes using support vectors and margins defined by those support vectors. In practice, however, the class-conditional distributions may overlap. Thus, SVM tries to maximize the margin while softly penalizing points that lie on the wrong side of the margin boundary. A regularization coefficient $C > 0$ is introduced to control the trade-off between minimizing training errors and managing model complexity.

In our implementation, we used SVM with linear kernel. Thus, the decision function will have this form:

$$f(\overrightarrow{x_n}) = sign(\overrightarrow{w}.\overrightarrow{x_n} + b) \quad (1)$$

where $\overrightarrow{w}$ is a weight vector, b is scalar representing the bias and $\overrightarrow{x_n}$ is a normalized data instance.

We adopted 5-fold cross-validation for the parameters setting of the learning process. We retrained the SVM classifier by adjusting the parameters and chose the best parameter setting to test the model. Also, the parameters are tuned by trying a geometric sequence of the Regularization Parameter C from $1e-3$ to $1e3$ by a factor of 10. One strategy of tuning the Scaling Factor S is to retrieve the original kernel scale ks and use it as new kernel scale factor of the original. Therefore, we multiplied ks by the seven values from the same previous geometric sequence ranging from $1e-3$ to $1e3$. Finally, we found that the best parameter setting is obtained for $C = 1$ and $S = 0.122357$. We trained the classifier and applied the constructed model to predict the labels of the testing instances. After training process, the obtained number of support vectors is 140. The results are reported in the section V.

2) k-Nearest Neighbors Model: k-Nearest Neighbors [11] is a very simple algorithm but works incredibly well in practice. It classifies the patterns by finding the k data points that are closest in distance to a query point. The predicted class of the query point will be the result of majority votes of the nearest neighbors. The problem here is what distance metric to use and how to determine a good value for k, the number of neighbors. This parameter is usually determined experimentally. Starting with $k = 1$, we use a 5-fold cross validation to estimate the error rate of the classifier. This process can be repeated each time by incrementing k to allow for one more neighbor. The k value that gives the minimum error rate should be selected. Considering our case, we tried various distance metrics, and we found that "cosine similarity" gives the best performance. Besides, when we vary the number of neighbors from 1 to 30, we found that the minimum cross-validation error value is obtained for $k = 19$.

3) Naive Bayes Method: Naive Bayes [12] is a statistical classifier which performs probabilistic prediction of class membership probabilities. As a probabilistic framework, Naive Bayes considers each attribute and class label as random variables. Given a record with attributes $(A_1, A_2, ..., A_n)$, the goal is to predict class $C = 0$ or 1. Specifically, we want to find the value of C that maximizes $P(C|A_1, A_2, ..., A_n)$. The approach is to compute the posterior probability $P(C|A_1, A_2, ..., A_n)$ for all values of C using the Bayes Theorem. Thus, maximizing $P(C|A_1, A_2, ..., A_n)$ is equivalent to choosing the value of C that maximizes $P(A_1, A_2, ..., A_n|C)P(C)$. To estimate the likelihood, we assume the independence among attributes A_i when class is given:

$$P(A_1, A_2, ..., A_n|C) = P(A_1|C)P(A_2|C)...P(A_n|C) \quad (2)$$

A new point is classified to class C if $P(A_1|C)P(A_2|C)...P(A_n|C)$ is maximal among all other classes.

In the same way as previously mentioned, we used the testing set containing 703 testing data entries to infer the performance discussed in section V.

V. Performance Evaluation

For SVM Model, the classification accuracy for testing data is equal to 97.72%, which is a very reliable accuracy. SVM achieves the best performance among the three classification algorithms. A classification precision of 95.83% is achieved. The Area Under Curve (AUC value) is equal to 0.9910. The confusion matrix is illustrated in Table II. Out of 467 testing instances labeled as "Short-term load", the algorithm succeeded to predict the correct labels of 457 instances. For the data instances labeled as "Long-term load", only 6 entries are misclassified from the total of 236 instances. Table I summarizes the results in terms of accuracy, precision, recall and F1 values.

TABLE I: Performance evaluation of different classification algorithms

	SVM	k-nn	Naive Bayes
Accuracy	**97.72%**	96.16%	93.60%
Precision	**95.83%**	91.22 %	85.45%
Recall	97.46 %	**98.35%**	97.92%
F1	**96.64%**	94.65%	91.26%
AUC	**0.9910**	0.980551	0.965911

TABLE II: Confusion matrix for SVM. The rows indicate the true labels and the columns indicate the predicted ones.

		Predicted Class	
		Short-term Load	Long-term Load
Actual Class	Short-term Load	**457**	10
	Long-term Load	6	**230**

For the k-nearest neighbors classifier, the accuracy and precision values are slightly smaller than SVM. However, a classifier with an accuracy of 96.16% and a precision of 91.22% still highly precise. Using k-nearest neighbors model, we achieved an AUC equal to 0.980551.

Naive Bayes method achieves an accuracy and a precision of 93.60 % and 85.45 %, which are acceptable values even

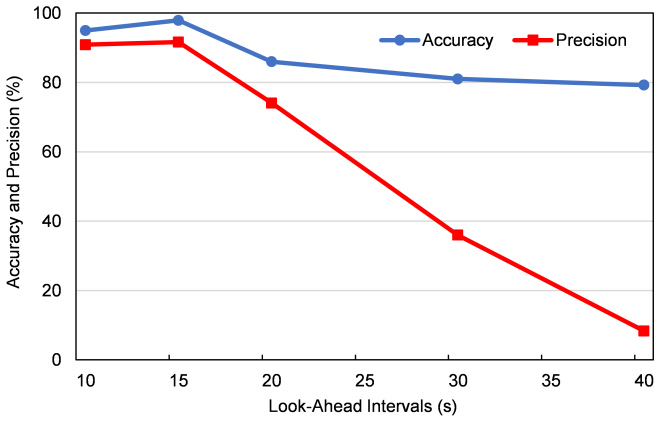


Fig. 6: Accuracy and Precision for real-time evaluation

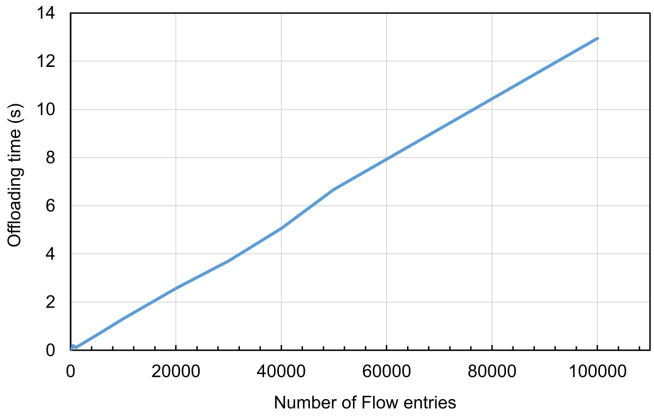


Fig. 7: Offloading delay for different number of flow entries using an OVS virtual switch and a POX controller

though they are lower than other methods. Naive Bayes misclassified 40 instances, originally labeled as "Short-term load". That is what causes the classifier performance to decrease.

Support vector machines methods are effective in high dimensional spaces and have a regularization parameter, which makes the learner avoids over-fitting. Moreover, the determination of the model parameters corresponds to a convex optimization problem, where any local solution represents a global optimum. All these factors explain the resulting performance of SVM compared to other methods.

For the real-time evaluation, we have used the same SDN simulated testbed that consists of 9 switches and 64 hosts. Now, after learning the decision function from the generated dataset, we have generated a hybrid traffic which contains both long-term load and spikes to test our framework in a real-time scenario. Since SVM model gave us the best performance, we included the corresponding generated decision function (see equation 1) in the prediction module within our controller.

When the controller load reaches the overload threshold (set to 80% in our case), the system immediately predicts the type of load, stores the result and launches a thread to track the load over a defined look-ahead interval. During this interval, if the load falls under the overload threshold, the traffic load is automatically considered as short-term. However, if the look-ahead interval expires and the load remains above the 80% threshold, we can definitely conclude that the controller is dealing with a long-term load. Finally, the true labels are compared with their corresponding predictions results. This process was repeated with the same decision function, but with different look-ahead intervals values. The results are depicted in Fig. 6. The accuracy and precision values diminish proportionally with higher look-ahead intervals. With the unbalanced nature of observations, the accuracy can be misleading. In this case, the precision is considered more accurate performance metric. In Fig. 6, the precision drops from 95% (when the look-ahead interval is equal to 10s) to 8.33% (when the look-ahead interval is equal to 40s). The best performance is obtained for the look-ahead interval equal to 15s. At this point, the accuracy and the precision are respectively equal to 97.93% and 91.67%. The obtained optimal look-ahead interval value is explained by the fact that the spikes of load durations, when exceeding the overload threshold, are between 7s and 11s. Here, the notion of "long-term load" or "short-term load" is strongly linked to the traffic patterns generated. Thus, the look-ahead interval should be set in a way to cover the total period of a load spike. In all experiments, the traffic manager module successfully offloaded the controller. We chose to let the controller take the offloading action after making two consecutive predictions of the long-term load for more stability and in order to mitigate the ping-pong effect resulting from switching between the reactive and proactive modes.

We further investigate the controller offloading delay by installing a static flow entries into the switches. We installed both Open vSwitch 2.3.2 virtual switch and POX controller into an Intel Xeon CPU X5550 2.67 GHz 8 cores machine with 40 GB memory. The data yielded by this experiment, depicted in Fig. 7, highlights a strong evidence that the offloading delay increases linearly with the number of flow entries. For 124 flow entries, the controller needs only 47.36ms to copy the static flow table into the switch. Meanwhile, the same action takes 12.94s for 100000 flow entries. Such delay in the order of seconds can significantly affect the overall network performance. Our proposed solution avoid such latency by installing proactive flow table (*Table 1*) into all switches. Thereby, the switches will not fetch the proactive flow entries from the controller for each long-term load alert. The latter had just to send a control packet to trigger the swap between reactive and proactive flow tables in the switches. Using the same topology for the online evaluation and 100Mbps links with 1ms delay, we measure the latency of the network with and without introducing our framework. We found that our scheme results in a negligible increase in the average network latency from 47.02ms to 50.45ms (the increase is equal to 3.43ms). This negligible latency is due to the small overhead of the control packets in addition to the delay of switching between *Table 0* and *Table 1* in the data-plane devices.

Independently from the number of flow entries, the proposed solution always keep the offloading delay negligible compared to the look-ahead intervals used. Furthermore, we investigate possible implications of our solution on the throughput by sending from and to each of the 64 hosts data streams of 300 MB. The resulting maximum link throughput (90 Mbps) remains unchanged. As a result, we conclude that our method does not have any implications on the network throughput.

VI. Related Work

Our design spans monitoring the SDN controller state, load type prediction, and traffic management, offering a general solution for SDN controller failure mitigation. Earlier research work in [13] introduced a framework for load demand prediction in cloud data center networks. The latter used neural networks and stochastic models, such as auto-regressive filters, to predict the future load and determine the optimal resource allocation by minimizing the energy consumed while maintaining required performance levels. Basically, this paper tackles reducing power consumption in cloud data centers, while our work targets controller offloading in SDN networks. In addition, this previous work considers a regression problem, which does not take into account the type of load.

CheetahFlow [14] predicts various communication pairs via Support Vector Machine and reroutes them to the non-congestion path efficiently by applying blocking island paradigm. However, unlike our proposed solution, CheetahFlow mainly targets avoiding congestion along a fixed path.

There has been much work on failure recovery [15], [2], [16]. The cited approaches introduce schemes based on failure detection and establishing preconfigured backup paths. Although those methods have succeeded to reduce the recovery time after failure detection significantly, they were limited to link-level failure and did not address the controller failure.

Zhang et al. [17] proposed rollback recovery in SDN by periodically recording the controller state during normal operation and storing it in non-volatile storage. Once a failure is detected, the controller is easily restored to a previous state. In [18], the authors have shed light on various techniques, such as reliable flooding, global snapshots, and replicated controllers, aiming to solve the SDN availability issues. They discussed the impact of possible failure in any network level and how to deal with it. In contrast, our work adopts a preventive approach to deal with failures.

VII. Conclusion

In this paper, we have employed prediction techniques to detect long-term load on POX controller. Our learning module was able to achieve an accuracy of 97.72% in the offline evaluation. Furthermore, we propose an offloading solution based on switching between reactive and proactive modes to relax the controller. Based on the learned decision function, the offloading process is fully automated and can be initiated by the controller without introducing extra latency. Besides, we conducted further experiments to validate our model in a real-time online evaluation scenario using look-ahead intervals to judge the predictions already made. The prediction module succeeded to differentiate long-term load from the short-term load with a success rate of 97.93%. These results provide a framework that can be trusted to automate the process of controller offloading. The proposed solution mitigates a potential crash of the controller due to overload, which may bring down the entire network.

For future work, we aim at extending our framework to reinforcement learning and dealing with the common problems related to such type of learning like convergence problem. In addition, we intend to test our solution on a larger scale and using other traffic patterns.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, Mar. 2008.

[2] N. L. M. v. Adrichem, B. J. v. Asten, and F. A. Kuipers, "Fast Recovery in Software-Defined Networks," in *Proceedings of the 2014 Third European Workshop on Software Defined Networks*, ser. EWSDN '14, 2014.

[3] P. A. Morreale and J. M. Anderson, *Software Defined Networking: Design and Deployment*. Boca Raton, FL, USA: CRC Press, Inc., 2014.

[4] A. Al-Shabibi and M. McCauley, "POX Wiki," *Retrieved from Stanford University: https://openflow.stanford.edu/display/ONL/POX+Wiki*, 2013.

[5] psutil Package Index. https://pypi.python.org/pypi/psutil.

[6] W. Braun and M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," *Future Internet*, vol. 6, 2014.

[7] Mininet Network Emulator. http://mininet.org/.

[8] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "ElastiCon: An Elastic Distributed SDN Controller," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14, 2014.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, 2010.

[10] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[11] T. Cover and P. Hart, "Nearest Neighbor Pattern Classification," *IEEE Trans. Inf. Theor.*, vol. 13, no. 1, Sep. 2006.

[12] R. Caruana and A. Niculescu-Mizil, "An Empirical Comparison of Supervised Learning Algorithms," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06, 2006.

[13] J. J. Prevost, K. Nagothu, B. Kelley, and M. Jamshidi, "Prediction of cloud data center networks loads using stochastic and neural models," in *System of Systems Engineering (SoSE), 2011 6th International Conference*, 2011.

[14] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "Cheetahflow: Towards low latency software-defined network," in *2014 IEEE International Conference on Communications (ICC)*, 2014.

[15] A. Capone, C. Cascone, A. Q. T. Nguyen, and B. Sans, "Detour planning for fast and reliable failure recovery in SDN with OpenState," in *Design of Reliable Communication Networks (DRCN), 2015 11th International Conference*, 2015.

[16] S. Paris, G. S. Paschos, and J. Leguay, "Dynamic control for failure recovery and flow reconfiguration in SDN," in *2016 12th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2016.

[17] Y. Zhang, N. Beheshti, and R. Manghirmalani, "NetRevert: Rollback Recovery in SDN," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, 2014.

[18] A. Akella and A. Krishnamurthy, "A Highly Available Software Defined Fabric," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIII, 2014.