Assisting Delay and Bandwidth Sensitive Applications in a Self-Driving Network

Sharat Chandra Madanapalli
University of New South Wales
Sydney, Australia
sharat.madanapalli@student.unsw.edu.au

Hassan Habibi Gharakheili
University of New South Wales
Sydney, Australia
h.habibi@unsw.edu.au

Vijay Sivaraman
University of New South Wales
Sydney, Australia
vijay@unsw.edu.au

ABSTRACT

Packet networks are agnostic to applications, which have served to keep the Internet infrastructure simple and scalable over the past several decades. However, the best-effort model is now seen as an inhibitor to meeting user experience expectations for the diverse applications such as streaming video, gaming, browsing, and social media. Current methods for prioritization of certain application types are static, and do not react to changes in network conditions or user experience. We envisage a self-driving network that is able to continuously monitor user experience and intervenes to assist applications as and when needed. Our contributions are: (1) We propose a self-driving network architecture that directly measures, optimizes, and dynamically controls application performance. We develop a method to measure and model application state in real-time using network behavior data. (2) We apply our framework to two representative applications, video streaming and gaming, and show how the network can detect application deterioration in terms of playback buffers and ping latency respectively, and apply remedial action to improve application performance without requiring any explicit signaling.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Network performance modeling**; **Programming interfaces**; **Network measurement**;

1 INTRODUCTION

User-perceived application experience is of paramount importance in broadband as well as cellular networks, be it for video streaming, teleconferencing, gaming, or web-browsing. The best-effort delivery model of the Internet makes it challenging for Application/Content Providers to maintain user experience, requiring them to implement complex methods such as buffering, rate adaptation, dynamic CDN selection, and error-correction to combat unpredictable network conditions. Network operators, also eager to provide better user experience over their congested networks, often employ middle-boxes to classify network traffic and apply prioritization policies. However, these policies tend to be static and applied on a per-traffic-class basis, with the benefits to individual applications being unclear, while also potentially being wasteful in resources.

It is envisaged that Self-Driving Networks of the future will be able to address this problem through a combination of continuous network measurement, automated inferencing of application performance, and programmatic control to protect user experience. This paper represents a first step towards this goal, leveraging recent developments in programmable networks and machine learning. Our aim is to show that the network need not be manually pre-configured for resource sharing amongst applications; instead, it can autonomously deduce application experience at run-time, and provide assistance as and when required to specific traffic streams, thereby restoring user experience in a self-driving manner (aka without any explicit signalling).

We begin by outlining the architecture of our system that uses a trained machine to dynamically deduce the application state and apply corrective actions when application performance deteriorates to an unacceptable state (§2). We then prototype our system and apply our state inference methods to two applications, namely Netflix video streaming (that is sensitive to network bandwidth) and Gaming (that is sensitive to network latency), and show that network assistance can protect application experience in a timely manner in the face of changing traffic conditions, without requiring any explicit signalling (§3). Our work paves the way towards a network that can self-manage user experience without human configuration.

2 SYSTEM ARCHITECTURE AND DESIGN

To realize the self-driven network assistance, three tasks are needed to be performed automatically and sequentially: (a) "measurement", (b) "analysis and inference", and (c) "control", as shown by a closed-loop in Fig. 1.

In our architecture, a programmable switch is placed in-line on the link between the access network and the Internet. In a typical ISP network, this link is the bottleneck (and hence the right place to do traffic shaping) as it multiplexes subscribers to a limited backhaul capacity. First, traffic of a desired application (*e.g.,* video streaming) is mirrored to *FlowFetch* module which exports flow-level counters (measurement) to a classifier model. Next, the network telemetry data is used by a classification model to determine the current experience state of an application (analysis and inference) which is sent to the state-machine module. If a critical event of the application behavior (*e.g.,* video re-buffering) is detected by the state-machine (arising due to a transition among states), an assist request is sent to *actor* module. Lastly, the actor

NetAI '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3341312

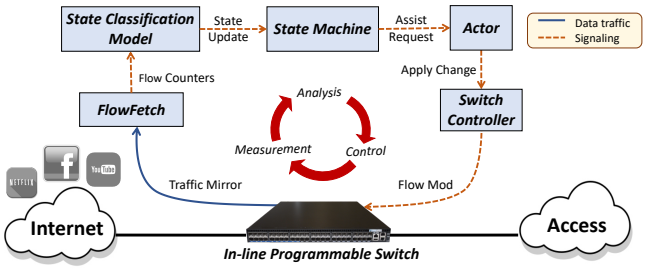


Figure 1: System architecture of assisting applications.

requests changes (*e.g.,* queue provision) to the switch controller which in turn sends *FlowMod* and *Queue Configuration* messages to the switch, executing the corresponding action in order to elevate the application's performance.

In order to automatically infer the performance (directly affecting the quality-of-experience) of an application, we model its network behavior using a state machine. Every application begins in state "*start*", when its first packet is seen on the network. Subsequently, it transitions into different states depending on the type of application. We illustrate in Fig. 2 an example of performance state-machine for a video streaming application as a sequence of states: *init* → *buffering* → *stable* → *stable* → *depleting* → *terminate*. Depending upon the policies of network operator for video streaming, a required action can be taken automatically at any of these states (*e.g.,* when it is found at depleting state, a minimum amount of bandwidth is provisioned to corresponding flows, till the application comes back to its stable state).

2.1 Data Collection

To realize such a system architecture, we first need to acquire network activity data for the applications of interest, labeled by their behavioral states. This enables the network operator to train classifiers and build state machines which can infer application behavior without the need of any explicit signals from either the application provider or client. We have developed a tool for generating application dataset – the high-level architecture of our tool is shown in Fig. 3. It consists of three main components namely *Orchestrator*, *Application* player, and *FlowFetch*. The orchestrator performs two tasks: (a) initiates and runs the application instance, and keeps track of its behavioral state, and (b) signals to the FlowFetch for recording the corresponding network activities (*i.e.,* time-trace of flow counters). The optional network conditioner module can be used to impose (synthetic) network conditions such as limited bandwidth or extra delays to capture various behaviors of the application.

Labeling Application States: As mentioned earlier, important application states need to be labeled since they help the state machine determine when a network assist is required. For example, stall/buffer-depletion, high latency, and lag/jitter are crucial states for video streaming, online gaming, and teleconferencing applications, respectively. Having identified the critical behavioral states of an application, the orchestrator is configured to detect and label these states. In prior research, authors have used GUI interaction tools [1], javascript APIs [6] and web automation tools (*e.g.,* Selenium library) [14] to automatically interact with the application and capture its behavior.

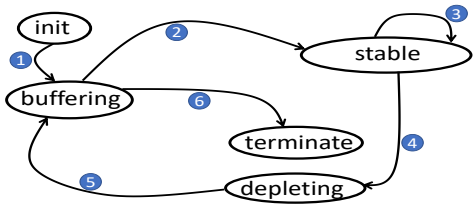


Figure 2: Example of performance states transition for a video streaming application.

Measuring Network Activity: The network activity of applications can be measured in several ways ranging from a basic packet capture (expensive recording and processing) to proprietary HTTP loggers combined with proxies (limited scalability). We propose a method that strikes a balance by capturing flow-level activity at configurable granularity using conditional counters. This method stores less data due to aggregation on a per-flow basis, and can be deployed using hardware accelerators like DPDK or be implemented in the data-plane using P4 [4].

We have built a tool called *FlowFetch* in *Golang*, which records flow-level activity by capturing packets from a network interface. By a flow, we mean a transport-level TCP connection or UDP stream identified by a unique 5-tuple consisting of *source IP*, *source port*, *destination IP*, *destination port* and *protocol*. Each flow has a set of conditional counters associated with it – if an arriving packet satisfies the condition, then the corresponding counter increments by a defined value. For example, a counter to track the number of outgoing packets greater than a volume-threshold (important to identify video-streaming experience [18]) can be expressed using a volume-based condition on each packet – if satisfied, the counter increments by one. Similarly, other basic counters to track volume of a flow can be defined to increment by the byte-size of its arriving packet (without any explicit condition). The set of counters are exported at a configurable granularity (*e.g.,* every 100 ms) – it depends on the complexity of application behavior.

Application data record: Our tool generates two time-series data for each application "run". The first one consists of application states (*e.g.,* for a video streaming, *buffering* for first 10 seconds, followed by 40 seconds of *stable* and then *termination*). The second one contains the corresponding flow-counters collected from the network activity of the application at the configured granularity (say, tracking byte and packet counters every 100 ms).

2.2 State Classification and State Machine

The training set, consisting of multiple labeled application runs, is used to train and generate a model which will classify the application state given its network activity patterns. Certain states can be identified from prior knowledge of application (*e.g.,* video streaming always starts in buffering state). For other states which require pattern recognition on the network activity, it requires to extract important traffic attributes computed over a time window (say 10 seconds) and build an ML-based classifier. Thus, the State Classifier is composed of rule-based and/or ML-based models which together classify application's current state that is passed as an update to the state-machine, shown in Fig. 1.

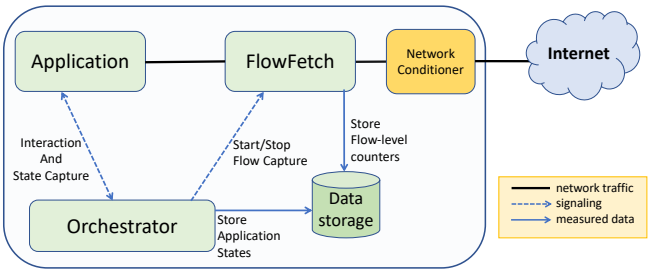


Figure 3: Our data collection tool.

State Machine Generation: The state machine of the application is generated using the behavioral state labels available in the dataset along with corresponding transitions. As explained earlier, the orchestrator continuously labels states of the application as it is running which can be used to capture the transitions which happen in regular usage of that particular application We note that all possible transitions may not occur for an application during data collection, and hence we may need to edit the state machine manually.

Experience-Critical Events Annotation: The state machine that models application behavior needs to be annotated with Experience Critical (EC) events that require assistance from the network. When such events occur within the state machine, a notification is sent out to the *Actor* module (in Fig. 1). There might be multiple types of EC events. For instance, a transition to "bad" state or spending long time in a certain state indicate QoE impairments, and thus are considered as EC events. To exemplify, in a video-streaming application, a transition from buffering state to stall state is a QoE impairment and thus an EC event. Further, a prolonged buffering state is also not desirable as the buffer level is very close to playback. This EC event can be annotated using a timer on the buffering state. Such events trigger a notification to the *Actor* (by the state machine) while receiving the real-time state updates from *State Classifier*.

2.3 Actor: Enhancing Experience

Upon receiving assist requests from the State Machine, the Actor is responsible for enhancing the performance of the application via interaction with the Switch Controller. Typically the application's poor performance can be alleviated by prioritizing its traffic over others in a congested scenario. The choice of what kind of applications to prioritize would depend on the policies of the ISPs and their customers' requirements. This paper focuses on providing a mechanism to enhance the experience. This can be done in multiple ways including but not limited to: (a) strict priority queues where priority levels are assigned depending on the severity of the assist requests, (b) weighted queues where more bandwidth is provisioned to applications in need, or (c) use packet coloring and assigning different drop probabilities to different colors, *e.g.,* a two-rate three-color WRED mechanism [11]. Assisting methods are confined by the capability of the programmable switching hardware and the APIs it exposes. Nonetheless, the actor needs to request the switch controller to map the flow(s) of the application to the prioritizing primitive (changing queues or coloring using meters, etc.).

Note that the assisted application needs to be de-assisted after certain time for two reasons: (a) to make room for other applications in need (to be prioritized), and (b) the performance (QoE) of the assisted application has already improved. However, doing so might cause the application to suffer again and thus results in performance oscillation (*i.e.,* a loop between assistance and de-assistance). To overcome this, we propose that the de-assisting policy could be defined by the network operators using the network load (*i.e.,* link utilization). A primitive policy could be to de-assist an application when the total link utilization is below a threshold, say, 70%. This would ensure that the de-assisted application has enough resources to (at least) maintain the experience, if not improve it. These policies could be further matured depending on the number and type of applications supported and also various priority levels defined by the operator.

3 ASSISTING SENSITIVE APPLICATIONS

We now implement our framework and assist two applications, namely, Netflix (representative of bandwidth sensitive video streaming) and ping (representative of latency sensitive online gaming). Although ping is relatively simple when compared to actual gaming applications, we note that the requirement of the application still remains the same, *i.e.,* low latency. In what follows next, we describe our measurements, state classification models of the applications behavior, and subsequently elaborate on assistance methodology which enhances the user experience.

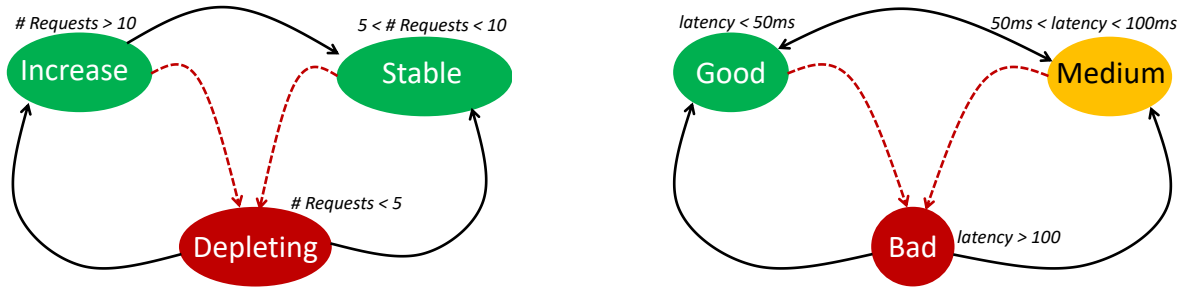
3.1 Dataset and State Classification

Dataset: We use our data collection tool, shown in Fig. 3, to orchestrate sessions of Netflix video streaming and ping as follows. For Netflix, we use a web client on a chrome browser (*i.e.,* the Application block in Fig. 3) which is controlled by a python script (*i.e.,* the Orchestrator) using Selenium web automation library. The network data is collected using our FlowFetch tool described in §2. The orchestrator also captures user experience by enabling a menu that offers multiple video playback metrics. Of them, we consider *Buffer Health* (in seconds and bytes) and *Bitrate* (denoting the quality of video playback) to understand experience.

We detect bad experience in terms of buffer depletion which often also leads to bitrate degradation as the video client adapts to poor network conditions. Prior studies [6, 14, 16] have found that chunks transfer in a flow starts by an upstream request packet of large size (other small upstream packets are generally ACKs for the contents received). To capture such transfers, we employ three conditional counters: "ByteCount" transferred both downstream and upstream, "PacketCount" both downstream and upstream, and "RequestCount" for upstream packets greater than a threshold (say, 500 Bytes). We collected these flow counters every 100 ms, over 6 hours worth of Netflix video playback.

For gaming (represented by ping), the experience metric, latency, is measured both at the client-end and in the network using the FlowFetch. On the client, we have built a python wrapper which reads the output of the ping utility. On the network, the FlowFetch keeps track of the ICMPv4 flow using the 4-tuple sourceIP, destIP, Protocol and ICMP ID. It calculates the latency by subtracting the timestamp in request and response packets. We note that the latency measured from network is slightly lower than measured on client as it does not include the latency in the access network.

Classifying Buffer-State for Video Streaming: In our dataset, we have observed that Netflix client: (a) in buffer-stable state, it



(a) Buffer-based state machine for video streaming.

(b) Latency-based state machine for online gaming.

Figure 4: State machine of sensitive applications.

requests one video chunk every 4 seconds and an audio chunk every 16 seconds, (b) in buffer-increase state, it requests contents at a rate faster than playback, and (c) in buffer-depleting state, it requests less number of chunks than being played. Given this knowledge of Netflix streaming, we devise a decision tree-based classifier for the count of requests over a window of 20 seconds. To maintain the buffer level over this window, the Netflix client should ideally request for 7 chunks, *i.e.,* 5 video chunks (of 4 second duration) and 2 audio chunks (of 16 second duration). Thus, this naturally indicates a threshold to detect buffer increase (>7 chunk requests) and buffer depletion (<7 chunk requests). However, in practice, deviations from ideal behaviour are observed – we, therefore, built our decision three by slightly broadening the threshold values as depicted in Fig. 4(a). Therefore, using the uplink request packet measurements, we are able to build a simple classifier which detects the user experience in terms of buffer health. We acknowledge that it is needed to have a combination of counters and statistical techniques to isolate chunk data and extract features to predict bitrate switches and startup delay, as studied in [6], to ideally capture the experience of video streaming. However, the scope of this paper is limited to develop a framework for automatic assistance of applications by acting upon triggers detected by real-time network measurement. The framework can incorporate any number states reported by sophisticated models and assist the applications when experience-critical events are detected.

Classifying Latency-State for Gaming: In multiplayer online gaming applications, an important experience metric is latency which represents the end-to-end delay from the gaming client to either the servers or other clients (*i.e.,* peers). The latency (also referred to as "lag", "ping rate", or simply "ping"), arises by the distance between end-hosts (static), and congestion in the network (dynamic) which causes packets to wait in queues. Our solution attempts to alleviate the gaming performance by reducing the delay caused in congested networks. Although the latency requirements differ depending on the type of game being played, typically at least a latency of under 100 ms is desired to have a smooth experience [17] – although top gamers prefer a latency of up to 50 ms. Using the latency measurements, we define three states of gaming, *i.e.,* "good" (0-50 ms), "medium" (50-100 ms) and "bad" (>100 ms), as depicted in Fig. 4(b) – these latency ranges were reported by players of various popular gaming applications such as Fortnite, Apex Legends and CSGO. Any transition into the bad state triggers a notification requesting an assist to the actor.

3.2 Performance Evaluation

With state machines and classification models built, we now demonstrate the efficacy of our framework by implementing the end-to-end system from measurement to action in a self-driving network. Our lab setup consists of a host on the access network running Ubuntu 16.04 with a quad-core i5 CPU and 4 GB of RAM. The access network is connected to the Internet via an inline SDN enabled switch (*i.e.,* Noviflow model 2116). On the switch, we have capped the maximum bandwidth of the ports at 10 Mbps. We have pre-configured three queues (*i.e.,* A, B, and C) on two ports (*i.e.,* P1: upstream to the Internet and P2: downstream to the access) which are used to shape the traffic, assisting sensitive applications. Queue A, is the lowest-priority default queue for all traffic and is unbounded (though maximum is still 10 Mbps). Queue B has medium priority and Queue C has the highest priority. This means that packets of the queue C are served first, followed by the queue B, and then the queue A.

We now set up a scenario with three applications – Netflix client on Chrome browser representing video streaming application, `ping` utility representing gaming, and `iperf` to create cross-traffic on the link. First, we use the applications without any assistance wherein all network traffic is served by one queue without prioritizing any traffic (*i.e.,* best-effort) – performance of applications is shown in Fig. 5.

The flow of events is as follows. At t=0, we start a ping to 8.8.8.8 – this traffic persists during the entire experiment (400 seconds). At t=10, we launch the chrome browser and log in to Netflix. We observe that ping latency (shown by solid orange lines), which is initially at around 2 ms, starts increasing to 100 ms once the user logs into Netflix. The user, loads a Netflix movie ("Pacific Rim") and starts playing it at t = 30. From this point onward, we observe that the ping latency rises up to 300 ms, and Netflix requests chunks and transfers contents at its peak rates (purple lines) – the link utilization hits 100%, as shown by solid black lines in the bottom plot. On the Netflix client, we see that the buffer-health is increasing slowly (solid blue lines), and the client selects the highest available bitrate of 2560 Kbps (dashed red lines).

At t = 70, we initiate a downstream flow of UDP traffic with a max rate of 9 Mbps using the iperf tool to create congestion. We immediately notice that both sensitive applications start to suffer with the link utilization remains at 100%. The buffer level on the client starts depleting from 110 to 100 after which the Netflix client

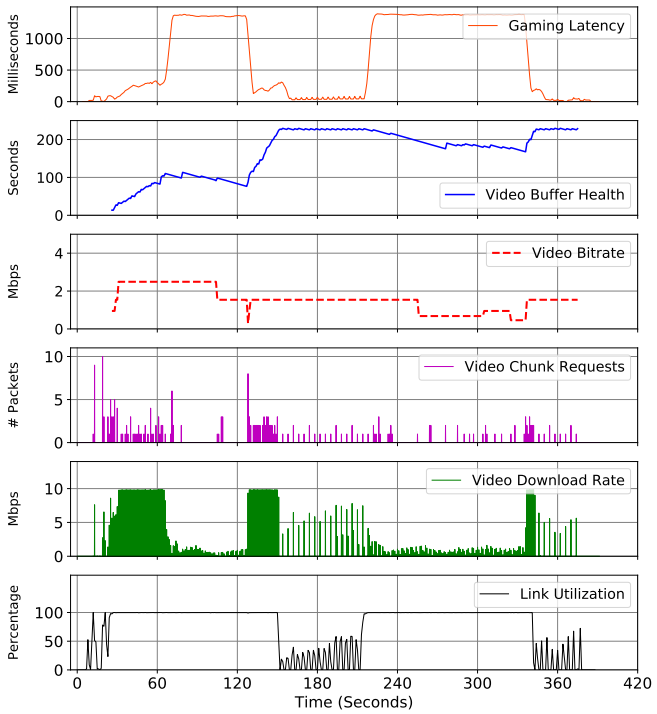


Figure 5: Performance of sensitive applications without network assistance.

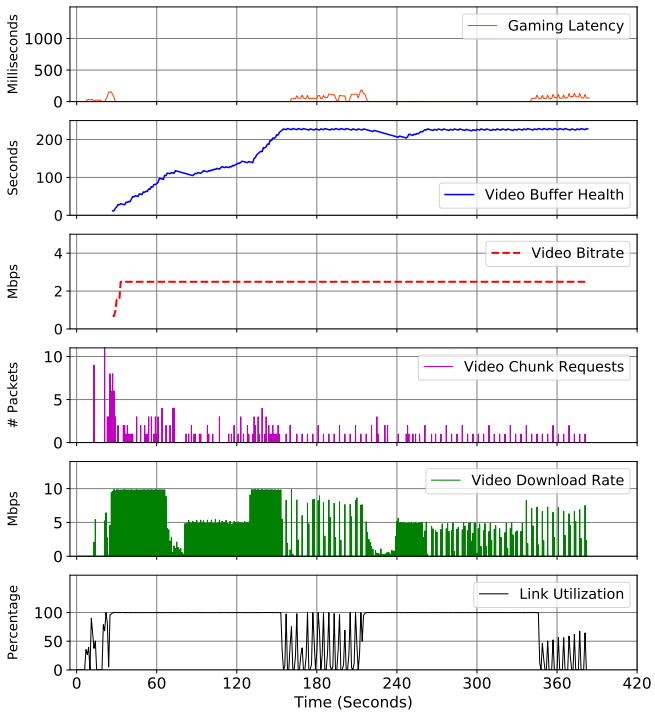


Figure 6: Performance of sensitive applications with network assistance.

switches to a lower video bitrate. The video client does not request enough chunks as shown by a gap in the purple curve. It only starts sending out requests again at around t = 100, when the video bitrate dropped. The ping suffers even more and the latency reaches to 1300-1400 ms. Once the download finishes at t=130, we notice that the video starts to ramp up its buffers, but at a lower bitrate (because it just detected poor network conditions) and reaches the stable buffer value of 4-minute at around t = 140. The ping also displays a better performance with the latency between 300-400 ms (during video buffering), but it gets even better dropping to 100 ms when the video enters into its stable state.

At t = 220, we initiate another UDP traffic stream which makes the applications suffer again. This time, we notice that video transitions into buffer-depleting state from buffer-stable state. Again we observe gaps in video chunk requests, clearly indicating decrease in buffer, and subsequently the video download rate falls below 2 Mbps. Ping reacts similarly by reporting the latency of over a second. Note that during this congestion period, the video client switches its resolution three times, with two events of lowering the bitrate that result in poor user experience. Upon completion of the download, we note that both sensitive applications display an acceptable performance.

For our second scenario, we demonstrate the automatic assistance from a self-driving network which continuously monitors the applications states and intervenes whenever needed. We use the state machines and classification models explained earlier in §3.1 to capture the application states and trigger assist requests to the Actor. The Actor, assists the application by shifting its flows to a separate priority queue (*i.e.,* queue B or C) and de-assists the application by putting it back into the default queue A. In our prototype, we allocate the highest priority queue C to gaming applications, which will ensure reduction in latencies. The video streaming flows, when require assistance, are served by the queue B. Note that we configured the max-rate on the queue B at 4 Mbps – when exceeded, the priority of exceeded packets becomes equal to of the queue A. In other words, traffic will be prioritized as long as it consumes less than 4 Mbps beyond which it needs to compete with traffic on the queue A. The need for such a mechanism is due to the elastic nature of video streaming application, it will take up as much bandwidth as available. If streaming video is given a pure priority over the default traffic, it will throttle the default traffic to almost 0.

With these settings, we notice a significant improvement in the experience of both sensitive applications as shown in Fig. 6. As described earlier, we start with only ping where it reports a very low latency (*i.e.,* <5 ms). Logging into Netflix at t = 20 causes ping latency to go beyond 100 ms. First, the classifier finds the gaming application in the medium state (a transition from the good state) which results in a request for assistance. The actor elevates the ping experience by shifting its flow to the queue C. Following this action, we observe that the ping latency immediately drops back to around 2 ms. Meanwhile, the video stream starts and is detected to be in the buffer-increase state, given the large number of chunk requests. At t = 70, when the UDP iperf traffic (*i.e.,* download) is introduced, we note that the buffer depletes and no chunk requests are sent for a few seconds. Our classifier now detects the video state at buffer-depleting which initiates an assist request. Following it,

all flows corresponding to the video stream are pushed to the queue B. Upon assisting the video, we observe that buffer starts to rise again. Note that the buffer rises slower this time because Netflix application gets about 4-5 Mbps due to the queue configuration. Nonetheless, this ensures that the video performs better without heavily throttling the download on the default queue. When the download stops, the buffer steeply rises till it enters into the stable state. At this point, latency values go up to 100ms. This happens due to de-assist policy which pushes back the applications' traffic to the default queue as the link utilization falls below the 70% threshold (for video) and 40% threshold (for gaming) respectively.

At t= 220, the iperf generates traffic again. As soon as the ping values go above 100 ms, the ping flow is assisted, and thus its performance is improved. Similarly, the video application is re-assisted as it is found in the buffer-depleting state. This time the video buffer fills up very quickly, taking the application back to its stable state. Note that the video stream is not de-assisted since the iperf traffic is still present (*i.e.,* high link utilization), and the video download rate is capped at around 4-5 Mbps. Once the download traffic subsides (and thus the link utilization drops), both video stream and ping traffic are pushed back to the default queue A.

In summary, with automatic assistance from the network, despite of heavy congestion the user experience of sensitive applications was maintained by detecting critical experience impairment events using data-driven classification model and behavioral state machine. We clearly see that the video stream was always played in the highest available bitrate with a reasonable buffer-health, and the gaming application always had a latency under 100 ms.

4 RELATED WORK

Prior studies measure user experience for sensitive applications (video streaming [14], gaming [12] and interactive calls [5]), but they do not attempt to enhance the quality of experience. Those solutions can be readily used in our scheme by providing means to classify the state of application performance. Several approaches have leveraged network programmability offered by SDN to dynamically respond to QoE requirements and improve the end-user experience. Works in [8] and [10] propose interactions between applications and the network via APIs. This interaction can be used for various purposes including separating traffic on priority lanes to improve user experience as suggested in [10]. Although such an approach would be ideal to enhance user experience by receiving triggers from the application, realizing such method seems challenging since today application/content providers typically do not cooperate with network provider and even if they do, a mature standard of implementation is yet to come [13].

Multiple methods have used the paradigm of SDN to improve the quality of applications without any active participation from the applications. One class of approaches have looked at improving general QoS features (*e.g.,* packet loss, delay, jitter) by using resource management and path planning [7, 15]. These proposals work in the context where every network element is SDN-enabled which is not yet a reality for many ISP networks. The other class of approaches came up with designs and methodologies to improve QoE of individual applications, mostly video streaming. SDNHAS [3] and SDNDASH [2] propose an SDN-enabled architecture that enhances the QoE of HTTP Adaptive Video Streaming applications. However, their system design requires modifications in the video streaming application to communicate with the controller. Work in [9] improves QoE fairness by using an OpenFlow switch at the point of congestion similar to our proposed architecture. However, it attempts to read Manifest Files from the network traffic which would only work for unencrypted traffic.

5 CONCLUSION

Sensitive applications need dynamic prioritization from a self-driving network that reacts to changes in user experience. In this paper, we have proposed an architecture for continuous monitoring and dynamic control over the performance of sensitive applications. We have developed data-driven models for the behavioral state of applications in real-time. Lastly, we showed how our scheme is able to detect performance deterioration and take remedial action for two popular sensitive applications, video streaming and gaming.

REFERENCES

[1] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan. 2014. Prometheus: Toward quality-of-experience estimation for mobile apps from passive network measurements. In *Proc. ACM HotMobile*. Santa Barbara, CA, USA.

[2] A. Bentaleb, A. C. Begen, and R. Zimmermann. 2016. SDNDASH: Improving QoE of HTTP Adaptive Streaming Using Software Defined Networking. In *Proc. ACM Multimedia*. Amsterdam, Netherlands.

[3] A. Bentaleb, A. C Begen, R. Zimmermann, and S. Harous. 2017. SDNHAS: An SDN-enabled architecture to optimize QoE in HTTP adaptive streaming. *IEEE Transactions on Multimedia* 19, 10 (2017), 2136–2151.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[5] S. Chen, C. Chu, S. Yeh, H. Chu, and P. Huang. 2014. Modeling the QoE of Rate Changes in Skype/SILK VoIP Calls. *IEEE/ACM Transactions on Networking* 22, 6 (Dec 2014), 1781–1793.

[6] G. Dimopoulos, I. Leontiadis, P. Barlet-Ros, and K. Papagiannaki. 2016. Measuring video QoE from encrypted traffic. In *Proc. ACM IMC*. Santa Monica, CA, USA.

[7] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp. 2012. OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks. In *Proc. IEEE APSIPA*. Hollywood, CA, USA.

[8] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. 2013. Participatory networking: An API for application control of SDNs. In *Proc. ACM SIGCOMM*. Hong Kong, China.

[9] P. Georgopoulos et al. 2013. Towards network-wide QoE fairness using openflow-assisted adaptive video streaming. In *Proc. ACM FhMN*. Hong Kong, China.

[10] H. Habibi Gharakheili. 2017. *The Role of SDN in Broadband Networks*. Springer.

[11] J. Heinanen and R. Guerin. 1999. A Two Rate Three Color Marker. (1999).

[12] E. Howard, C. Cooper, M. P. Wittie, S. Swinford, and Q. Yang. 2014. Cascading Impact of Lag on Quality of Experience in Cooperative Multiplayer Games. In *Proc. IEEE NetGames*. Nagoya, Japan.

[13] J. Jiang, X. Liu, V. Sekar, I. Stoica, and H. Zhang. 2014. EONA: Experience-Oriented Network Architecture. In *Proc. ACM HotNets*. Los Angeles, CA, USA.

[14] T. Mangla, E. Halepovic, M. Ammar, and E. Zegura. 2018. eMIMIC: Estimating HTTP-based Video QoE Metrics from Encrypted Network Traffic. In *IEEE TMA*. Vienna, Austria.

[15] F. Ongaro, E. Cerqueira, L. Foschini, A. Corradi, and M. Gerla. 2015. Enhancing the quality level support for real-time multimedia applications in software-defined networks. In *Proc. IEEE ICNC*. Garden Grove, CA, USA.

[16] I. Orsolic, D. Pevec, M. Suznjevic, and L. Skorin-Kapov. 2017. A machine learning approach to classifying YouTube QoE based on encrypted network traffic. *Springer, Multimedia tools and applications* 76, 21 (2017), 22267–22301.

[17] R. Presser. 2018. The Importance of Latency in Online Gaming. https://bit.ly/2GdQXrB.

[18] D. Tsilimantos, T. Karagkioules, and S. Valentin. 2018. Classifying flows and buffer state for youtube's HTTP adaptive streaming service in mobile networks. In *Proc. ACM MMSys*. Amsterdam, Netherlands.