Design and Implementation of TCP-Friendly Meters in P4 Switches

Shie-Yuan Wang®, Senior Member, IEEE, Hsien-Wen Hu, Member, IEEE, and Yi-Bing Lin, Fellow, IEEE, ACM

Abstract—Metering the traffic of a flow and dropping the portion of traffic that exceeds the target rate set by the meter is very important to provide quality of service (QoS) in a network. Most existing switches on the market provide the meter function. Although these meters regulate the traffic of a UDP flow very well, according to our tests, they are not TCP-friendly. For example, we found that a TCP flow passing the meter of a commercial switch can only achieve about 10% of the target rate. Based on our studies, we found that this poor performance is due to the bad interactions between the TCP congestion control and the meter function. In this paper, we design and implement a TCP-friendly meter in the packet processing pipelines of a P4 switch. Experimental results show that our meter regulates a TCP flow very well and can maintain its achieved rate within 5% of the target rate. Compared with the meters in several hardware switches, our meter improves the achieved rate of a TCP flow by almost 85% of the target rate.

Index Terms—Meter, P4, QoS, TCP.

I. INTRODUCTION

QuaLITY of service (QoS) for traffic flows in a network is an essential issue. To save costs, network operators have abandoned the concept of using separate physical networks to carry network traffic of different QoS requirements. Instead, a single physical network is used to carry all types of traffic [1]. However, the traffic of some applications may consume too much bandwidth in a network, and other applications may not be allocated enough network bandwidth and thus have bad quality of experiences (QoE).

By using a QoS scheme, the network operator can control the amount of network bandwidth allocated to different types of network traffic in various levels. In this way, QoS can be guaranteed to traffic flows with different QoS requirements. A QoS scheme cannot increase the amount of network bandwidth. Rather, it allows network bandwidth to be shared in a non-equal manner. The first step in controlling the allocation of network bandwidth is having the capability to limit the rate of a flow that has exceeded the rate specified in the service contract. To achieve this capability, a meter scheme is deployed to limit the rate of a flow in a switch, which measures the arrival rate of a flow and assigns different processing priorities to its packets according to its current rate. With such a scheme, the portion of packets that exceed the specified rate for the flow are marked with low priorities and they may be dropped by the meter. Mostly, a meter (e.g., [2], [3]) is configured with a target rate and the size of a token bucket, which limits the maximum size of bursts allowed by the meter.

In the past, because switch platforms were closed and only switch vendors could implement QoS schemes inside a switch, researchers could not implement their QoS schemes in a switch to see how well these schemes regulate real network traffic. As a result, most metering studies could only use mathematical models to analyze the behavior of proposed meter algorithms or use simulations to show their effects (e.g., [4], [9], [10], [13], [14]). Recently, the P4 (Programming Protocol-Independent Packet Processor) programming language ([15], [16]) has been proposed to program the data plane of hardware switches so that developers can design and implement their algorithms to process packets in the pipelines of a hardware switch. With this capability, one can study the performance of QoS schemes by designing and implementing them in a P4 hardware switch and experimenting with their internal designs and parameter values to see how well they regulate network traffic.

We have tested the meters provided in three commercial hardware switches [23], [26], [27]. We found that these meters can regulate a UDP flow very well. That is, the achieved rate of a UDP flow is very close to the rate set by the meter. However, we found that if a TCP flow passes through the meter in the Netgear switch [26], the achieved rate of a TCP flow is very low and is only about 10% of the target rate set by the meter. We obtained these results by using 128 KB as the bucket size for the meter. The bucket size of a meter controls the acceptable burst size of the flow. The range of bucket size that can be varied on this switch is between 1 KB and 128 KB [14], and we found that using a bucket size smaller than 128 KB would further lower the achieved rate of a TCP flow. Because using a large bucket size for a meter is bad for network congestion control (this is because when the bursty traffic of multiple flows arrive at a switch, they are more likely to be dropped by the switch due to buffer overflow), we decided to use 128 KB as the bucket size for the meters in the other two switches [23], [27] and measured the achieved rate of a TCP flow regulated by these meters. Experimental results showed that the results were about the same for all of the three tested switches.

Based on our detailed studies, we found that such a problem is caused by the bad interactions between the meter algorithm and the TCP congestion control. In this paper, we design and implement a P4-based TCP-friendly meter scheme to enable
Fig. 1. The architecture of a spine-leaf network in which meters are used in rack switches to support QoS in the network.

the achieved rate of a TCP flow to approach the target rate set by a meter. In our scheme, we integrate a single rate two color meter algorithm, the Explicit Congestion Notification (ECN) scheme [5], the Random Early Detection (RED) scheme [6], and an adaptive control scheme [7] and implement them in a P4 programmable hardware switch [23]. With these mechanisms, our meter scheme enables the achieved rate of a TCP flow to closely approach the rate set by the meter and the difference is less than 5% of the target rate. Using the same bucket size of 128 KB, compared with the meters in the three tested hardware switches, our meter scheme can improve the achieved rate of a TCP flow by almost 85% of the target rate set by the meter.

In addition to the above contribution, our work makes the following contributions as well. Firstly, most commercial switches on the market provide their meters in a blackbox manner, which cannot be analyzed or changed for further improvements. In contrast, our meter scheme is designed and implemented in the pipelines of a switching ASIC by using the P4 programming language. Our P4 code presented in this paper enables other researchers to develop their P4-based meters on top of our meter. Secondly, in addition to being used in high-speed data center switches, P4 is already used in free software switches and low-cost smart network interface cards (NICs). As a result, our P4-based meter scheme can be readily and widely used in smart NICs, switches, routers, and gateways to regulate the traffic of a flow.

The rest of the paper is organized as follows. Section II provides background information and present related works. In Section III, we briefly present the P4 switch architecture to help readers understand our P4 code. We also present the functional blocks of our scheme and explain their relationships to the P4 switch architecture. Section IV presents the P4-based design and implementation of our meter scheme. In Section V, we evaluate the performances of our meter scheme and compare its performances with those of other existing meters. We discuss future work in Section VI. Lastly, we conclude the paper in Section VII.

II. BACKGROUND AND RELATED WORK

In this section, we describe a network architecture suitable for using meters, the meter algorithms defined in RFC-2697 [2] and RFC-2698 [3], and the previous studies about meters.

A. A Network Architecture Suitable for Using Meters

Fig. 1 illustrates an architecture for a spine-leaf network, which is used by most data centers such as those at Facebook [8]. In such an architecture, meters can be used in the rack switches to provide QoS supports for the flows set up between the hosts in racks. A rack switch marks packets sent from a host, possibly on a per-flow basis, by using a pre-configured “marking profile.” In this architecture, suppose that a host in rack A (Fig. 1 (1)) establishes a TCP connection to a host in rack F (Fig. 1 (2)). When the packets of the TCP flow arrive at rack switch A (Fig. 1 (3)), the meter in the rack switch regulates the rate of the TCP flow based on the administrator’s settings. If the rate of the flow conforms to its marking profile, the packets are marked as “pass.” Otherwise, the packets are marked as “drop” and they are dropped by the meter. For those packets that are allowed to pass through the meter, they are sent through an aggregation switch (Fig. 1 (4)), a core switch (Fig. 1 (5)), another aggregation switch (Fig. 1 (6)), and another rack switch (Fig. 1 (7)) to the destination host (Fig. 1 (2)).

B. Meter Algorithms

Two meter algorithms are defined in RFC-2697 [2] and RFC-2698 [3]. The former algorithm describes a “Single Rate Three Color Marker,” which has three traffic parameters — Committed Information Rate (CIR), Committed Burst Size (CBS), and Excess Burst Size (EBS). CBS and EBS are the sizes of two token buckets, whose tokens are added at the CIR rate. The meter marks an incoming packet as either green, yellow, or red, depending on whether the size of the incoming packet is less than the current level of tokens in
First, we focus on designing and implementing a TCP-friendly meter scheme to enable the achieved rate of a TCP flow to approach the meter’s target rate. Compared with the meters in several commercial switches, our meter scheme can greatly improve the achieved rate of a TCP flow. Second, we have designed our meter scheme and implemented it in a real P4 hardware switch with 100 Gbps as the port bandwidth. Our experimental results obtained on this hardware switch show the real performance of our meter scheme when it regulates real-life TCP traffic. In contrast, due to the closed switch platforms in the past, most existing works could not implement their meter methods in a hardware switch to see how well these methods regulate real-life traffic.

### III. P4 Switch Architecture

A programmable hardware switch, in which we designed and implemented our meter scheme, allows developers to design and implement their algorithms to process packets in the pipelines of its switching ASIC at line rate. The P4 programming language ([15], [16]) is an important language that was recently proposed to program the pipelines of a hardware switch. P4 provides a way to configure how packets are processed in the pipelines. A P4 program specifies how packet headers are parsed and how their fields are used by the “match + action” tables. The primitive actions defined in the language or user-defined actions can modify the fields of packet headers or the content stored in the switch’s metadata registers or register arrays, which can be accessed in the pipelines according to the P4 application’s needs. The parser can extract the header fields of an incoming packet following the parser state machine defined in the P4 program. A P4 program developer can also define his packet headers and desired parser state machine that operates on these user-defined headers. Using P4, in this work, we have designed and implemented our TCP-friendly meter scheme in a P4 hardware switch and shown the advantages of our meter scheme over other existing meters.

At present, in addition to being used in high-speed data center switches, the P4 technology has been used in several free software switches such as the BMv2 software switch [17] and low-cost P4 smart NICs [18]. To let readers understand the P4-based design and implementation of our meter scheme, which will be presented in the rest of the paper, here we briefly explain the P4 switch architecture.

Fig. 3 illustrates the P4_14 abstract forwarding model [20]. This model contains three components: the Parser/Deparser (Fig. 3 (1) and (2)), Match+Action tables (Fig. 3 (3)), and Traffic Manager (TM; Fig. 3 (4)). Some of these components are declared in the P4 program that configures the switch (Fig. 3 (5)). Parse graph describes how parser/deparsers finite state machine works. Control program specifies the order of the tables to be applied in the processing flow. Table configuration describes which header fields are matched by the tables.

---

**Fig. 2.** The flow chart of a single rate two color meter algorithm.

These two buckets. The latter algorithm describes a “Two Rate Three Color Marker,” which has two rates — Peak Information Rate (PIR) and Committed Information Rate (CIR), and their associated burst sizes. The meter marks an incoming packet as either green, yellow, or red, depending on whether the current rate of the flow has exceeded PIR or CIR.

Although these meter algorithms use two rates or two burst sizes and mark packets in one of three colors, most switch vendors also offer their meters in a simpler form. These simple meters are specified by only one rate and one burst size and these meters mark packets in one of two colors — green or red. For example, the meter in the commercial switch [14] has only two parameters — Committed Rate and Committed Burst Size.

We have no information about how switch vendors internally design and implement their meters. Because using a single rate and a single burst size to configure a meter is commonly used in commercial switches, we decided to take the same approach to design and implement our own meter scheme. Fig. 2 shows the flow chart of a single rate two color meter algorithm that is used in our scheme. Like the meters defined in RFC-2697 [2] and RFC-2698 [3], the meter internally uses a token bucket to accumulate tokens that are generated at the rate of the meter. The burst size parameter value of a meter is used as the bucket size. The meter marks incoming packets as either red or green. If the size of an incoming packet exceeds the current level of tokens in the bucket, it is marked red and dropped. Otherwise, it is marked green and allowed to pass the meter.

**C. Previous Studies**

In the literature, the meter concepts ([2], [3]) and the methods to improve the performance of a meter have existed for a long time ([4], [9], [10]). Several studies (e.g., [11], [12]) used an adaptive control concept to control the rate of a TCP flow. Some of these schemes (e.g., [13]) improved the TCP rate by using a larger number of categories for a meter’s output. Others (e.g., [14]) proposed to use dynamic parameter values to improve the rate of a TCP flow. Due to the closed switch platforms in the past, most of these works used mathematical analyses or simulation tools to study the performances, and very few of them implemented and evaluated the performance of their methods in real hardware switches.

Our work differs from these previous works as follows. First, we focus on designing and implementing a TCP-friendly meter scheme to enable the achieved rate of a TCP flow to approach the meter’s target rate. Compared with the meters in several commercial switches, our meter scheme can greatly improve the achieved rate of a TCP flow. Second, we have designed our meter scheme and implemented it in a real P4 hardware switch with 100 Gbps as the port bandwidth. Our experimental results obtained on this hardware switch show the real performance of our meter scheme when it regulates real-life TCP traffic. In contrast, due to the closed switch platforms in the past, most existing works could not implement their meter methods in a hardware switch to see how well these methods regulate real-life traffic.
and what actions are executed by the tables. The parser is followed by the ingress and egress pipelines (Fig. 3 (6) and (7)). The ingress pipeline generates an egress specification that determines the output port for a packet. The values of packet headers can be modified in both the ingress and egress pipelines. Between these two pipelines, there is a traffic manager (TM; Fig. 3 (4)). TM provides queuing functions for the packets. In the P4 language’s definition, traffic manager also has the responsibility of dropping packets that are marked as “drop” by the ingress/egress pipelines. At the deparser (Fig. 3 (2)), the headers and the payload are assembled back into a well-formed packet.

Based on the P4 architecture shown in Fig. 3, Fig. 4 shows the functional blocks of our meter scheme, which contains several modules: RED_ECN (Fig. 4 (1)), meter (Fig. 4 (2)), per-sec byte counter (Fig. 4 (3)), and the adaptive controller (Fig. 4 (4)). The first three modules are implemented by using the Match+Action tables (Fig. 3 (3)) in the ingress pipeline (Fig. 3 (6)). When a packet enters the P4 switch, it enters into the RED_ECN module first. The RED_ECN module uses the Explicit Congestion Notification (ECN) scheme [5] to pre-alert the TCP sender to reduce its sending rate before the meter has to drop its packets due to insufficient tokens in the bucket. (Actually, the meter just marks packets as “red” or “green” and the “red” packets are dropped by the TM. However, to be brief, sometimes we just say that a meter drops packets when this difference is not important.) For those packets marked as “green,” the per-sec byte counter uses them to compute the current rate of a TCP flow and provide the computed rate to the adaptive controller (Fig. 4 (5)). For a TCP flow to achieve the target rate, the adaptive controller (Fig. 4 (4)) dynamically computes the new token adding period based on the information from the per-sec byte counter and uses the new value to configure the meter (Fig. 4 (6)). The adaptive controller is implemented as a program running in the control plane. In our meter scheme, both the data-plane P4 program and the control-plane controller program are run in the same switch.

IV. DESIGN AND IMPLEMENTATION

In the following sections, we present the design and implementation of each module used in our meter scheme.

A. Meter

We modify the meter algorithm shown in Fig. 2 and implement it by using the P4 programming language. It can be viewed as a standard RFC-2698 meter whose two rates are set to the same value and whose two burst sizes are also set to the same value.

We do not use the meter object available in the P4_14 language (which is a black box whose design and implementation cannot be changed), but instead design and implement this functionality by ourselves using registers. In our design, one token can allow one byte of data to pass through the meter. Therefore, the target rate is equal to the token adding rate (Fig. 2 (1)). In our scheme, we use a timer provided by the P4 switch to add tokens periodically. Equation (1) calculates the token adding period for a given target rate.

$$\text{token adding period} = \frac{\text{number of tokens added in a period}}{\text{target rate}}$$

The reasons for setting the number of tokens added in each period to 128 are as follows. This number was chosen so that the computed token adding periods for the target rates tested in our experiments did not become too short or too long. Ideally, the token adding period should be set to the period that is for adding just one token. However, using such a short period will incur too much overhead for the hardware timer. On the contrary, using a long period and adding a large number of tokens in such a long period will cause tokens to be added to the bucket of a meter in bursts. This will make the behavior of a meter deviate from its theoretic behavior by too much. To achieve a balance between the two concerns, we chose to use 128 as the number of tokens added in each period.

We use a register in our P4 program to record the current bucket level (Fig. 2 (3)). When the token adding timer periodically expires, our program adds 128 tokens to the register. The number of tokens stored in the register is always less than or equal to the bucket size. To store and carry the meter_tag shown in Fig. 2 with a packet, we define the meter_metadata metadata in the P4 program. The following code shows how we define it. The meter_tag field in the meter_metadata is used...
to store the meter module’s output, which is either “red” or “green.” These colors are defined as 1 and 0, respectively.

header_type meter_meta_t {
    fields {
        meter_tag: 8;
    }
}
metadata meter_meta_t meter_metadata;

As presented in Fig. 4, the meter module is placed in the data plane. The following code shows how we implement the meter module.

apply(packet_detect_T);
apply(forward_T);

With this code, a packet will sequentially pass through the packet_detect_T table and the forward_T table. The following code shows the designs of the packet_detect_T table, including the match fields and the actions that can be executed.

table packet_detect_T {
    reads {
        ethernet.srcAddr: exact;
        ethernet.etherType: exact;
    }
    actions {
        reg_update_A;
        set_meter_A;
        nop;
    }
}

The packet_detect_T table is used to determine whether an incoming packet is a TCP packet whose rate should be metered or a control packet generated by the token adding timer. We have presented that the meter module uses a timer to add tokens periodically. The hardware timer in the used P4 switch can be configured to periodically generate packets and send them to a specified input port. When such a packet enters the switch, our P4 code will add 128 tokens to the bucket. To distinguish these token-adding packets from the TCP packets whose rate is to be regulated by the meter, the packet_detect_T table uses the Ethernet source address and Ethernet type as the match fields and we set the Ethernet source address and Ethernet type of TCP packets to special values such as aa:aa:aa:aa:aa:aa and 0×880, respectively. Note that because only TCP traffic existed in our tests, as just a prototype, currently our meter scheme does not use the ipv4.protocol field as a match field to test whether a packet is a TCP packet. In the future, if our meter scheme is integrated into a network for production uses, certainly many details should be properly handled.

If the incoming packet is generated by the timer, the packet_detect_T table will execute the reg_update_A action to add tokens to the bucket. Otherwise, the packet_detect_T table will execute the set_meter_A action, which is based on the meter algorithm shown in Fig. 2 and will set the meter_tag field in the meter_metadata metadata to either red or green.

The following code shows the designs of the forward_T table, including the match fields and the actions that may be executed on packets.

table forward_T {
    reads {
        ethernet.dstAddr: exact;
        meter_metadata.meter_tag: exact;
    }
    actions {
        setEgr_A;
        setEgrWithCount_A;
        _drop;
        nop;
    }
}

The forward_T table uses the Ethernet destination address and the meter_tag field in the meter_metadata metadata as the match fields. If the meter_tag field is marked as red, the forward_T table will execute the _drop action to drop the packet. Otherwise, the forward_T table will execute either the setEgr_A action or the setEgrWithCount_A action to set the egress port for the packet. The setEgr_A action just sets the egress port for a packet and is used for packets not regulated by our meter scheme. The setEgrWithCount_A action not only sets the egress port but also counts the number of bytes of the packet. This action implements the per-sec byte counter shown in Fig. 4 and is used for the packets that are regulated by our meter scheme. If the direction of the packet is from the TCP sender to the TCP receiver, the executed action is the setEgrWithCount_A action as we want to compute the current rate from the TCP sender to the TCP receiver. If the incoming packet is a control packet generated by the token adding timer, it will be dropped in the forward_T table without being processed.

B. RED_ECN

RED_ECN is a mechanism to pre-alert the TCP sender to reduce its sending rate before the meter module has to drop its packets. RED_ECN is composed of two mechanisms — Random Early Detection (RED) and Explicit Congestion Notification (ECN). We integrate the designs of these two mechanisms and implement them in the P4 switch.

1) Random Early Detection: Random Early Detection [6] was designed for congestion avoidance in networks. A network equipment such as a gateway detects incipient congestion by computing the average queue size. The network equipment can notify connections of congestion either by dropping their packets arriving at the equipment or by setting a bit in the packet headers. When the average queue size exceeds a certain threshold, the network equipment either drops or marks each arriving packet with a certain probability, which can be a constant value or a function of the average queue size.
2) Explicit Congestion Notification: Explicit congestion notification (ECN) is an extension to the internet protocol (IP) and the transmission control protocol (TCP) and is defined in [5]. ECN allows end-to-end notification of network congestion without dropping packets. Conventionally, TCP/IP networks signal congestion by dropping packets. After ECN is successfully negotiated between a TCP sender and a TCP receiver, an ECN-aware network equipment may set a mark in the IP header of a packet instead of dropping the packet to notify the TCP sender of impending congestion.

In the network layer, this scheme uses a 2-bit ECN field in the IP header to mark four different ECN codepoints, ‘00’ to ‘11.’ The not-ECT codepoint ‘00’ indicates a packet that is not using ECN. The ECN-Capable Transport (ECT) codepoints ‘10’ and ‘01’ are set by the sender to indicate that the end-points of the transport protocol are ECN-capable. These two codepoints are called ECT(0) and ECT(1), respectively. The network equipment treats the ECT(0) and ECT(1) codepoints equivalently. Senders are free to use either the ECT(0) or the ECT(1) codepoint to indicate ECT. The Congestion Encountered (CE) codepoint ‘11’ is set by a network equipment to indicate congestion to the end nodes.

In the transport layer, the TCP sender and TCP receiver use the ECN-Echo (ECE) flag in the TCP header to negotiate the ECN-Capability. Bit 9 in the reserved field of the TCP header is designated as the ECN-Echo flag. Bit 8 is used as the Congestion Window Reduced (CWR) flag. When the network equipment detects congestion and finds that an ECT codepoint is set in the packet, the equipment sets the CE codepoint in the IP header and forwards the packet. When the TCP receiver receives the packet with the CE codepoint, it sets the ECN-Echo flag in its next TCP ACK packet to be sent to the TCP sender. The TCP sender receives the ACK packet with the ECN-Echo flag set, and reacts to the congestion as if the packet had been dropped. The TCP sender sets the CWR flag in the TCP header of the next packet to be sent to the TCP receiver to acknowledge its receipt of and reaction to the ECN-Echo flag.

In summary, ECN uses the ECT and CE flags in the IP header for signaling between the network equipment and connection endpoints, and uses the ECN-Echo and CWR flags in the TCP header for TCP-endpoint to TCP-endpoint signaling. Currently, the Windows operating system supports ECN. For the BSD operating systems, both FreeBSD and OpenBSD support ECN. For the Mac operating system, it supports ECN as well.

3) Design and Implementation of RED_ECN: In the original design of RED, the network equipment computes the average queue size to detect congestion. In our design, at present, we use the current token bucket level (Fig. 2 (3)) to replace the average queue size. At first, we planned to use the average token bucket level rather than the current token bucket level to replace the average queue size. However, as P4_14 does not support the multiplication operation and our meter scheme was implemented in the P4_14 language, at present we can only use the current token bucket level to replace the average queue size. However, as P4_14 does not support the multiplication operation and our meter scheme was implemented in the P4_14 language, at present we can only use the current token bucket level to replace the average queue size. In the future, when we port our P4 program from P4_14 to P4_16, which has a better support for the multiplication operation, our scheme may be able to use the average token bucket level to achieve better results.

As shown in Fig. 5, the RED_ECN module first computes the ratio of the bucket level to the bucket size (Fig. 5 (1)). Then, the RED_ECN module compares the ratio with a certain threshold R (Fig. 5 (2)). The value of the threshold R is...
between 0 and 1. If the ratio is less than R (Fig. 5 (3)), the packet is in the mark zone (Fig. 5 (4)) and will be selected randomly with the probability $P_M$ (Fig. 5 (5)). If the packet is selected, RED_ECN sets the CE codepoint ‘11’ (3 in the decimal form) (Fig. 5 (6)) in its IP header and forwards the packet to the next module.

The module that follows RED_ECN is the meter module. As shown in Fig. 6, no matter whether the CE codepoint is set in the ECN field of a packet, RED_ECN forwards it to the meter module, which will mark it as either red or green (Fig. 6 (1)). In our design, in order to provide a higher probability to forward packets that are marked with the CE codepoint, if the meter_tag is set to red by the meter module and the ECN field in the IP header is set to the CE codepoint by RED_ECN (Fig. 6 (2)), the meter module compares the current bucket level with a half of the packet length (Fig. 6 (3)). If the current bucket level is larger than a half of the packet length, the meter_tag is set to green to forward the packet to the TCP receiver. Otherwise, the meter_tag is still set to red. The following check (Fig. 6 (4)) decides whether to forward or drop the packet based on its meter_tag. If the meter_tag is set to green, the switch forwards it; otherwise, the switch drops it.

As presented in Fig. 4, the RED_ECN module is placed in the data plane and implemented by the P4 code. The following code shows how we implement the RED_ECN module with other modules in the pipelines.

```p4
apply(calculateREDParameter_T);
apply(subtractBucketLevelToThreshold_T);
apply(compareBucketLevelToThreshold_T);
apply(RndDrop_T);
apply(RED_T);
apply(packet_detect_T);
if(ipv4.ECN == $3$ and
   meter_metadata.meter_tag == RED){
   apply(compareECNParameter_T);
}
apply(forward_T);
```

Threshold_T table, the RndDrop_T table, the RED_T table, the packet_detect_T table, the compareECNParameter_T table (if the conditions hold), and the forward_T table. The calculateREDParameter_T table and subtractBucketLevelToThreshold_T table calculate the ratio of the current bucket level to the bucket size (Fig. 5 (1)). The mathematical form of comparing the ratio with the threshold R is shown in Equation (2). Because the P4 language does not support the division operation, we transform Equation (2) into Equation (3).

$$\frac{\text{bucket\_level}}{\text{bucket\_size}} < R \quad (2)$$

$$\text{bucket\_level} - R \times \text{bucket\_size} < 0 \quad (3)$$
The adaptive controller mechanism is composed of the per-sec byte counter module (Fig. 4 (3)) and the adaptive controller module (Fig. 4 (4)). The per-sec byte counter module is implemented by P4 code in the data plane. The adaptive controller module is implemented by python code in the control plane and is run by the CPU. The per-sec byte counter module and the adaptive controller module are implemented and run on the same P4 switch. Every second, the adaptive controller module computes the current achieved rate of a TCP flow by using the per-sec byte counter information. It uses the python APIs provided by the P4 switch compiler [22] to access the counter in the data plane (Fig. 4 (5)). Besides, it maintains the achieved rate of a TCP flow near the target rate by periodically adjusting the token adding period (Fig. 4 (6)). Because the adaptive controller module needs to use floating-point operations to compute the current achieved rate of a TCP flow but accurate floating-point operations cannot be supported in the pipeline, currently it needs to be implemented in the control plane rather than in the data plane.

To maintain the achieved rate of a TCP flow near the target rate, we chose to use the Proportional-Integral-Derivative (PID) control algorithm to adjust the used token adding period. The PID control algorithm [7] is a control-loop feedback mechanism, which includes elements with three functions. The proportional element is referred to as the “P element,” the integral element as the “I element,” and the derivative element as the “D element.” The $u(t)$ function is considered as the controller’s output. The controller attempts to minimize the error over time by adjustment of $u(t)$. The mathematical form of the PID algorithm is described in Equation (4), where $K_p$, $K_i$, and $K_d$ respectively are the proportional gain, integral gain, and derivative gain. These parameter values are adjustable. The error function $e(t)$ is defined in Equation (5), where $R$ is the target rate, $r(t)$ is the current achieved rate, and $t$ is the present time. A PID controller periodically calculates an error function $e(t)$ and applies correction to the meter module based on the output function $u(t)$ to minimize the error.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \quad (4)$$

$$e(t) = R - r(t) \quad (5)$$

To compute the current achieved rate of a TCP flow, the per-sec byte counter counts the number of bytes of the TCP flow passing through the meter module in each second. The current achieved rate is computed by the following formula:

$$r(t) = \frac{r_c(t)}{\Delta t} \quad (6)$$
Here, $r(t)$ and $r_c(t)$ are the current achieved rate and the value of the per-sec byte counter at time $t$, respectively. Because the byte counter is reset to 0 every second, the $\Delta t$ in Equation (6) is 1. After the adaptive controller gets the current achieved rate $r(t)$, it computes the error $e(t)$ by using Equation (5), and then the adaptive controller computes output $u(t)$ by using Equation (4). Parameter values for $K_p$, $K_1$, and $K_d$ are set by the network administrator. The adaptive controller uses Equation (7) to compute the new token adding period, which is based on Equation (1).

$$T(t) = \frac{N}{(R + u(t))}$$  \hfill (7)

Here, $T(t)$ is the new token adding period at time $t$ and $N$ is the number of tokens added in a period (which is set to 128 in our scheme). After the adaptive controller obtains the new token adding period, it sets the meter with the new token adding period via python APIs (Fig. 4 (6)).

V. PERFORMANCE EVALUATION

We have implemented the designs of our meter scheme in an Inventec P4 switch [23], which uses the Barefoot Tofino chip [22] as its switching ASIC. This chip provides 6.5 Tbps front-panel bandwidth divided into 64 100-Gbps QSFP28 ports. The P4 switch can lower the line rate of a 100-Gbps QSFP port to 40 Gbps and divide the port to 4 subports each having 10 Gbps bandwidth. With this capability, we used a QSFP to 4xSFP breakout cable to connect a sub-port to a host, whose NIC could only support 10 Gbps.

We used two hosts to act as the TCP sender and TCP receiver, respectively. The TCP sender host used a 4-core Intel(R) Core(TM) i5-6500 CPU that operates at 3.2 GHz and was equipped with 32 GB RAM. The TCP receiver host ran Linux kernel 4.15.

Fig. 8. The topology used for the performance-evaluation experiments.

We used two hosts to act as the TCP sender and TCP receiver, respectively. The TCP sender host used a 4-core Intel(R) Core(TM) i5-6500 CPU that operates at 3.2 GHz and was equipped with 32 GB RAM. The TCP receiver host used a 12-core Intel(R) Core(TM) i7-8700 CPU that operates at 3.2 GHz and was equipped with 16 GB RAM. The TCP sender host and TCP receiver host ran Linux kernel 4.15.

Fig. 8 shows the topology formed by the TCP sender, TCP receiver, and Inventec P4 switch. The TCP sender was connected to port 7/0 (i.e., the sub-port 0 of port 7) of the P4 switch while the TCP receiver was connected to port 7/3 of the P4 switch. Both the TCP sender and TCP receiver used the standard TCP/IP protocol stack implemented in the Linux kernel, which uses TCP CUBIC as the congestion control algorithm. Since version 2.6.31, in its default setting [24], the Linux kernel does not automatically request ECN on outgoing connections but will enable ECN when requested by incoming connections. Thus, we used the command: `sysctl net.ipv4.tcp_ecn=1` to enable the Linux kernel to request ECN on outgoing TCP connections. We set the duration of each experiment to 180 seconds and used `iperf` [21] to report the average achieved rate of a TCP flow in an experiment.

A. Performance Comparison With the Meters in Other Switches

Here, we compare our meter scheme with the meters in other switches. We used three switches that were available in our lab — (1) Netgear M4300-8 x 8F switch [26]; (2) Edge-Core AS4600 [27] switch, and (3) Inventec P4 switch [23]. For the Netgear and Edge-Core switches, we used their CLI interfaces to configure and use their meters without using any P4 program. For the Inventec P4 switch, we wrote a P4 program to call a P4 blackbox function to configure and use its meters. For these meters, we set their bucket sizes to the same size of 128 KB, which is the default size used by (and also the maximum size allowed by) the Netgear M4300 switch [14]. As mentioned in Section I, we have tested and found that using a bucket size smaller than 128 KB would further lower the achieved rate of a TCP flow regulated by these meters.

We used the same TCP sender/receiver hosts to evaluate these meters and our scheme. To measure the performance of these meters under different target rates, we varied the target rate from 100 to 800 Mbps. For the configuration of our meter scheme, we set the RED threshold $R$ to 1/2 and the mark probability to 20%. As for the adaptive controller, we set $K_p = 0.7$, $K_d = 0.15$, and $K_d = 0$. According to our extensive tests, these parameter values work well for our meter scheme. Our tuning of these parameters are as follows.

First, we gradually increased $K_p$ until the achieved rate of the TCP flow got close to the target rate. Although now the achieved rate could be closer to the target rate, it fluctuated quite greatly. Then, we gradually increased $K_d$. Doing this could decrease the level of fluctuation. As for $K_d$, we did not tune it but just used its default value of 0 because its default value already worked well for our scheme. We found that if we increased the value of $K_d$, this control-loop algorithm would cause excessive response and degrade the performance of our scheme.

Fig. 9 shows the achieved rates under different target rates and different meters in the tested switches. In this figure, the Tofino’s meter refers to the Inventec P4 switch. This is because the Inventec P4 switch uses the Tofino chip as the switching ASIC and we wrote a P4 program that used a standard RFC-2698 meter objects defined in P4_14 specification to meter a TCP flow.

For easier comparisons, we place the bars filled with the checkered pattern into the figure to represent the target rates. Fig. 10 shows the relative error to the target rate. It is defined as $(\text{achieved rate}/\text{target rate}) - 1$. Ideally, it should be 0%.
One can see that under the control of the meters in the Netgear M4300 and Edge-core AS4600 switches and the standard meter used in the Inventec P4 switch, the achieved rate of a TCP flow is much less than the target rate. For the target rates ranging from 100 Mbps to 800 Mbps, the relative errors of the meter in the Netgear M4300 switch were between $-85\%$ and $-90\%$, those of the meter in the Edge-core AS4600 switch were between $-42\%$ and $-83\%$, and those of the standard RFC-2698 meter in the Inventec P4 switch were between $-78\%$ and $-90\%$, respectively. These relative errors were very large, showing that these meters were not friendly to TCP flows.

In contrast, as can be seen, the achieved rates of a TCP flow under our meter scheme were very close to the target rates. The relative errors of our meter scheme in the same target rate range were between $-2\%$ and $-9\%$ only. Comparing the relative errors of the four meters for each target rate, one can see that our meter scheme performed much better than the three tested meters. For most target rates, our meter scheme could regulate a TCP flow very well and maintain its achieved rate within 5% of the target rate. These results show that our meter scheme is TCP-friendly. Compared with other meters, our meter scheme could improve the achieved rate of a TCP flow by almost 85% of the target rate.

### B. Evaluation of the Effectiveness of Each Module in Our Scheme

Here, we evaluate the effectiveness of each module used in our meter scheme and show that all of these modules are useful to be included into our scheme. In this section, we design experiments to evaluate these modules separately. As shown in Table I, the meter module was used in all experiments (Exp. for short) indexed from 0 to 3 because it is the necessary module used to limit the rate of a flow. As for RED_ECN and the adaptive controller modules, we evaluated their effectiveness in Exp. 1 and Exp. 2, respectively. Exp. 0 only used the meter module as the baseline. Exp. 3 instead used all modules and showed the performances of our complete scheme. The target rates were set to 100, 200, 400, and 800 Mbps, respectively.

Fig. 11 shows the achieved rates under different designs of our meter scheme and Fig. 12 shows their relative errors to the target rates. As shown in Fig. 11 and Fig. 12, when we only enabled the meter module, the achieved rate was much less than the target rate. When we enabled both the meter module and the RED_ECN module (Exp. 1), the achieved rate increased and became closer to the target rate when the target rate was high (e.g., 800 Mbps). However, as shown in Fig. 12, Exp. 2 could not make the achieved rate close to the target rate when the target rate was low (e.g., 100 Mbps). This can be seen as an indication that the RED_ECN mechanism was more effective when the target rate was higher. This phenomenon could be explained as when the delay and bandwidth (i.e., the target rate) product of a TCP flow is large, its congestion window size can increase to contain more TCP packets per round-trip time (RTT). With more packets in an RTT, the TCP sender will have a better chance to trigger its fast retransmit mechanism without the need to wait in long-period transmission timeouts. On the other hand, with fewer packets in an RTT, the TCP sender will have a smaller chance to successfully use the fast retransmit mechanism to continue its transmission. As a result, the achieved rate of a TCP flow will be lower.

When we enabled both the meter module and the adaptive controller module (Exp. 2), the achieved rate increased and became close to the target rate when the target rate was high (e.g., 800 Mbps). However, as shown in Fig. 12, Exp. 2 could not make the achieved rate close to the target rate when the target rate was low (e.g., 100 Mbps). This can be seen as an indication that the adaptive controller module was more effective when the target rate was lower. This phenomenon could be explained as when the delay and bandwidth (i.e., the target rate) product of a TCP flow is small, its congestion window size can increase to contain fewer TCP packets per round-trip time (RTT). With fewer packets in an RTT, the TCP sender will have a smaller chance to trigger its fast retransmit mechanism without the need to wait in long-period transmission timeouts. On the other hand, with more packets in an RTT, the TCP sender will have a larger chance to successfully use the fast retransmit mechanism to continue its transmission. As a result, the achieved rate of a TCP flow will be lower.
the relative error of -20% was still large when the target rate was 100 Mbps. This result showed that the adaptive controller module, although could increase the achieved rate when the target rate was high, still could not perform well enough when the target rate was low.

When we enabled all of these modules (Exp. 3), the achieved rate became very close to the target rate even at low target rates and the relative error to the 100 Mbps target rate could be further reduced from -20% to only -10%. Exp. 3 showed that the RED_ECN module helped the meter module and the adaptive controller module to further approach the target rate when the target rate was low. Without using the RED_ECN module, our meter scheme could not reduce the relative error to a value less than -10% when the target rate was 100 Mbps.

Based on these experimental results, one can see that all of the modules used in our meter scheme are useful. The meter module limits the rate of a flow. The adaptive controller module raises the achieved rate of a TCP flow. Without doing so, the achieved rate of a TCP flow will be always less than the target rate due to the sawtooth behavior of the TCP congestion window size. As for the RED_ECN module, it is TCP-friendly as it pre-alerts the TCP sender to reduce its sending rate before the meter module has to drop its packet. Without using it, more TCP packets will be dropped by the meter in bursts, which will cause the TCP sender to time out often and thus greatly lower the achieved rate of the TCP flow.

C. Evaluation of the Performance Under Multiple TCP Flows

Here, we evaluate the performance of our meter scheme under multiple TCP flows. Each TCP flow was regulated by a separate meter. We used the same topology as shown in Fig. 8 to evaluate the performance. During experiments, we established up to four TCP flows between the TCP sender host and TCP receiver host. Because the port bandwidth of each host was 10 Gbps and the target rate of a TCP flow was set under 1 Gbps in our experiments, the ports on these hosts could support four TCP flows at the same time without problems. We designed three sets of experiments to evaluate our meter scheme. In the first set of experiments, the number of TCP flows was varied from 1 to 4 and their target rates were all set to 400 Mbps.

Fig. 13 shows the achieved rates under different numbers of TCP flows when all of their target rates were set to 400 Mbps.
In the second set of experiments, the number of TCP flows simultaneously regulated by our meter scheme was fixed to four and their target rates were set to 100, 200, 300, and 400 Mbps, respectively. Fig. 17 shows the achieved rates of these TCP flows. From the figure, one can see that even when multiple TCP flows, each specified with a different target rate, were regulated by our meter scheme, their achieved rates could still approach their respective target rates.

For comparison purposes, we performed the same experiments using the meter in the Edge-core switch, the meter in the Netgear switch, and the meter in the Tofino chip. Fig. 18, Fig. 19, and Fig. 20 show their performance, respectively.

As can be seen, the achieved rates of TCP flows under the regulation of these meters were far below their target rates, which were set to 100, 200, 300, and 400 Mbps, respectively. Fig. 21 shows the achieved rates of the TCP flows in the third set of experiments, which used our meter scheme and the target rates of these TCP flows were set to 400, 500, 600, and 700 Mbps, respectively. As can be seen, our scheme still performed well when the target rates of these TCP flows were set to 400, 500, 600, and 700 Mbps, respectively.

Due to space limitation, in the following we only present the performance of the meter in the Netgear switch in Fig. 22.
The results for the meter in the Edge-Core switch and the Tofino meter were similar and thus are omitted here.

VI. FUTURE WORK

The design and implementation of our meter scheme presented in this paper are just a proof-of-concept demonstration showing how the modules in our meter scheme can enable a TCP flow to achieve the target rate of a meter. As a prototype, currently our meter scheme uses one timer to periodically add tokens to the bucket that is associated with a meter. Due to the constraint on the number of hardware timers, at present the number of flows that can simultaneously be regulated by our meter scheme is limited. To overcome this hardware constraint, in the future we will take the approach used by the Linux kernel. On the Linux system, although the kernel and many processes may schedule many timeout events that need to be invoked in the future to perform some actions, the Linux kernel just uses one hardware timer to trigger and execute all scheduled events. We plan to use this efficient mechanism in the future to be able to regulate more flows in our meter scheme.

Another place worth more investigations in the future is developing a method to find the best set of parameter values for our meter scheme. The RED_ECN and adaptive controller modules have several parameter values to set. Currently, we conduct extensive experiments to find a set of parameter values for our meter scheme to perform well. In the future, we plan to design machine learning-based techniques to do this job. Ideally, these techniques can automatically find a good set of parameter values at run time based on the current network traffic. We leave these works as our future works.

Yet another place for further studies is to test how our meter scheme would perform with different versions of TCP congestion control algorithm. In our work, the TCP/IP protocol stack in the Linux hosts used the default TCP CUBIC congestion control algorithm. Although Linux hosts are very popular and commonly used in the world, there are other TCP congestion control algorithms such as TCP reno, new reno, taho, vegas, BBR, DCTCP, C2TCP, etc. For these TCP congestion control algorithms, the parameter values of our meter scheme may need to be adjusted to achieve good performances. For example, TCP vegas uses packet delay, rather than packet loss, as a congestion signal to help the source node determine the rate at which to send packets. Since our scheme uses packet marking by ECN and packet dropping by the meter to control the sending rate of a TCP flow, which is different from the packet delay-based approach used by TCP vegas, finding a good set of parameter values that can work well with TCP vegas may require an extensive search in the parameter space.

The performance of our scheme can be improved further if floating-point operations can be accurately supported in the pipeline. Currently, we can only use a power of 2 (e.g., $2^{-1}$) as the threshold for RED-ECN. Clearly, given only a very limited number of these coarse-grain choices, our scheme cannot achieve the best performance that it can possibly achieve. Thus, it is hoped that accurate floating-point operations can be supported in the next-generation P4 switches.

VII. CONCLUSION

Providing QoS in a network is important. Nowadays, various network applications co-exist on a network and share its bandwidth. These applications may have different bandwidth demands for their usages. To allocate bandwidth in a non-equal manner to network applications, using a meter to drop the packets of a flow whose rate exceeds the rate specified in the service contract is a necessary measure. There are many situations where a TCP connection needs to send a large amount of data. For example, to reliably send a virtual machine (VM) image file over a network, a TCP connection needs to be used. Since the size of a VM image file can easily go beyond 512 GB, the rate of the TCP flow should be regulated by the meter during the long transfer of data.

According to our experiments, the meters in the three tested commercial switches are not TCP-friendly. The achieved rates of a TCP flow under their regulations are much less than the target rate set by the meter. For some target rates, the relative error to the target rate could go up to -90%. To solve this problem, in this paper, we design and implement a TCP-friendly meter scheme in a P4 hardware switch. This scheme enables the achieved rate of a TCP flow to approach the target rate.

We have conducted many experiments to evaluate the performance of our meter scheme under different conditions. The factors that we varied include the target rate and the number of TCP flows that are simultaneously regulated by our meter scheme. Experimental results showed that, for most target rates tested in our experiments, our meter scheme could keep the achieved rate of a TCP flow within 5% of the target rate. Compared with the three tested switches, our meter scheme could improve the achieved rate of a TCP flow by almost 85% of the target rate.

Currently, P4 can already be supported by hardware switches, software switches, and smart NICs. Therefore, our P4-based TCP-friendly meter scheme can be used in all of these different kinds of devices. As P4 programmable switches and devices become more commonly used in the future, we expect that our meter scheme will be used at more places in the network.
ACKNOWLEDGMENT

The authors would like to thank Barefoot/Intel for their supports of this work.

REFERENCES


Shie-Yuan Wang (Senior Member, IEEE) received the master’s and Ph.D. degrees in computer science from Harvard University in 1997 and 1999, respectively. He is currently a Full Professor with the Department of Computer Science, National Chiao Tung University (NCTU), Taiwan. He has published many high-quality journal and conference papers in the fields of computer networks. His research interests include the Internet of Things, P4 programmable networks, software-defined networks, cloud and edge computing, networks security, and wireless networks. He received the Outstanding Information Technology Elite Award of Taiwan government in 2012, which was bestowed by the Vice President of Taiwan government. He authors the NCTUns/EstiNet network simulator and emulator, which is a famous tool used by many researchers in the world. He has served as the general chair, the technical program co-chair and a member for many prestigious IEEE conferences, such as ICC, GLOBECOM, NOMS, PIMRC, VTC, and ISCC. He is also serving as an Associate Editor for ACM Computing Surveys.

Hsien-Wen Hu (Member, IEEE) received the bachelor’s degree in communication engineering from National Central University, Taiwan, in 2017, and the master’s degree in network engineering from National Chiao Tung University, Taiwan, in 2019, under the supervision of Prof. S.-Y. Wang. He participated in and completed this work when he was with National Chiao Tung University. He is currently working with MediaTek Inc., and participating in a project related to developments of embedded systems for 5G networks. He received research interests include software-defined networks and P4 programmable networks.

Yi-Bing Lin (Fellow, IEEE) received the bachelor’s degree from National Cheng Kung University, Taiwan, in 1983, and the Ph.D. degree from the University of Washington, Seattle, WA, USA, in 1990. From 1990 to 1995, he was a Research Scientist with Bellcore (Telcordia). He then joined National Chiao Tung University (NCTU) in Taiwan. In 2010, he became a lifetime Chair Professor of NCTU, and in 2011, the Vice President of NCTU. From 2014 to 2016, he was the Deputy Minister of the Ministry of Science and Technology, Taiwan. Since 2016, he has been appointed as the Vice Chancellor of the University System of Taiwan (for NCTU, NTHU, NCU, and NYMU). He is currently a Winbond Chair Professor of NCTU. He is a fellow of AAAS, the ACM, and IET.