P4I/O: Intent-Based Networking with P4

Mohammad Riftadi, M.; Kuipers, Fernando

DOI
10.1109/NETSOFT.2019.8806662

Publication date
2019

Document Version
Accepted author manuscript

Published in
Proceedings of the 2019 IEEE Conference on Network Softwarization

P4I/O: Intent-Based Networking with P4

Mohammad Riftadi
Delft University of Technology
M.Riftadi@student.tudelft.nl

Fernando Kuipers
Delft University of Technology
F.A.Kuipers@tudelft.nl

***Abstract*—Switches that can be (re)programmed through the network programming language P4 are able to completely change – even while in the field – the way they process packets. While powerful, P4 code is inherently static, as it is written and installed to accommodate a particular network requirement. Writing new P4 code each time new requirements arise may be complex and limits our agility to deal with changes in network traffic and services.**

In this paper, we present P4I/O, a new approach to data-plane programmability based on the philosophy of Intent-Based Networking. P4I/O provides an intent-driven interface that can be used to install and/or remove P4 programs on the switches when needed and which is easy to use. In particular, to realize P4I/O, we (1) describe an extensible Intent Definition Language (IDL), (2) create a repository of P4 code templates, which are parsed and merged based on the intents, (3) provide a technique to realize the resulting P4 program in a programmable switch, while accommodating intent modifications at any time, and finally (4) implement a proof-of-concept to demonstrate that intent modifications can be done on-the-fly.

I. Introduction

Recent advances in data-plane programmability have enabled the implementation of customized high-speed network functions directly into programmable switches. Examples range from performing network telemetry on the switches [1], [2], to fast congestion detection on the data-plane [3], to offloading distributed consensus algorithms to the data-plane [4].

The data-plane abstraction language P4 [5] enables network operators to realize custom network functions without having to tailor to the networking hardware in use. Those hardware details will be taken care of by the compiler. This level of abstraction makes P4 code powerful, yet it is inherently static, because the code would have to be rewritten whenever a new network requirement manifests, which caps our agility in responding to network changes. We are in need of a system that is able to accommodate changes in network requirements and realize them quickly with minimum disruption to any existing services. To this very problem, *we present P4I/O, an Intent-Based Networking (IBN) framework that allows users to express their desired network functionality in the form of easy-to-grasp intents, which are subsequently translated into P4 code*, as illustrated in Figure 1. We adopt the concept of IBN to shift the network control paradigm from an imperative manner – which requires knowledge of the networking protocols and the network topology – to a declarative manner, allowing users to reap the benefits of data-plane programmability in an intuitive manner.

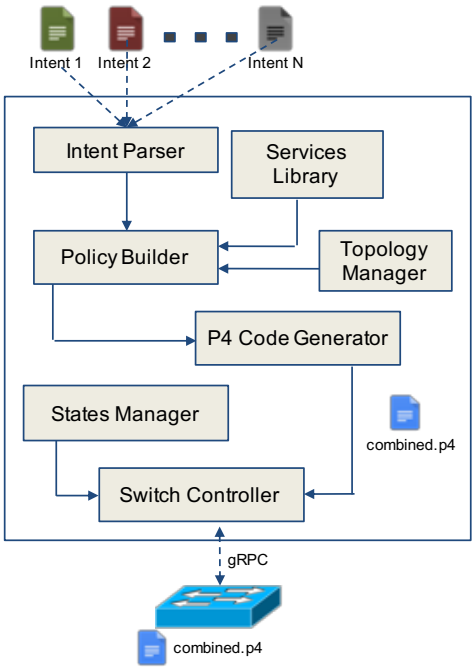

Figure 1. P4I/O architecture.

In order to realize P4I/O, in this paper, we present the following key contributions:

Extensible Intent Definition Language (IDL). In order to describe various kinds of network services as intents, we devise in §III a high-level language that is close to the human language, yet precise enough to be interpreted unambiguously by the network controller. Furthermore, this language is extensible so that we can define any kind of data-plane functionality.

Template-Based P4 Code Generation. We construct a repository of relevant network functions in the form of P4 code templates. These templates are then parsed and represented in a specialized data structure that facilitates combining the network functions, following the intent instructions. The code templates are then finally merged together to form a valid P4 program, as described in §IV.

Dynamic Intents Realization. We provide, in §V, a technique to install the resulting P4 code in a programmable switch, while permitting intent modification at any time. We realize intent modifications, with minimal disruption to the traffic forwarding process, through a state-transfer mechanism.

Framework Evaluation. In §VI, we demonstrate that P4I/O

works, by building a proof-of-concept. P4I/O code has been released as open-source code [6].

II. Network Telemetry Use Cases

P4I/O can be used for any network function, but to explain its building blocks, we will consider several use cases from the domain of network telemetry. Network telemetry functions provide interesting challenges to solve, because they require a state to be stored in the switch, e.g. in the form of P4 registers [7], along with complex pipeline processing logic to compute the states. We consider the following use cases: (1) Threshold-based Heavy-Hitter (HH) detection, (2) Distributed Denial of Service (DDoS) victim detection, and (3) Super-Spreader (SS) detection, which is the inverse of DDoS detection. All of these use cases are implemented using sketches, in particular *Count-Min Sketch* [8] and *Bitmap* [9].

For example, consider use case (1): HH detection is a mechanism to compute whether a packet is part of an HH flow, which we define as a flow that has more than our threshold of T packets. Consequently, the switch must track the number of packets in each flow, while it has very limited memory and processing resources. To effectively make use of the limited resources, we use the Count-Min sketch, which enables our switch to track a virtually unlimited number of flows at the expense of slightly reduced accuracy.

Our goal is to run a network in which network functions, such as the described telemetry functions, can be (de)activated at any time by adding/removing intents in the controller. Realizing this goal is challenging, because intent modification translates to the changing of switch pipeline configurations. As each set of network functions has its own requirements for state containers, we must consider the problem of preserving the state from one set of functions to another. Moreover, as the new set of functions might not have the same state containers available, we need a mechanism to manage the unused state values. Finally, we also have to minimize the effect of intent modifications on the traffic forwarding process.

The following sections describe how we tackle these issues.

III. Intent Definition Language

This section discusses the Intent Definition Language (IDL) that is employed to express network intents. We begin by identifying the requirements for such a language and subsequently present a working solution based on extending an existing IDL.

A. Requirements

While Han et al. [10] claim that there is currently no standardized definition of *network intent*, they also argue that intent is generally perceived as a business-level goal of how the network should behave, abstracting the implementation details from the operators. Reflecting on this objective, we identify the following requirements to be satisfied by our IDL:

Readability. Network operators should be able to intuitively express their intended services with minimum training. We propose to cater to this need by developing a language that is close to a natural language, e.g. English, yet still concise enough to be interpreted unambiguously by the controller.

Abstraction. The language should abstract out technical details of the implementation. To realize this, we have to move the details of the implementation to a lower layer. The lower-layer implementation should still be accessible by operators with advanced technical knowledge for debugging purposes.

Flexibility. The language should be flexible enough to be extended with any kind of required network functions. This means that we have to design for a modular architecture, which facilitates easy extensions of new network functions.

Listing 1. Drop Heavy-Hitter Action Example.

```
import drop_heavy_hitters

define intent drop_hh_any_any:
  from   endpoint('any')
  to     endpoint('any')
  for    traffic('any')
  apply  drop_heavy_hitters
  with   threshold('more or equal', 20)
```

B. Nile Language Extension

We employ Nile [11] as the base language for our IDL. Nile is a network intent language with the goal of providing an intermediate layer between a natural language and lower-level policies. In [11], user utterance is processed into a network intent expressed in the Nile language. Nile satisfies our requirements of readability and abstraction, but does not facilitate the import of external module definitions. To that end, we introduce several constructs:

1) *import*: to define custom actions and import them from the actions repository,
2) *apply*: to apply the custom action by specifying the name of the action in the intent definition, and
3) *with*: to provide parameter values required by the custom action.

The import construct can be used multiple times to define more than one custom action. The specified action is then executed whenever the specified conditions in the intent definition are satisfied.

For example, consider the code snippet in Listing 1. We define a new action named *drop_heavy_hitters* that performs a threshold-based HH detection, with threshold $T = 20$ packets. The *drop_heavy_hitters* action is to drop HH flows. This intent applies to all traffic, from any source or destination.

IV. P4 Code Templates

In this section, we describe our template-based method for forming P4 code. A knowledgeable party predefines P4 code templates that correspond to specific actions. The code templates are then imported into a special repository in the network controller. To render a template, the controller takes various attributes defined in the intent as inputs. If there is

```
const bit<8> HH_THRESHOLD = {{ hh_threshold_val }};
```

```
const bit<8> HH_THRESHOLD = 1000;
```

```
if (meta.minRegVal > HH_THRESHOLD) {
    drop();
}
```

Figure 2. (Top) Template example with placeholder. (Mid) Template example with rendered placeholder value. (Bottom) Manipulation section example.

more than one intent or the intent itself is more complex, we may also need to "merge" multiple P4 code templates into one final P4 program.

The templates are written in a templating-language that has several constructs that are syntactically distinguishable from the actual text. The constructs act as placeholders that can be populated with string values. This way, the final P4 code can be rendered by populating each placeholder with precomputed string values. We employ Jinja2 [12], one of the most popular templating-languages. Figure 2 (Top and Mid) illustrates the placeholder and its associated rendered code. The $\{\{variable\_name\}\}$ struct is used as a placeholder for a string named *hh_threshold_val*.

A. Network Telemetry Function Structures

The P4 code templates for the majority of network telemetry functions can be built upon the following structures:

Constant Definition. Integer constants, see Figure 2 (Top and Mid).

Parser Definition. The state machine to parse the required packet headers.

Metadata Definition. The metadata fields that are required by the network function to perform its computation.

Packet Identification. A computation to identify whether a packet belongs to a specified group of traffic (or vice versa).

Packet Manipulation. In this final phase, actions are taken on the identified traffic. In the P4 language, the possible actions are virtually limitless, but common ones are: count, mark, drop, meter and/or a combination of them. Figure 2 (Bottom) depicts an example for this phase.

B. Template Representation

In this section, we present a formal data structure to facilitate combining intents and their corresponding P4 code. As the network functions that we are dealing with are logical representations of packet pipeline processing, the data structure should also be able to move from one condition to the next, which is possible via a Directed Acyclic Graph (DAG) that is based on PGA's [13] graph structure.

1) Policy Graph Structure: To help illustrate our graph structure, we use the HH example code in Figure 3. The constant and metadata definitions contain no flow information and therefore are not processed further. The parser definition code can be represented as a directed graph with the vertices representing the packet header names and the edges representing a possible state transfer from one header to another.

2) Combining Templates: We proceed to define how several policy graphs can be combined. Each of the phases in §IV-A has different characteristics and should be treated differently. The simplest cases are the constant and metadata definitions. To combine constant and metadata definitions from several intents, we need to make the names unique by prepending each name with a unique text – preferably generated from the intent id number – and then do a string concatenation to combine all of the constant and metadata definitions from various intents together.

For the parser graphs, we expect many overlapping nodes coming from several policies, which can be resolved via computing the union of the vertices and the union of the edges from all of the graph policies. The resulting edges and vertices comprise the final graph for the parser.

Finally, the identification and manipulation actions are stitched together one after another. Formally, let's consider two intents $I_1 := (i_i, m_1)$ and $I_2 := (i_2, m_2)$, with (i_n, m_n) defined as a sequence of identification and manipulation actions. After the combination, we get the action sequence of $Combined := (i_1, m_1, i_2, m_2)$.

3) P4 Code Generation: For the final P4 program, we need another *base template* that contains placeholders for the aforementioned P4 code structures in §IV-A. This *base template* represents a valid P4 program with several empty sections, ready to be filled in with the rendered policies.

V. Intent Realization

This section starts by defining the software components used to realize the controller and finally discusses how the system handles intent modification. For ease of explanation, we limit the scope of the intent realization to one programmable switch connected to several hosts, but our framework can be extended to intent realization over multiple switches.

A. Software Components

As depicted in Figure 1, P4I/O consists of the following components:

Intent Parser. The intent parser parses the intents and stores them in a hierarchical key-value storage intended to be used by the policy builder.

Actions Library. The actions library acts as a repository for defined actions represented as P4 code templates. It parses the P4 code templates into a policy graph as explained in §IV-B1.

Topology Manager. The topology manager keeps track of each host connected to the switch ports. The information of which host is connected to which port is used to build a correct P4 match-action table entry required for the packet forwarding process.

Policy Builder. The policy builder executes the policy graphs join computation as explained in §IV-B2. It outputs the combined policy graph required to generate the final P4 program.

P4 Code Generator. The P4 code generator has as main objective to generate a working P4 program with the input

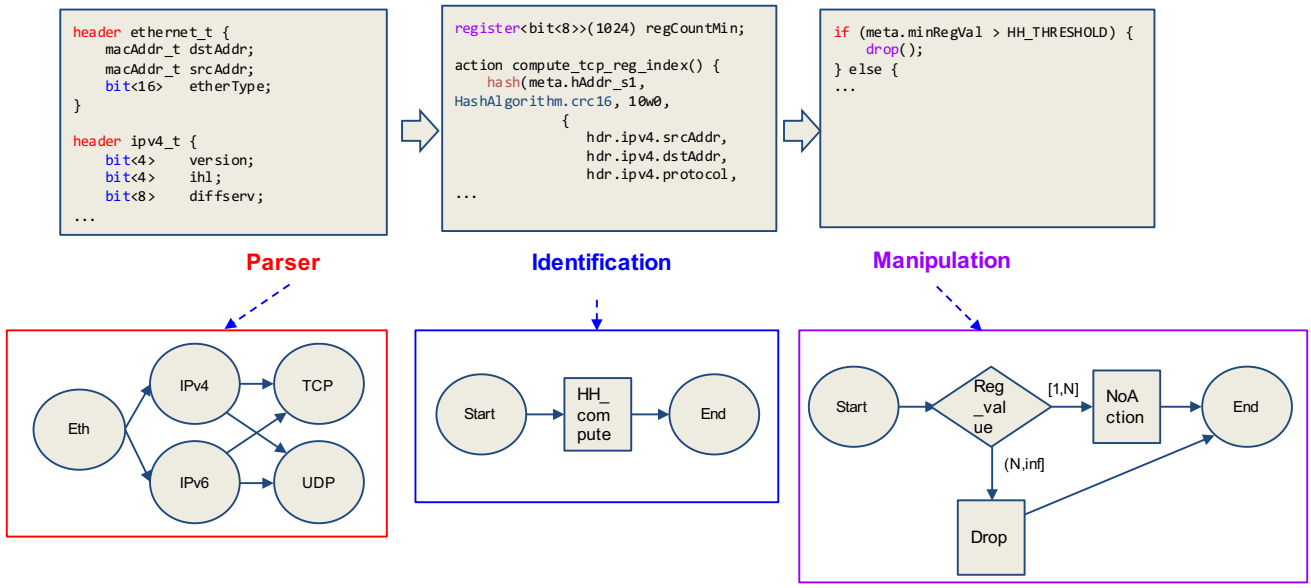

Figure 3. P4 code template translation to a Directed Acyclic Graph (DAG) structure.

of a policy graph, as described in §IV-B3.

States Manager. The states manager keeps track of all match-action table entries that are implemented on the switch. Each time a new P4 program is installed in the switch, the switch will lose all of its old entries and therefore needs to be repopulated by the entries stored in this component.

Switch Controller. The switch controller is responsible for pushing the generated pipeline configuration file into the switch and maintaining the communication channel to the switch, e.g. via gRPC. This process is done on-the-fly, while the switch is forwarding traffic.

B. Intent Modification

Switch pipeline configuration modification is realized via the interface defined in the P4Runtime specification [14], via the procedure of *SetForwardingPipelineConfig*. By default, each time a BMV2 [15] *simple_switch_grpc* is instructed to load a new pipeline configuration, it will lose all of the match-table entries and all of the other states, e.g. register, counter, meter, etc. We refer to this problem as the *state preservation problem*.

We handle the problem of state preservation by providing two mechanisms:

Match-Table Entries Preservation. Preservation is done by rewriting the match-action table entries every time a switch pipeline configuration is reloaded. The entries are stored at the *State Manager* component.

Switch States Preservation. The states in the switch can be preserved via two approaches: *external* and *internal backup*.

External Backup. In this approach, the value in the states is first read by the network controller. After the states are completely read, the controller reloads the switch with the new configuration and writes back the state arrays into the switch. The writing process can be done in two ways: (1) multiple single values are written into the switch memory, or (2) single multiple values are written by passing the whole array in one write procedure. The default *simple_switch_grpc* only supports method (1).

Internal Backup. This method does the backup within the internal memory of the switch. As it does not require any external communication, there will be no external communication overhead. However, the switch should have enough memory for the backup and dedicate some computation resources for the states duplication procedure.

For our proof-of-concept, we adopt the *internal backup* mechanism by developing a custom P4 software switch based on BMV2's *simple_switch_grpc* that has the functionality to preserve state from one configuration to another. It will first store all of its state to its internal memory. After the new pipeline configuration is enforced, the switch will try to restore the backed-up state, provided the previous state container still exists in the new one. The algorithm for the switch state preservation is depicted in Algorithm 1. This process happens while incoming packets are temporarily stored on an input buffer in the switch.

foreach *registers* ∪ *counters* ∪ *meters* ∪ ... **do**
  $backupState[stateName] \leftarrow oldValue$;
end
ENFORCENEWPIPELINECONFIG;
foreach *element in backupState* **do**
  if *element exists in new pipeline* **then**
    $newValue \leftarrow backupState[stateName]$;
  end
end

Algorithm 1: State Transfer Algorithm.

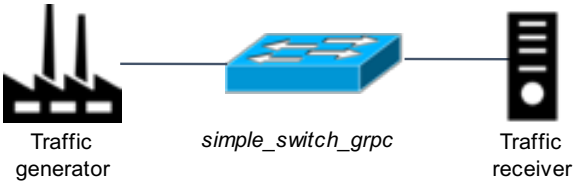

Figure 4. Proof-of-Concept topology.

VI. Evaluation

We evaluate the feasibility of P4I/O through a proof-of-concept implementation. Our prototype implementation is written in Python and contains approximately 4.530 lines of code.

A. Proof of Concept Setup

We define 6 epochs, $t'_0, t'_1, .., t'_5$, to aid us in evaluating intent modifications on P4I/O. In each epoch, different intents are defined. In t'_0, we run DDoS and SS detection functions. In t'_1, we add one Heavy-Hitter (HH) detection intent, which drops traffic exceeding a threshold of $T = 1800$ packets. In t'_2, we remove the SS detection. Further, in t'_3 we also remove the HH detection, leaving us with only DDoS detection. We then add back HH detection in t'_4, this time with a threshold of $T = 3600$ packets. Finally, we enter the last epoch of t'_5 by removing the HH detection function, effectively leaving DDoS detection as the only running function.

We use a simple topology with our modified *simple_switch_grpc* software switch connected to a traffic generator and a traffic receiver as illustrated in Figure 4. The generator then injects random UDP traffic at 1 mbps. We use a single IP address per sending/receiving side. The topology is run on top of Mininet [16], which in turn runs on top of a Ubuntu Linux 16.04 desktop with a dual-core 1.6 GHz Intel i5 processor and 2GB of RAM. The resources are shared between the host, Mininet, *simple_switch_grpc*, and P4I/O.

B. Result and Discussion

Figure 5 depicts the throughput graph as observed from the receiving side. At epoch t'_0, the traffic rate is relatively stable until we reach t'_1, which temporarily causes a fluctuating rate, due to the actualization of the new P4 code. We can note that at approximately $t = 27s$, the HH threshold $T = 1800$ is reached causing the traffic to be dropped. At epoch t'_2, the HH state from the previous epoch is successfully preserved as proven by the absence of traffic. The removal of HH detection network function at the beginning of epoch t'_3 gives us back the inbound traffic. Likewise, t'_4 also demonstrates the same phenomenon as t'_1, but with a twice as long delay before we reach the new threshold of $T = 3600$.

Reflecting on this result, we believe that our approach is applicable to real-world use cases. We can conclude that the existing hardware is fast enough to do pipeline configuration modifications with minimum interruption to the traffic forwarding process. While we see some fluctuating throughput rate in the result, it can be explained by the condition of our testbed that performs all computation, e.g. P4 code generation and traffic forwarding, in a single machine.

VII. Related Work

Programmable Network. Pyretic [17] allows building network services by means of network programmability. However, their approach to the network programming language is of imperative nature. PGA [13] addresses the problem of network policies reconciliation, i.e. aligning overlapping/conflicting policies, by devising a graph abstraction that inspires our DAG abstraction. PGA is implemented as an extension of Pyretic. Janus [18] extends the work of PGA by adding the notion of dynamic policies and incorporating QoS constraints. Janus' implementation is also based on Pyretic.

Intent-Based Networking. Nile [11] provides a human-readable IDL for implementing network services and is focused on translating user utterance into an IDL and incorporating user feedback to improve the translation model. Marple [1] and Sonata [2] generate pipeline configurations – also in P4 – from dynamic queries, but do not focus on the technique for generating the code. Moreover, they do not consider the problem of intent modification on-the-fly. Donovan & Feamster [19] propose the notion of intention-based monitoring, which offloads the task of matching traffic to the data-plane. Their Pyretic-based implementation matches traffic based on attributes with static mapping like domain name and AS number.

Sketch-Based Network Telemetry. OpenSketch [20] utilizes sketches for various flow measurement tasks. However, their implementation is on NetFPGA, which provides no mechanism to reload the switch pipeline configuration on-the-fly. UnivMon [21] contests OpenSketch's approach by proposing a universal sketch algorithm adopted from universal stream theory. UnivMon focuses on evaluating the performance and memory utilization of the universal sketch.

VIII. Conclusion

In this paper, we have presented P4I/O, an intent-based networking framework that facilitates a simple adoption of P4 data-plane programmability. Through P4I/O, programmable switches can be quickly and easily programmed, without having to code in P4. In order to build P4I/O, we have described an extensible Intent Definition Language, used a P4 code template approach, and enabled intent modifications on-the-fly. We have made an open-source Proof-of-Concept implementation of P4I/O, with which we have demonstrated that intents can indeed be installed/removed in the field.

References

[1] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-Directed Hardware Design for Network Performance Monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*, 2017, pp. 85–98. [Online]. Available: http://web.mit.edu/marple/marple-sigcomm17.pdf http://dl.acm.org/citation.cfm?doid=3098822.3098829

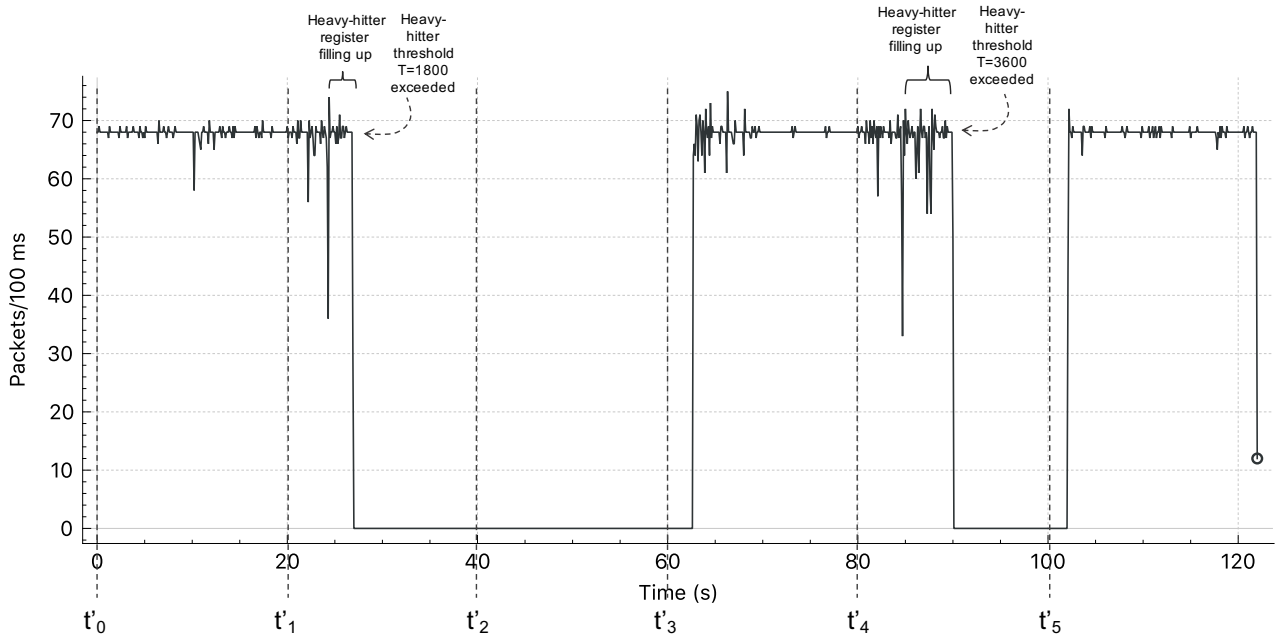

Figure 5. Throughput as observed from the receiving side in various epochs.

[2] A. Gupta, R. Harrison, A. Pawar, R. Birkner, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-Driven Network Telemetry," 2017. [Online]. Available: http://arxiv.org/abs/1705.01049

[3] B. Turkovic, F. Kuipers, N. van Adrichem, and K. Langendoen, "Fast network congestion detection and avoidance using p4," in *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*, ser. NEAT '18. New York, NY, USA: ACM, 2018, pp. 45–51. [Online]. Available: http://doi.acm.org/10.1145/3229574.3229581

[4] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:7. [Online]. Available: http://doi.acm.org/10.1145/2774993.2774999

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[6] M. Riftadi and F. Kuipers, "riftadi/p4io: Intent-based p4 code generation framework," https://github.com/riftadi/p4io, 2019, [Online].

[7] The P4 Language Consortium, "P416 language specification," https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, 2017, [Online; accessed 13-January-2019].

[8] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.jalgor.2003.12.001

[9] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high-speed links," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 925–937, Oct. 2006. [Online]. Available: http://dx.doi.org/10.1109/TNET.2006.882836

[10] Y. Han, J. Li, D. Hoang, J.-h. Yoo, and J. W.-k. Hong, "An Intent-based Network Virtualization Platform for SDN," 2016.

[11] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, "Refining Network Intents for Self-Driving Networks," in *Proceedings of the Afternoon Workshop on Self-Driving Networks - SelfDN 2018*. New York, New York, USA: ACM Press, 2018, pp. 15–21. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3229584.3229590

[12] A. Ronacher, "Jinja2," http://jinja.pocoo.org/, 2019, [Online; accessed 11-January-2019].

[13] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787506

[14] The P4.org API Working Group, "P4runtime specification," https://s3-us-west-2.amazonaws.com/p4runtime/docs/v1.0.0-rc4/P4Runtime-Spec.html, 2018, [Online; accessed 13-December-2018].

[15] Barefoot Network, "Behavioral model repository," https://github.com/p4lang/behavioral-model, 2019, [Online; accessed 23-January-2019].

[16] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[17] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482629

[18] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting Diverse Dynamic Intent-based Policies using Janus," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies - CoNEXT '17*, no. 1. New York, New York, USA: ACM Press, 2017, pp. 296–309. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3143361.3143380

[19] S. Donovan and N. Feamster, "Intentional network monitoring: Finding the needle without capturing the haystack," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIII. New York, NY, USA: ACM, 2014, pp. 5:1–5:7. [Online]. Available: http://doi.acm.org/10.1145/2670518.2673872

[20] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," *Proceedings 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, vol. 13, pp. 29–42, 2013.

[21] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, V. B. Johns, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," *Sigcomm*, no. Question 24, pp. 101–114, 2016. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934906