Round-Robin Bloom Filters Based Load Balancing of Packet Flows

Örs Szabó and Csaba Simon

Abstract **—SDN gives the possibility to design new solutions for flow based load balancers, needed by the handling of quickly growing Internet data, and end user demands. A key element of this can be the Bloom filters and its probabilistic techniques to reduce information processing and networking costs. We selected a Bloom filter variant optimized for low footprint and designed and implemented a flow based load balancer solution. We identified an issue of such load balancers during their initialization phase in case of plug and play deployments. We propose a solution to alleviate this problem and evaluated its performance.**

Keywords- Bloom Filter; Load Balancing; packet flow

I. INTRODUCTION

Internet data traffic is quickly growing, as more and more people are getting easy access to different kinds of services, such as file sharing, video streaming, video-on-demand (VoD), IPTV, Voice-over-IP (VoIP), etc. Thus the data that needs to be transported from and to different nodes through a meshed network is increasing. This may cause capacity and performance issues on the serving nodes, which leads to the need of scaling. A commonly used technique is to group the serving nodes into a cluster, but still offer the service over a single access point (e.g., well known address). By doing so, the clients will still reach the service the same way as before. For this to work, a solution was needed to cleverly distribute the demand among the server cluster members. This functionality is provided by the load balancer: it tries to share the load within the cluster [1]. The packets transported over the Internet can be viewed as part of a session defined by the endpoints (e.g., source and destination addresses, port numbers).

The load balancer should be able to identify the different sessions (or flows) and should direct the packets belonging to it to the same server within the cluster. When the load balancer deals with flows, it has a dual task: it should both balance the load among the served output ports (assuming that each port leads to a different server) and to evenly distribute the amount of traffic among these ports. The problem is that the carried traffic volume might differ from flow to flow, thus it is not enough to focus on the per-flow traffic distribution. It is neither acceptable to focus solely on the equal traffic load distribution, because then different packets from the same flow would be sent out on different ports, potentially to different servers.

The authors are with the HSN Lab, Department of Telecommunication and Media Informatics, Budapest University of Technology and Economics, Budapest, Hungary (email: simon@tmit.bme.hu)

Most of the theoretical models that address the load balancing problem try to provide a solution optimizing the resource usage of the control process and/or focus on the long term stability of traffic load distribution. (see the rest of this Section). Nevertheless, the proper operation of a load balancing mechanism is vulnerable to the initialization of the mechanism itself, as we learned it during the implementation and testing of a stateful load balancing proposal (called Round Robin Bloom Filter) optimized in terms of resource usage. In what follows we introduce the reader into our motivation to work with and the environment in which we implemented the particular load balancing solution. Later on in the paper (see Section IV.C) we show how does this initialization problem (named *startup transient*) manifest in a backbone network by conducting dedicated experiments with our implementation. Further on we propose a solution to alleviate this problem and discuss its applicability using further experiments. Thus the main contribution of our paper is to show how the startup transient issue was handled, as it has a crucial role in sustaining the balance between the dual role of load distribution and flow integrity preservation of a stateful load balancer.

When we searched for potential environments to design and deploy a load balancer, we quickly converged to a decision on selecting Software-Defined Networking (SDN), as it is widely recognized as an enabler of dynamic network behavior to adapt to changes in demand [2]. The use of SDN to respond to increasing load is a natural choice and has already been investigated by the networking community. Load balancing in SDN can be achieved in different ways, one of which is to use Finite State Machine (FSM) models with defined network policies described in [3]. Nevertheless, the memory footprint and the required reaction speed for such solution when we have to control a backbone link of the networks calls for the use of new mechanisms. We oriented ourselves towards probabilistic techniques, as many of the network solutions today utilize them to reduce information processing and networking costs. Having millions of data elements in any network it became increasingly important to develop efficient solutions for storing, updating, and querying them. One great idea introduced by Bloom filters (BF) is that by allowing the representation of the set of elements to lose some information, the storage requirements can be significantly reduced [4]. The BF is a space-efficient probabilistic data structure that supports set membership queries. This data structure provides a probabilistic way to represent a set that can never have false negatives (saying that an inserted element is not in the set) but can have false positive returns (saying that an element is part of the set when in fact it is not).

Networking applications of different BFs emerged back in the late 90s. Broder and Mitzenmacher published a survey [5] on network applications of BFs in 2004. A very recent survey [2] by Tarkoma et al. from 2012 reviews over 20 BF variants and their applications for caching, peer-to-peer systems, routing and forwarding, and measurement data summarization. For our work, the interesting application field of Bloom filters is the flow based load balancing.

Regardless of the wide range of different BF flavors, we found that apart of the proposal of Szabo in [6] no suitable BF adaptation for efficiently applying actions consistently to an event stream, based on past decisions assigned to events within a time window. This solution retains the control states assigned to subset of events according to their arrival time slot up to a time window. The Round-robin Bloom filters (RRBF) proposed there would suit well the server farm load sharing or the path load balancing scenarios. An alternative proposal was the Time-Decaying Bloom filter (DBF) [7], which uses bounded counters in each filter bit position. It increments by 1 at the hash positions of the tested/inserted item and decreased, and decreases it periodically each counter. Two proposals were made to add duplicate flow detection to Bloom Filters. Shen and Zhang in [8] proposed to use a DBF and a BF in pair together with a counting sliding window, while Changling et al. in [9] used a time-based sliding window together with a Round robin Buddy Bloom Filter structure. Nevertheless, both these proposals omit the possibility to distribute flows over multiple filters. A different approach from the above ones, using the adaptive highest random weight (HRW) method to account for the uneven flow size popularity, is described in [10].

Note that it is possible to extend the BF scheme of [6] to counting sliding windows. Counting filters provide a way to implement a delete operation on a Bloom filter without recreating the filter afresh. Counting filters were introduced by Fan [11], proposing to extend the filter positions from a single bit to an n-bit counter. These filter variants require more memory space than the basic Bloom filters, as it have to store the value of the counter.

In the following Section II we present the Round-Robin Bloom Filter (RRBF) analysis, summing up those parts of the RRBF proposal that are interesting for our work. We designed and implemented a flow based load balancer solution using RRBF supporting plug and play deployment, the details of design and implementation being presented in Section III. We ran a set of experiments in an emulated environment. In Section IV we discuss the startup transient issue occurring at the time when starting up or connecting the load balancer to the network and the evaluation of our solution to this issue. Finally Section V concludes our work.

II. ROUND-ROBIN BLOOM FILTER

A. Background

The accuracy of the Bloom filter depends on the filter size (m), the number of hash functions (k), and the number of elements included (n). The more elements are added to a Bloom filter, the higher the probability that the query operation reports false positives. A Bloom filter requires space O(n) and can answer membership queries in O(k) time. The below **TABLE 1** examines the behavior of the three key parameters when their values are either decreased or increased.

TABLE 1
BLOOM FILTER KEY PARAMETERS

Bloom Filter parameters	Increase
Number of hash functions (k)	More computation, lower false positive probability as $k \rightarrow k_{opt}$
Size of filter (m)	More space is needed, lower false positive probability
Number of elements in the set (n)	Higher false positive probability

Increasing or decreasing the number of hash functions towards k_{opt} we can lower the false positive probability but the computations for the insertions and lookups will increase. The cost is directly proportional to the number of hash functions. A larger filter will result in fewer false positives.

The calculation of false positive probabilities and the optimal number of hash functions for that is derived in [6]. It is shown that the false positive probability decreases as the size of the Bloom filter (m) increases, and it increases as more elements are added (n). In order to maintain a fixed false positive probability, the length of a Bloom filter must grow linearly with the number of elements inserted in the filter. The optimal Bloom filter size (m) for the expected number of elements (n) and false positive probability (p), is described in [6].

B. Round-Robin Bloom Filter

Once a member is added to a BF, it cannot be deleted – during the lifetime of a BF, after a certain operation time it starts to be inefficient, because lots of expired flows are still referenced in the BF. In order to improve the applicability of BFs, several mechanisms were proposed to allow deleting members from BFs, as presented in the previous section. One of the most efficient solutions is the Round-Robin Bloom Filter (RRBF).

The design of the RRBF is presented based on [6]. The operations over the RRBF are defined as follows:

- *Membership query*: we query for the existence of an event from the oldest to youngest filters and if a match is found then a corresponding action is executed without further queries.
- *New element insert*: if no match is found during the membership query, then the event is inserted into the youngest filter.

- *Window jump*: before entering into the next time slot, we reset the oldest filter and make a jump to select the next youngest and oldest filters.

In the following we explain in brief the operation of a RRBF, summing up the detailed description from [6]. Fig. 1 presents an example of the RRBF operation. In each time slot, only one filter (red) will be used to insert new elements, while the others (cyan) are queried in decreasing age order. After N time slots, the oldest filter is reset and the next oldest and youngest filter will be selected in a round-robin fashion. When the event $e(\tau)$ arrives in the 7th time slot, its hash is tested for containment in filters v_4; v_5; v_1; v_2 sequentially. If a match is found, the corresponding A_i action is executed and further querying is stopped. If no match is found, the event is inserted into filter v_3 (the latest filter marked with red) and the corresponding A_3 action is executed.

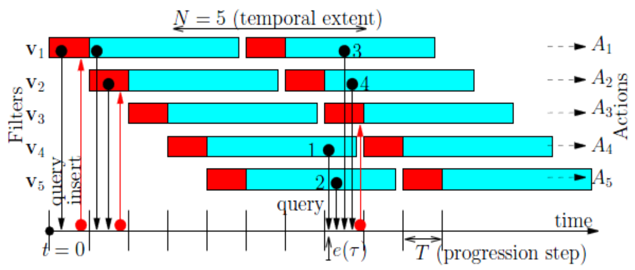

Fig. 1 Round-Robin Bloom Filter design (source: [6])

If we treat the incoming events as IP packets and their fingerprints being calculated from their IP headers, which we can call individual flows, then we can see that a consecutive number of packets of the same flow will have the same action executed, at least for *(N-1)*T* times, after which a reroute can happen. Assigning different output ports for the different actions means that we can create a load balancing solution among alternative paths.

III. FLOW BASED LOAD BALANCER: DESIGN & EXPERIMENTS

In the previous sections we introduced RRBF, as a candidate solution to support efficient load balancing for large number of flows – typically in the backbone links. In this section we present an implementation of this system. Since the packet based networks are very dynamic and diverse, the implementation should be flexible, easily configurable and customizable. During the traffic engineering process the controlled flows, the available paths and the associated ports may change in time, which results in the request to change the RRBF configuration and the port assignments.

SDN has been introduced in the last decade in computer networking to address this issue, separating the control and data planes from each other, and putting an open interface in-between [2]. The cornerstone of the SDN framework is the control protocol over this interface that commands the switches, called OpenFlow. Having freedom in the control plane it gives room for innovation and makes the implementation of such a complex control process practically feasible.

A. Design

We show the design of the load balancer prototype supporting plug and play deployment by eliminating the startup transient at system start time. The issues presented in Section I. were considered during implementation, and the prototype was built in a way that the RRBF solution could be verified with different traffic characteristics. The system consists of the following modules:

- *Emulated environment* – provided by Mininet [12].
- Emulated *OpenFlow controller* – used to configure the switch with the necessary ports, tables and actions. Each table will have a different role, e.g., to match IP packets, select Bloom filters or select output ports.
- Emulated *OpenFlow switch* [13] – implemented with internal RRBF system parameters, e.g. expected number of ingress flows, desired false positive probability, etc.

During the implementation of these modules we reused the open source Bloom filter code of Virkki [14] and the MurmurHash2 code of Appleby [15].

Fig. 2 shows the designed internal structure of the RRBF load balancer. There is only one flow table with one flow entry. The input port is passing all ingress packets to the flow entry. All packets are matched against the condition, eth_type=0x800, that is only IP packets will be processed further. The flow entry will pass the packets to the first group entry in the group table. Each bucket of the group entry contains a Bloom filter. Based on the Round-robin Bloom filter algorithm one bucket will be selected. Each bucket has the same action, passing the packets to the next group entry. The next group entry has a number of buckets, each associated with an output port. Based on the output port selection algorithm the packets will be sent out from one of the output ports.

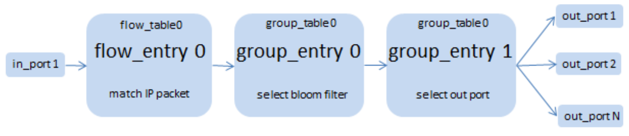

Fig. 2 Internal architecture of the RRBF based load balancer

B. Control mechanism

The following system parameters are implemented and can be controlled:

- Number of Bloom filters (*B*) – it's set by the controller via the OpenFlow protocol.
- Number of output ports (*P*) – it's set by the controller via the OpenFlow protocol.

- Packet type – it is set by the controller via the OpenFlow protocol.
- Period (*T*) – time period in seconds after which there is a shift in Bloom filters, and the next one will be the youngest accepting new flows.
- Epoch (*Ep*) – time period for the youngest Bloom filter until it becomes the youngest again, i.e. *(B-1)*T* seconds.
- False positive probability (*E*) – used to calculate the number of hash functions (*K*) for the Bloom filters.
- Number of elements (*N*) – number of expected flows per Bloom filter and it is used to calculate the size (*M*) of the Bloom filters.
- Transient packets (*Tr*) – used to set the number of packets to be used at the beginning for flow transient elimination.
- Redistribution threshold (*R*) – used to decide if new output port can be associated with a filter depending on the number of the ingress packets in the last period (*T*) compared to the number of ingress packets in the last epoch (*Ep*).

The first two parameters define the structure of the RRBF mechanism, whereas the *T* and *Ep* parameters define its dynamics. Once the number of BFs within the scheme is fixed, the memory footprint of the RRBF depends on the *E* and *N* parameters.

The effect of the *Tr* and *R* parameters determine the efficiency of the load balancing process. *Tr* affects the reaction of the mechanism at startup, as detailed in Section III.E. Note that according to the original proposal we should only select a new port for a BF if all its flows terminate, but in reality this rarely happens. That is why we need to make a compromise between the goal of keeping the flows on the same port and having equal traffic distribution within the output ports. This tradeoff is controlled by the *R* parameter and its details are discussed in Section IV.D.

C. *Bloom filter selection*

The implemented Bloom filter selection algorithm is compliant with the OpenFlow standard. The publicly available BF implementations were re-used and enhanced with the RRBF algorithm.

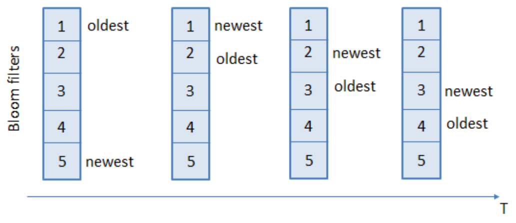

Fig. 3 Shifting of Bloom filters after each *T* period

For each ingress packet, after being matched as an IP packet by the flow entry and passed to the first group entry, we calculate the fingerprint as the XOR of source and destination IP addresses and then we check if it's time to switch to the next youngest filter (see Fig. 3). If not, then we go through all the filters, from oldest to youngest, and do a membership query operation for the given fingerprint. If the fingerprint is present in any of the filters, we select that one; otherwise we do an insert operation to the youngest one. On the other hand, in case the period (*T*) has ended, we clear the oldest filter and recreate it, then shift the youngest filter to the next one and do the same as described above. Each bucket in the first group entry containing a filter, has the same action of forwarding the IP packet to the next group entry.

D. *Output port selection*

The implementation was designed such as the output port selection supports our load balancing goal. After the IP packet arrives to the second group entry, which has a number of buckets each having an action of forwarding the packet to an output port, it will be sent out on the selected port. Important to know, that after each period (*T*), when the youngest filter is shifted, we also search and select the least loaded port and assign it to the filter. So, at any given time, each filter has a port assigned to it which are continuously sending out packets, then at timeout, the youngest filter might get a new port assigned based on the number of transmitted packets on each port.

E. *Startup transient elimination*

As introduced in Section I, the startup transient problem must be solved before load balancing solutions can be deployed to support packet flows. Startup transient period is present at the time of connecting (or activating) the load balancer to the network. The input traffic can contain a lot of already active flows, and it is a concern that in the first period (*T*), the youngest filter can be filled with most of the ingress flows, and the system can remain unbalanced, because the first filter will not be cleared until the first epoch (*Ep*) has ended. In practice this means that all incoming flows during the first period (*T*) will be registered in the first youngest filter, and all other filters will remain empty. Therefore the youngest filter may fill up in a short period of time and so the false positive probability can increase significantly. The other issue introduced by the startup transient is that the flows present at the startup of the load balancer will always belong to the same filter. The system will get balanced only after most of the original flows decay, which can take time.

To compensate for this *startup transient* problem, and to make plug and play deployment possible, the startup transient needs to be eliminated. In this paper we propose a solution that for a configurable amount of packets (*Tr*) at the beginning, the system will always rotate the youngest filter, as it would happen at the end of each period (*T*). The Bloom filter selection algorithm will be called for each consecutive packet, assuring that the existing flows will be associated to their original filters, and only the new flows will be stored in the youngest filter.

Rotating the filters at every incoming packet gives the possibility for the incoming flows to be evenly distributed within the filters, ameliorating the effect of the startup phenomenon. The solution can be further enhanced to achieve even better distribution of flows within the filters, e.g. by rotating the youngest filter only if the respective incoming packet was belonging to a new flow. This way we make sure that each filter will register new flows.

IV. EVALUATION OF THE LOAD BALANCER

A. *Testing scenario*

A set of experiments were conducted with different system parameter combinations to test our proposal. The goal was to prove that the system is performing as expected under real-life traffic scenarios. We simulated the hot deployment scenario by running real-life traffic samples through the system. The publicly available Internet traffic traces were used provided for the research community from the SimpleWeb project [16]. The sample traces contain ~8000 to ~16000 TCP/IP packets per period (*T*), and the ratio of new flows vs. old flows is around 35% to 92% per period (*T*) depending on the period length.

In this paper we illustrate our measurement results using three traffic traces selected from these. Fig. 4 shows the flow distribution of sample trace #1 we use in this paper to explain our design decisions. This trace had a 14534 pckt/sec average packet rate. It can be seen that on average ~65% of the flows in any period (*T*) were already active at least in the previous period (*T-1*), as well. As for trace #2, ~8% of the flows present in a particular period were also present at least in the previous period and the average packet rate was 7804 pckt/sec. Trace #3 was only used for initial tests, conducted to calibrate the N and E parameters. It contains ~1000 flows and ~50% of the flows are longer than 1sec.

When startup transients are eliminated, the *Tr* parameter has a value of 10000, meaning that the first seconds (depending on the actual load) are used only to "initialize" the RRBF.

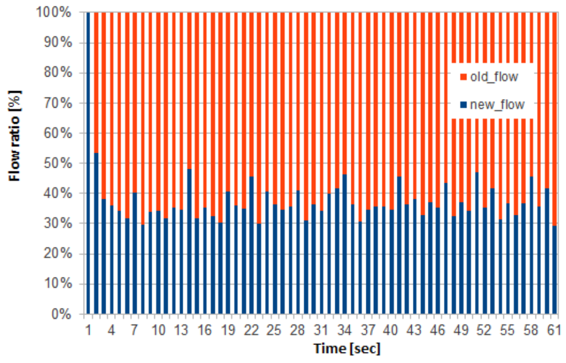

Fig. 4 Ingress flow distribution for trace #1

B. *Round Robin Bloom filters without periodical state reset*

We evaluate the effectiveness of the load balancing at packet level, because that increased the execution speed of our experiments. This does not affect the generality of our results, because the decision to associate a new port or not to the filter at the beginning of a new period is similar in both cases. The only difference is that instead of counting the number of packets per port, we would have to count the bytes contained within the packet. This would neither increase the granularity of the process.

TABLE 2
EXPERIMENTS WITH DIFFERENT SETTINGS OF THE NUMBER OF EXPECTED FLOWS (*N*) AND FALSE POSITIVE RATE (*E*)

RRBF with two BFs	**N=100, E=0.01%**	**N=1000, E=0.01%**	**N=1000, E=1%**
BF1 egress packets	37547	35652	35499
BF2 egress packets	40943	42838	42991
Egress packets due to false positive membership query	13593	0	1236
BF1 egress flows	637	602	599
BF2 egress flows	310	469	465
Egress flows due to false positive membership query	**124**	**0**	**7**
Total flows	1195	1071	1078

We start with a basic experiment using capture #3 and consisting of two Bloom filters each assigned to a separate output port. We want to examine the flow distribution process, focusing on the false positive rate depending on the BF configuration. After every incoming new flow the roles of the oldest and youngest filters were switched, so that each new flow could be stored in the next filter. Still, the bits of the filters are never reset, preserving the previous state, the memory of the all flows inserted earlier. That is the reason why we refer to these experiments as the ones were the periodical *state reset* was avoided. The results, shown in Table II above, prove that the false positive probability can be influenced by the Bloom filter parameter values. Because the filters were never reset, if the number of expected flows (*N*) was not high enough, false positives started to appear. We counted the number of egress packets and flows where the membership query resulted in false positive matches. This happened in case *N* was lower than the total number of ingress flows, or in case that *E* was too high.

These results confirm that RRBF with the original BF (i.e. without periodical reset of the BF bits) is not working and it is needed the mechanism proposed in [6], indeed.

C. *RRBF and startup transients*

In this sub-section we illustrate the effect on the original RRBF proposal of the flow transient problem. Based on our experiments we set the main parameters of the RRBF as follows: we used 10 hashes ($K = 10$), the false probability rate was reduced to $E = 0{,}01\%$ and the number of elements was set $N = 1000$. These parameters were used in all experiments presented from now on in this paper. The results in this sub-section were obtained by using traffic trace #1.

Fig. 6 shows the number of registered new flows per Bloom filter (*bf_flow N*) in every period (*T*). In the beginning the first

Bloom filter from a RRBF mechanism with five BFs will carry more flows than the other ones, because it “keeps locked in” the long flows and so the flow distribution gets unbalanced. After an epoch has passed the first filter will be the youngest again and it will keep carrying its old flows.

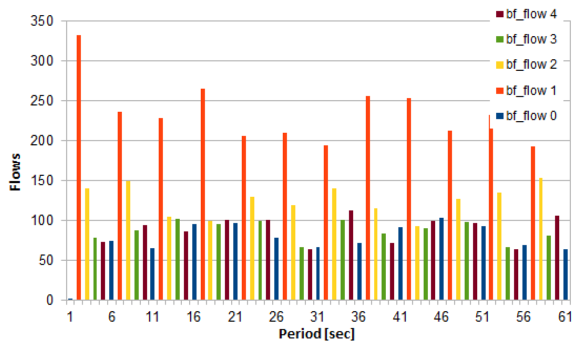

Fig. 5 Flows per Bloom filter with periodical state reset

Note that this impacts the per-packet load balancing performance of the mechanism, as well (Fig. 6). Within each epoch the port associated to the youngest filter will be loaded with the packets of the heaviest flows. Since we have 5 BFs and *T* = *1*, the curve breaks at each 5 seconds.

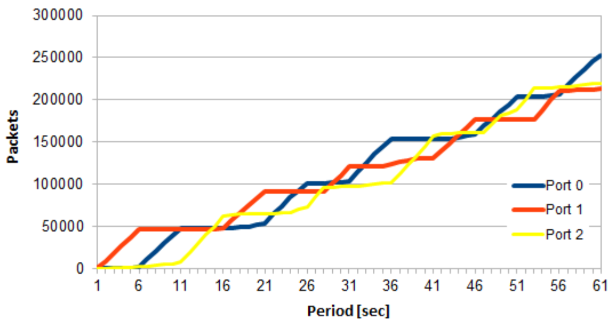

Fig. 6 Packets per output port with periodical state reset

These results illustrate our motivation to handle the transients during the startup.

D. RRBF with elimination of the startup transient

The experiments discussed in this sub-section deploy our solution to eliminate the startup transient phenomenon. For this we used our implementation of the method proposed in Section III. E and traces in Section IV. A. The implementation features the initialization until the first *Tr* packets are received, as described earlier. Additionally, we also use a mechanism that is controlled through the *R* parameter. This mechanism measures the ratio of the incoming packets during the last period *T* and those received during the whole epoch. If it is less than *R*, then we consider that the BF can be associated to a new port (i.e., the least loaded one), otherwise we do not change the association. After several test runs and based on empirical results we set the value of *R* to 10%.

Fig. 7 below shows the number of registered new flows per Bloom filter in every period (*T*). During the first period the incoming flows are evenly distributed within the Bloom filters, and this distribution keeps the system balanced throughout the measurement. Long flows (longer at least than one epoch) are carried by multiple filters.

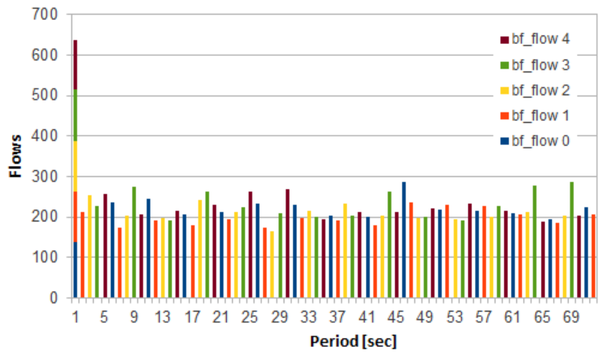

Fig. 7 Flows per Bloom filter with elimination of the startup transients

Applying our proposal also improves the quality of load balancing, as well. As seen in Fig. 8, the number of packets sent over the three ports has more even distribution.

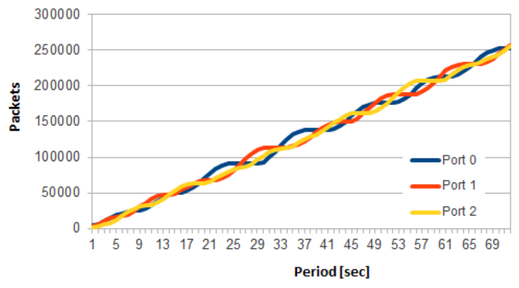

Fig. 8 Packets per output port with elimination of the startup transient

We repeated the experiment using trace #2, as well. The results were similar. Based on the theoretical results we started from this was expected for the flow distribution. But it proved that it works for the load distribution, too, as shown in Fig. 9.

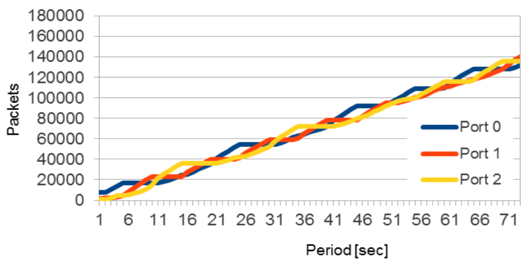

Fig. 9 Packets per port with elimination of the startup transient (trace #2)

Note that by increasing the number of BFs we obtain finer granularity and we can make a balance between the two goals: distribute flows among different ports and evenly distribute the traffic volume. By increasing *R* we have better load balancing, but this will affect some longer flows, as they might be distributed among several output ports. To remedy that, we can increase the number of filters from 5 to 15, while keeping *R* down at 10%. We run an experiment, where the only change compared to the one analysed in this sub-section above was the use of 15 Bloom filters, and it slightly better balanced the load of the flows. Nevertheless, if there were elephant flows in the traffic mix whose lifetime exceeded the epoch duration, then this change could not fully eliminate the effect of it.

We also tested higher R parameter values (e.g., 30% instead of 10%), which results in more aggressive port selection behavior, leading to better load sharing among the ports. Nevertheless, this affects more often the longer flows, forwarding their packets over different ports.

V. CONCLUSIONS

We designed an OpenFlow compatible flow based load balancer with Round Robin Bloom filters supporting plug and play deployment scenarios. We demonstrated in an emulated SDN environment, using real-life traffic traces that the solution can eliminate the startup transient problem during initialization and the system remains balanced, meaning that the ingress flows are more evenly distributed within the Bloom filters.

Our future plan is to further investigate our solution in more complex traffic conditions. Our main focus will be on separating elephant flows that carry large traffic volumes and have longer duration. We are considering the possibility to deploy separate RRBF systems in parallel and use on-the-fly traffic classification (e.g. short – HTTP, long – video) by controller to feed the different systems with such homogeneous traffic.

ACKNOWLEDGMENT

The authors thank to Robert Szabó for his valuable comments during the implementation work of the Round Robin Bloom filters.

REFERENCES

[1] Cardellini, V., Colajanni, M. and Philip, S.Y., 1999. Dynamic load balancing on web-server systems. IEEE Internet computing, 3(3), p.28-39.

[2] Open Networking Foundation: Software-defined Networking: The new norm for networks, (accessed on April 2016).

[3] Yuanhao Zhou, Li Ruan, Limin Xiao, Rui Liu, “A Method for Load Balancing based on Software Defined Network”, Advanced Science and Technology Letters Vol.45 (CCA 2014), pp.43-48.

[4] S. Tarkoma, C. Rothenberg, and E. Lagerspetz, “Theory and Practice of Bloom Filters for Distributed Systems”, IEEE Communications Surveys & Tutorials, Vol. 14. no. 1., 2012.

[5] A. Broder and M. Mitzenmacher, “Network applications of bloom filters:A survey,” Internet Mathematics, vol. 1, no. 4, pp. 485–509, 2004.

[6] R. Szabó, “A Round-Robin Bloom Filter for Stateful Control over Event Streams”, 2013 IEEE 4th International Conference on Cognitive Infocommunications (CogInfoCom), 2013.

[7] K. Cheng, L. Xiang, and M. Iwaihara, “Time-decaying bloom filters for data streams with skewed distributions,” in 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications, 2005. RIDE-SDMA 2005, Apr. 2005, pp. 63–69.

[8] H. Shen and Y. Zhang, “Improved approximate detection of duplicates for data streams over sliding windows,” Journal of Computer Science and Technology, vol. 23, no. 6, pp. 973–987, 2008.

[9] Z. Changling, X. Jianguo, C. Jian, Z. Bei, and L. Feng, “Approximate discovery of service nodes by duplicate detection in flows,” China Communications, vol. 9, no. 5, pp. 75–89, Jul. 2012. [Online]. Available:
http://www.chinacommunications.cn/EN/abstract/article 7912.shtml

[10] Chengcheng G. et al., „A load-balancing scheme based on Bloom Filters”, in. proc. of the IEEE Second International Conference on Future Networks, Washington DC, USA, 2010, pp. 404-407.

[11] Fan L., Cao P., Almeida J., Broder A., "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", IEEE/ACM Transactions on Networking, 8 (3): 281–293, 2000.

[12] Mininet, http://www.mininet.org/, (accessed on April 2016).

[13] CPqD OpenFlow switch: http://cpqd.github.io/ofsoftswitch13/ (accessed on April 2016).

[14] Jyri J. Virkki’s Bloom filter: https://github.com/jvirkki/libbloom (accessed on April 2016).

[15] Austin Appleby’s MurmurHash2: https://sites.google.com/site/murmurhash/ (accessed on April 2016)

[16] SimpleWeb, http://www.simpleweb.org/wiki/Traces/ http://cpqd.github.io/ofsoftswitch13/ (accessed on April 2016).

Örs Szabó has graduated as an MSc software engineer from the Budapest University of Technology and Economics. During his last student years he worked as a part-time employee in the field of packet-switched voice communication over 4G mobile networks at Ericsson Hungary. Currently he works at the same company as a system engineer.

Csaba Simon is a software engineer and he earned his PhD in 2012 at the Doctoral School of Informatics of the Budapest University of Technology and Economics and currently works as an assistant professor at the Department of Telecommunication and Media Informatics of the same university. His research area includes the topics of Future Internet, 5G networks and cloud systems.