DataSense: Min Overhead, Max Accuracy

Saad Naveed Ismail, Fareeha Irfan Khawaja
Computer Science Department, LUMS
Email: {14100055,15100235}@lums.edu.pk

***Abstract*—Software Defined Networking promises to simplify network management tasks by separating the control plane (a central controller) from the data plane (switches). OpenFlow has emerged as the standard for communication between the control and data plane. Apart from this, it also provides a high level interface to collect different types of statistics from the data plane. Network applications can use this high level interface to monitor network status without being concerned about the low level details. This mechanism is implemented as a pull-based service, i.e. the controller has to query the switches for statistics and then in turn, the switches respond with a reply corresponding to the type of statistics request. The frequency of polling and number of switches polled determines the accuracy and network overhead. An important aspect is to gather statistics with minimal overhead, while maintaining high accuracy so then that the results would be relevant. In this paper, we focus on leveraging the structure of k-ary FatTrees and use that to design an algorithm for querying a subset of the switches while maintaining accurate statistics of the whole k-ary FatTree topology. The accuracy of our solution is demonstrated through emulations in Mininet.**

I. Introduction

Monitoring statistics of the network is crucial for network management. It requires accurate statistics while not incurring a large overhead. In data centers we can use these statistics for load balancing, traffic engineering, enforcing Service Level Agreements (SLAs), and for even fault detection (thus tolerance). A well designed network monitoring framework should provide the management applications with a wide selection of network metrics.

Recently, Software Defined Networking (SDN) has emerged as new paradigm that promises to facilitate network programmability and ease management tasks. SDN proposes to decouple the control plane from the data plane. The OpenFlow protocol has been accepted as the standard interface between the control and data plane. OpenFlow provides statistics collection primitives at the controller. The controller could either poll a switch to collect statistics on the active flows or it can request a switch to push flow statistics (upon flow timeout) at a specific frequency.

PayLess [1] is a recently proposed network monitoring framework for SDN that adaptively changes the inter-pinging time of flow statistics request from all switches based on the differences in byte count. They have shown there work to be very effective in reducing network overhead while providing highly accurate results. However, one problem is that in case of large number of flows being sent from one host to another, the network overhead caused by requesting and getting responses for flow statistics would be very large.

The scenario we are concerned about is that of a data center. Where the network topology is that of a FatTree [2], while the traffic scenario is that as found in DCTCP [3]. In this paper, we propose DataSense, a network montoring scheme for FatTree data centers that leverages the structure of the FatTree topology to provide accurate statistics while abstaining from requesting for flow statistics.

The rest of the paper is organized as follows. We describe the details of DataSense and insights used while designing it in Section II. An analysis of the reduction in overhead by DataSense is presented in Section III. The related work is discussed in Section IV followed by some future work in Section V. We offer concluding remarks in Section VI.

II. DataSense Scheme

We define the overhead as being the amount of control traffic when querying and receiving responses. Accuracy would be defined as how closely our results match that of querying all switches/links for their statistics.

A. Background

PayLess [1] proposes to adapt the inter-pinging time based on the changes of byte count of flow statistics. If the difference is large, then it would decreases the inter-pinging time to maintain high accuracy. In case there is very little (or no) difference in byte count, then the inter-pinging time would increase so it would poll less often and therefore reduce the overhead. While their results are good, we would prefer to stay away from using flow statistics. This is because,the network overhead would be dependent on the number of flows in the data center. In data center networks, there are a couple of long flows but there are a large number of short flows (especially when in a cloud data center, the hosts would send back responses to user query) that would increase the overhead significantly.

B. Basic Idea and Assumptions

DataSense accurately gets the link utilization of all links by querying a subset of the links for their utilization and then using this to infer the utilization in the links that we have not queried for statistics. The principle: all that flows in to a switch (or node) equals to all that flows out of the switch (or node).

To modulate flows_{in} equals to flows_{out} without using flow statistics, we can think of a flow as a sequence of bytes. Thus, we can take our principle to be the numBytes_{in} of a switch in an interval would equal the numBytes_{out} of a switch in

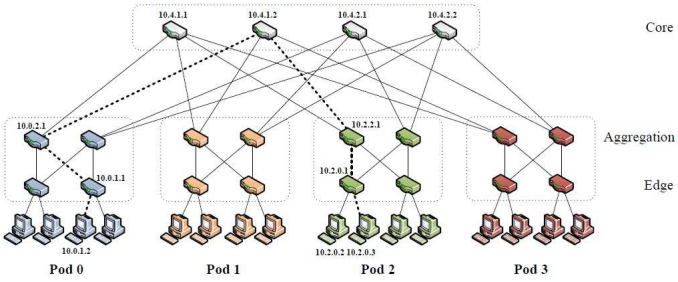

Fig. 1. k-ary FatTree design as proposed in [2], where k=4

that same interval. Thus by using simply sending port/queue statistic requests to switches, we can figure get the transmitted and received byte count in a switch. After that, by knowing the inter-pinging time and the capacity of the link, we can figure out utilization of the link.

Our scheme currently is slightly constrained in that, it makes the following assumptions about the data centers:

- The data center topology is a k-ary FatTree as defined by Al-Fares et. al. in [2]. We are sticking with this because the FatTree topology is a very common one in data center networks.
- In our data center, no link or switch will fail. Thus, we will be getting statistics from a fixed number of switches and their ports/queues. Their will not be variations due to a link connected to a port failing or due to a switch failing.
- There will be no delays, whether they are transmission delays, propagation delays or queueing delays. We make this assumption so then that our principle of flows$_{in}$ equals flows$_{out}$ would hold accurately, delays would add a bit of inaccuracy. We give a small mention about this in our Future Works in Section V.

C. Scheme

Our scheme was incrementally developed on a case-by-case basis. Each case adds another level of complexity w.r.t. the way traffic may flow and then correspondingly it adds another layer of complexity to our scheme because case by case our scheme develops to take into account more and more different ways that the traffic could flow in the data center.

In a k-ary FatTree [2], each pod has $k/2$ Aggregation (Agg.) switches and $k/2$ ToR switches. Under each ToR switch, there are $k/2$ host physical machines. Each Agg. switch is connected to $k/2$ Core switches and all of the $k/2$ ToR switches in its own pod. And each ToR switch is connected to the $k/2$ host physical machines under it and all the $k/2$ Agg. switches in its own pod (in Figure 1 you can see this for a 4-ary FatTree).

The base case way to get all statistics is query all ToR switches for port statistics on its upward and downward ports (for Agg. links and ToR links statistics, respectively). And get the Agg. switches to query all upward ports (for Core link statistics). The port statistics response contains both transmitted and received byte count fields. However, it also contains a few other fields that increase the size of the response but are not needed by us to get utilization of the link.

One change we can make here is that we query for queue statistics. It only contains the transmitted byte count field, not a received byte count field. So, we would make the ToR switch query for upward queue statistics to get the upward utilization of the Agg. link and the corresponding Agg. switches for downward queue statistics so then that we get the downward utilization for the Agg. link.

Overhead for 1 port statistic is:

bytes$_{request}$ + bytes$_{response}$ = 8 + 104 = 112.

Overhead for 2 queue statistics is:

2 x (bytes$_{request}$ + bytes$_{response}$) = 2 x (8 + 32) = 80.

However, the ToR, switch would have to query for port statistics so it can also get the byte count received because it is not possible to get the byte count transmitted from the host.

Case 1: All traffic that flows is from a host under one pod to a host under a different pod.

For every Agg. switch we query all $k/2$ ports for query statistics that are connected to ToR switches but only $k/2-1$ ports for query statistics that are connected to Core switches.

Case 2: However, traffic does not only flow in the scenario described in Case 1. Traffic can also remain within a pod. Where the traffic's destination is a host under a different ToR switch from the traffic's source host. This way, traffic would flow up from a ToR switch to the Aggregation switch, but them come down to another ToR switch in the same pod.

In this case, the change is that we query queue statistics the Agg. switch along the link that is connected to the $k/2$th Core switch to figure out how much it has transmitted to it (up utilization). But it infers amount received (down utilization) across the $k/2$th Core link via calculation.

After this, we can go about making one more optimization (**Improved Case 2**). Each ToR switch queries only $k/2-1$ ports that are connected to Agg. switches.

Case 3: After making the 2^{nd} optimization mentioned at the end of Case 2. Another scenario impacts our accuracy. Traffic could be traversing under the same ToR switch. So it goes from one host under a ToR switch to another host under the same ToR switch.

The solution for this would be the same as that provided in Case 2. The ToR switch would query the $k/2$ Agg. switch to get the amount transmitted (up utilization) and then it infers the amount received (down utilization) across the $k/2$ Agg. link via calculation in the same manner as done in Case 2.

III. Analysis

Stats for OpenFlow [4] and FatTree [2] are present in Table I.

Size of flow stats request	8 bytes
Size of flow stats response	88 bytes
Size of port stats request	8 bytes
Size of port stats response	104 bytes
Size of queue stats request	8 bytes
Size of queue stats response	32 bytes
Number of pods	k
Number of links at each level	$k^3/4$
Number of ToR , Agg. switches per pod	$k/2$, $k/2$
Number of ToR , Agg. switches in data center	$k^3/4$, $k^3/4$
Number of Core switches	$k^2/4$

TABLE I
TABLE OF DIFFERENT STATISTICS

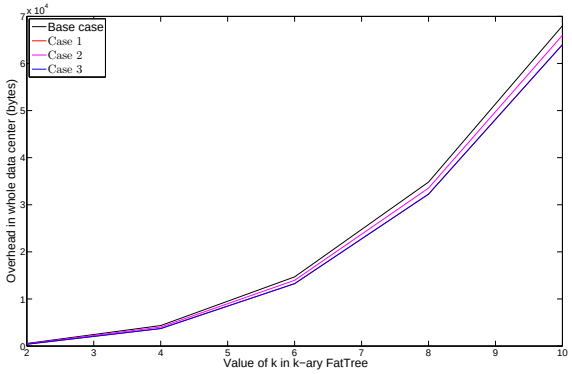

Fig. 2. Overhead for our scheme, case-by-case

In this section we would be analyzing the reduction in overhead by using our scheme. Values used can be referred to from Table I.

Base Case:

Overhead of a ToR or Agg switch getting Port statistics is 112 bytes. We make 3 x $k^3/4$ requests.

$$Base_{overhead} = 112 * 3 * k^3/4 = 84 * k^3.$$

Improved Base Case:

Overhead of a ToR or Agg switch getting Queue statistics is 40 bytes. We make 4 x $k^3/4$ requests.

Overhead of a ToR or Agg switch getting Queue statistics is 112 bytes. We make 1 x $k^3/4$ requests.

$$ImprovedBase_{overhead} = (112 * k^3/4) + 4 * (40 * k^3/4)$$

$$=> ImprovedBase_{overhead} = 28 * k^3 + 40 * k^3/4$$

$$=> ImprovedBase_{overhead} = 68 * k^3$$

Case 1:

Overhead of a ToR or Agg switch getting Queue statistics is 40 bytes. We make 2 x $k^3/4$ requests and 2 x $(k^3 - 2 * k^2)/4$ requests.

Overhead of a ToR or Agg switch getting Queue statistics is 112 bytes. We make 1 x $k^3/4$ requests.

$$Case2_{overhead} = 28 * k^3 + 40 * (2 * k^3/4 + 2 * (k^3 - 2 * k^2)/4)$$

$$=> Case2_{overhead} = 28 * k^3 + 10 * (4 * k^3 - 4 * k^2)$$

$$=> Case2_{overhead} = 68 * k^3 - 40 * k^2$$

Case 2:

Overhead of a ToR or Agg switch getting Queue statistics is 40 bytes. We make 3 x $k^3/4$ requests and 1 x $(k^3 - 2 * k^2)/4$ requests.

Overhead of a ToR or Agg switch getting Queue statistics is 112 bytes. We make 1 x $k^3/4$ requests.

$$Case2_{overhead} = 28 * k^3 + 40 * (3 * k^3/4 + 1 * (k^3 - 2 * k^2)/4)$$

$$=> Case2_{overhead} = 28 * k^3 + 10 * (4 * k^3 - 2 * k^2)$$

$$=> Case2_{overhead} = 68 * k^3 - 20 * k^2$$

Improved Case 2:

Overhead of a ToR or Agg switch getting Queue statistics is 40 bytes. We make 1 x $k^3/4$ requests and 3 x $(k^3 - 2 * k^2)/4$ requests.

Overhead of a ToR or Agg switch getting Queue statistics is 112 bytes. We make 1 x $k^3/4$ requests.

$$ImprovedCase2_{overhead} = 28 * k^3 + 40 * (1 * k^3/4 + 3 * (k^3 - 2 * k^2)/4)$$

$$=> ImprovedCase2_{overhead} = 28 * k^3 + 10 * (4 * k^3 - 6 * k^2)$$

$$=> ImprovedCase2_{overhead} = 68 * k^3 - 60 * k^2$$

Case 3:

Overhead of a ToR or Agg switch getting Queue statistics is 40 bytes. We make 2 x $k^3/4$ requests and 2 x $(k^3 - 2 * k^2)/4$ requests.

Overhead of a ToR or Agg switch getting Queue statistics is 112 bytes. We make 1 x $k^3/4$ requests.

$$Case3_{overhead} = 28 * k^3 + 40 * (2 * k^3/4 + 2 * (k^3 - 2 * k^2)/4)$$

$$=> Case3_{overhead} = 28 * k^3 + 10 * (4 * k^3 - 4 * k^2)$$

$$=> Case3_{overhead} = 68 * k^3 - 40 * k^2$$

Savings:

The lower bound on percentage savings is (we get this lower bound by getting rid of the "-40 x k^2" factor):

= (84 - 68) / 84 x 100

= 19 %.

Figure 2 shows the overhead for each case.

IV. Related Work

There exists a number of flow based network monitoring tools for traditional IP networks. NetFlow [5] from Cisco is one of the most prominent ones. JFlow [6] by Juniper Networks is very similar to NetFlow. Both are expensive and incur a large setup cost to be deployed.

OpenSketch [7] which allows for development of more expressive traffic measurement applications while using their three stage packet processing pipeline design. OpenTM [8] focuses on efficiently measuring a traffic matrix using existing

technology. [9] and [10] propose other methods for network monitoring.

There has been an everlasting trade-off between accuracy of statistics collection and resource usage overhead for monitoring in IP networks. We, however, are concerned with data center environments. According, to the best of the authors' knowledge, DataSense is the first of its kind scheme meant for data center networks, and in essence with the FatTree topology.

V. Future Work

Other Topologies: We would like to try out our scheme in other topologies. The core fundamental idea is that flows$_{in}$ equals flows$_{out}$ of a switch. Therefore, even if we try this out in a single rooted tree or other topologies, we would just have to adapt the switches that we query using this same idea of numBytes$_{in}$ equals the numBytes$_{out}$.

Combine with PayLess: We would like to implement DataSense in conjunction with PayLess. Or make a modified version of PayLess that would check for byte counts from port and/or queue statistics instead of flow statistics. And then combine our method with modified PayLess.

Other Factors: Currently our algorithm works excellently with the traffic scenario present in data centers [3]. We would like to incorporate other factors such as failure rates and link/queuing delays. We would be really interested in seeing that if our results would be accurate enough with standard failure rates. Looking at the transmitted and received bytes dropped could help us in this regard, but the validation of this assumption is left as a future work. To get failure rates, we could use insights from [11] and [12]. Taking standard queueing delays, it would be interesting to see that how accurate would our scheme remain. Intuitively, if we ping at approximately average queueing delay, then to maintain very high accuracy, we could do flows$_{out}$ equals flows$_{in}$ of previous ping. Or flows$_{out}$ of current interval equals to an average of flows$_{in}$ of previous interval and flows$_{in}$ of current interval.

Real Testbed: Scenarios in simulated environments like Mininet provide us with an ideal scenario and thus ideal results. We would like to test out in a real test bed scenario, preferably Emulab and see with the lag, failures of switches and links, would our algorithm maintain accuracy.

VI. Conclusion

In this paper, we proposed a scheme, DataSense, and completed an analysis to see how much overhead would be reduced. We find that DataSense does reduce the overhead of statistics collection compared to the base case.

References

[1] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *NOMS'14*.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM'08*.

[3] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *SIGCOMM'10*.

[4] "OpenFlow Switch Specification Version 1.0.0," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf.

[5] "Introduction to Cisco IOS NetFlow," http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html.

[6] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *SIGPLAN-SIGACT'99*.

[7] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *NSDI'13*.

[8] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "Opentm: Traffic matrix estimator for openflow networks," in *Springer'13*.

[9] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "Flowsense: Monitoring network utilization with zero measuremet ghost," in *Springer'13*.

[10] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches," in *HotICE'10*.

[11] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *SIGCOMM'11*.

[12] R. Potharaju and N. Jain, "When the network crumbles: An empirical study of cloud network failures and their impact on services," in *SoCC'13*.