Demand-Aware Flow Allocation in Data Center Networks

Dmitriy Kuptsov
Aalto University/HIIT
Espoo, Finland
dmitriy.kuptsov@hiit.fi

Boris Nechaev
Aalto University/HIIT
Espoo, Finland
boris.nechaev@hiit.fi

Andrei Gurtov
University of Oulu/CWC
Oulu, Finland
gurtov@ee.oulu.fi

ABSTRACT

In this work we consider a relatively large and highly dynamic data center network in which flows have small interarrival times and different demands for the network resources. Taking into account the properties and specifics of such networks we consider the problem of flow placement, i.e. assignment of an outgoing port for flows at each hop from source to destination. Using the characteristics of modern data centers from previous measurement studies, in this work we first simulate the flow allocation using several algorithms with and without global knowledge. We find that in all settings local forwarding decisions are almost as good as decisions made with global information at hand. This finding enables us to propose a fully distributed mechanism that relies only on local knowledge and allows to achieve fair and demand aware flow allocation in the data center network. The mechanism has low complexity and performs better than naive random flow allocation.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Algorithms, Design, Performance

Keywords

Data center, Flow Allocation, Fairness, Rate control

1. INTRODUCTION

Modern data centers comprise tens of thousands computers interconnected between each other with commodity switches. These networks are dynamic in nature, with new flows arriving and departing on sub-millisecond intervals. As several measurement studies indicate [2,5,9] there are two major types of flows in data center networks: those for which meeting deadlines is crucial, and those for which delays can be tolerated. Deadline sensitive flows (we call them *deadline flows*) are commonly produced by real time web applications such as web search and retail, that aggregate data from many locations in the data center. Such flows are typically small, unlike bulky delay tolerant flows (we call them *elastic flows*), which are caused for instance by virtual machine migration. For deadline flows meeting the deadlines is important because many commercial applications and services have user experience SLAs.

Typically, network operators use Equal Cost Multipath (ECMP) to statically stripe flows across available paths. This static flow mapping to outgoing ports does not take into account the current network utilization and requirements of a particular flow (such as desired rate). Some researchers suggest making this allocation in a centralized controller [1, 4] which can be infeasible to accomplish for every newly arrived flow.

Another problem associated with data center networks is congestion control. Recent studies [2, 10] indicate that TCP rate control, designed initially for the Internet, has poor performance in data center networks causing severe congestion situations. Therefore, more accurate rate control mechanisms are needed.

Thus, in this work we first compare several algorithms with and without global knowledge that are aware of flow demands and requirements. We report system performance as the average utility obtained by each flow. We do so to understand whether one can design a fully distributed flow allocation mechanism in such networks. We find that for many settings used in our simulations, algorithms with local knowledge perform not much worse than algorithms with global knowledge. Note, that we come to slightly different conclusions than authors of [1] because unlike that study we not only consider long lasting bulky flows, but also include deadline flows. We leverage our findings to propose a fully distributed mechanism that allows to allocate the flows and maintain fair network resource sharing. The mechanism has low complexity and requires little interaction with centralized controller.

The rest of the paper is organized as follows. In Section 2 we give background information and define the problem. In Section 3 we describe the simulation settings and our results. Further, in Section 4 we describe main aspects of the architecture for demand aware flow allocation and evaluate it using commodity hardware. We conclude the paper in Section 5.

2. PROBLEM SETTING

First, we ask a question of how should network performance of a particular application be measured? Undoubtedly, there is no single answer to this question which covers all possible applications. However, as defined in [8], the notion of network performance is best characterized in terms of utility, *i.e.* the amount of "happiness" of a particular user or satisfaction of application's needs.

In the referred paper the author formalizes utility functions as follows. Let the vector s_i describe the service delivered to ith user or application, such that s_i includes all relative measures, *e.g.* desired bandwidth and delay, of the delivered service. One then can define a function U_i that maps the service delivered into performance of the application or satisfaction of the user. Obviously, increase in U_i reflects improved performance.

The goal of the network design is then to find such allocations of resources that will maximize the sum of all utilities, *i.e.* $V = \max(\sum_{\forall i} U_i(s_i))$. The quantity V, as defined in [8], is the total *efficiency* of the architecture. For large scale and dynamic networks the computation of allocations which guarantee optimal V is challenging and perhaps infeasible given the high interarrival rate of new flows in the system. Instead, in this work we consider a slightly different problem: can we design a fully distributed flow allocation algorithm which is almost as good as algorithm with global information available?

2.1 Utility functions

We do not aim to cover all related classes of traffic in this paper, but only two that correspond to *elastic applications* and *deadline applications*. Indeed, these two classes of traffic are prevailing in the networks such as modern data centers: elastic applications are represented with the backup services (or in general any bulk data transfer) and deadline applications can be characterized by scatter-gather type of applications, *e.g.* MapReduce. In this work, the only metric that contributes to utility is bandwidth, *i.e.* s_i is bandwidth perceived by the flow.

DEFINITION 1. Utility function *is a mapping* $U : [0, C] \rightarrow R^+$*, where* C *is total amount of resources available to a particular network element (router, switch, network card, link, etc.), s.t.* $\forall i\ 0 \leq U(s_i) \leq u^{max}$ *and* $U(s_i) \leq U(s_i')$*, s.t.* $s_i < s_i'$*. In this work* $u^{max} = 1$.

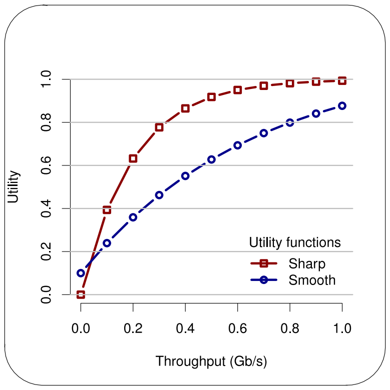


Figure 1: Elastic flow utility function

In this study we consider two distinct classes of utility functions: for elastic and deadline flows. For elastic flows we consider a strictly concave monotonically increasing function [6, 8] $U(s_i) = \alpha - exp(-s_i \cdot C_{shape})$, s.t. $0 < C_{shape} < 1$ and $\alpha \in R$. For the purpose of our studies we define two subtypes: *Sharp* function ($\alpha = 1$ and $C_{shape} = 5/Capacity$) in which the utility that application gets is not very sensitive to perceived bandwidth when the rate is high, but drops very fast when the rate gets smaller than $1/2$ of link capacity; and *Smooth* function in which the utility drops almost proportional to the decrease in application sending rate ($\alpha = 1.1$ and $C_{shape} = 1.5/Capacity$). Note, that unlike *sharp* utility function, with *smooth* one every elastic flow (i) gets some utility even if its sending rate is zero, *i.e.* elastic flows always wait for the channel to become available and are satisfied by that, and (ii) never gets utility 1 even when entire capacity is allocated to the flow, *i.e.* elastic applications always remain greedy for bandwidth. The shapes of the two functions are shown in Figure 1.

To illustrate intuition behind the shapes of these two utility functions, we can consider flows with different saturation. For instance, flow sending rate may be limited by computational capabilities of the machine, leading to saturation of e.g. 0.5 *Gbps* on a 1 *Gbps* link. For such flows the bottleneck is not the network but rather own processing capabilities and increase in sending rate over certain limit is unlikely to produce much benefit, which is modeled with the *sharp* utility function. Flows that are able to fully saturate the link capacity are best modeled with *smooth* utility function (the bottleneck is always network), which is flatter and has more uniform increase of utility in respect to increase in bandwidth over the whole interval of possible bandwidths.

For deadline flows we consider a step function which in its simple form can be defined as

$$U(s_i) = \begin{cases} u^{max} & \text{if } s_i \text{ was allocated} \\ 0 & \text{otherwise} \end{cases}$$

Note, that the function defined above can be generalized to accommodate multiple rates, if the application supports this. In this work we define a *deadline flow* as a flow which must complete within a specified time. To fulfill this requirement, we assign a fixed bit rate to each deadline flow.

DEFINITION 2. Flow admission: *A flow* f_i *is said to be admitted into the network if its demand is satisfied (i.e. it receives non-zero utility) across the entire path from source to destination and its placement does not lead to blocking of a previously admitted flow* $j, \forall j \neq i$*. The flow* f_i *is said to be blocked otherwise.*

This definition implies that a deadline flow is admitted only if the network is able to provide the required bandwidth, while an elastic flow gets admitted if its perceived utility is greater than 0 (note this can be true even if allocated bandwidth is 0). If a flow f_i was admitted into the network it is guaranteed to complete, *i.e.* it never gets removed from the system before completion. Also, in our simulations we do not move already allocated flows from one port to another.

2.2 Algorithms

In this paper we consider the following demand aware algorithms for flow scheduling:

- *Random (RND), or Oblivious*: A simple algorithm that randomly chooses the outgoing port for a flow. Essentially, this simple algorithm is similar to ECMP flow scheduling (with a distinction that our algorithm differentiates between flow classes and is aware of flow demands).

- *Greedy with local knowledge (GLK)*: A greedy algorithm which seeks the best assignment of the flow to an outgoing port based on the available local knowledge of other flows served by the device. Essentially, with this algorithm, forwarding element chooses such allocation of a newly arrived flow that maximizes its *own local utility* and not utility of the entire network.

- *Greedy with global knowledge (GGK)*: A greedy flow allocation algorithm which has the global knowledge of the system.

This simulates the centralized controller capable of fully optimal flow allocation (note that this optimization problem is solvable in polynomial time since the allocated flows cannot be reallocated upon arrival of a new flow).

3. SIMULATION

In this section we describe the metrics we use to assess performance of the three algorithms (introduced in the previous section), describe the implementation of the simulator and the simulation parameters, and finally report the results and draw conclusions.

3.1 Metrics

To assess the performance of algorithms we choose two metrics. The first metric we report is the *average utility* for all flows (blocked and non-blocked). This metric is a good indicator of the average system performance. Formally, we define the metric as follows

$$U_{avg} = \frac{V}{N_{total}},$$

where N_{total} is the total amount of flows (blocked and non-blocked), and V is the sum of utilities of all flows currently present in the network. Although U_{avg} metric indirectly reflects the effect of blocked flows, *i.e.* blocked flows receive 0 utility and thus push the average utility towards 0, we report the number of blocked flows using *average blocking probability* metric. In a formal way it can be represented as follows

$$p = \frac{N_{blocked}}{N_{total}}$$

3.2 Parameters

We could have used off-the-shelf simulating tools such as *Omnet++* or *NS-3*, however it appeared not feasible to complete simulations within reasonable time frames using these general-purpose simulators, given our desired values of network size (> 2000 nodes), flow arrival and departure rates, and duration of the simulation. We therefore implemented a custom simulator comprising almost 4000 lines of code. Though even in our custom simulator simulating 1 minute operation of a data center with almost 2500 network elements and mean flow arrival rate of $<$ 1msec, took several days.

In our simulations we use fat tree topology and use terms 'edge' and 'pod' as defined in [7]. Following [1], all links in our simulations have capacity of 1Gbps. We based our parameters selection on available measurement studies of production data centers [2,3,5]. The main parameters used in our simulations are flow arrival rate, flow duration and size, flow type probability (elastic or deadline) and distribution of flow source-destination pair (within the edge flow, inter-pod flow, etc). We combined various combinations of these parameters into four simulation configurations. We use the same parameters for simulating all three algorithms.

Configuration 1. In this configuration the mean arrival rate was on the sub-millisecond scale. In particular, in this configuration we model the arrival rate with *Weibull distribution* with parameters $\lambda = 1200, k = 2$, which gives median inter-arrival time of roughly 1ms. We also used *Weibull distribution* to simulate the duration of elastic flows. In this case the parameters were $\lambda = 1 \cdot 10^6, k = 2$, which corresponds to median duration of 850ms. For deadline flows we were using uniform distribution to draw the amount of bits transmitted within the flow $B \sim \mathcal{U}[7.5 \cdot 10^5, 8 \cdot 10^5]$, *i.e.* the minimum number of bytes was around 91kB. Furthermore, we used *exponential distribution* with parameter $\lambda = 0.0005$ to draw the required completion time of a deadline flow. We made sure that no deadline flow lasts less than 1ms (which also guarantees than no deadline flow will require rate greater than link capacity, *i.e.* 1Gbps). In this configuration we also set the probability of flow being elastic to $p_{elastic} = 0.7$. By letting elastic flows prevail, we study how the algorithms are different in terms of delivered average utility. Finally, we set the probability that the source-destination pair was within the same edge $p_{edge} = 0.02$, the same pod $p_{intrapod} = 0.3$, and otherwise within different pods $p_{different\ pods} = 0.68$. In this configuration we model the performance of elastic flows with the *sharp* utility function.

Configuration 2. In this configuration we keep all parameters identical to those in *Configuration 1*, but only increase the flow arrival rate and model it with *Weibull distribution* with parameters $\lambda = 800, k = 2$, which decreases median inter-arrival time to about 0.665ms and duration of elastic flows $\lambda = 2 \cdot 10^6, k = 2$ (median 1.7s). This experiment lets us understand whether arrival rate influences behavior of the algorithms (in terms of delivered average utility). In this configuration we model the performance of elastic flows with the *sharp* utility function.

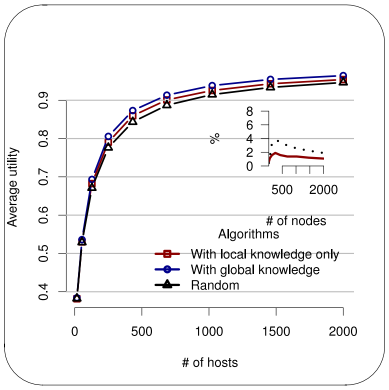
Configuration 3. In this configuration we further increase flow arrival rate by setting $\lambda = 600, k = 2$, which yields median inter-arrival time of 0.5ms. We also increased the duration of elastic flows which we modeled now with *Weibull distribution* with $\lambda = 5 \cdot 10^6, k = 0.5$. But most importantly, we now set the probability of flow being elastic to just $p_{elastic} = 0.25$. This allows us to understand how well can the algorithms reduce the overall blocking probability in network with prevailing deadline flows. In this configuration we model the performance of elastic flows with the *sharp* utility function.

Configuration 4. Our final configuration resembles *Configuration 2* but now we model the performance of elastic flows with the *smooth* utility function. This experiment allows to understand the performance of the algorithms when elastic flows have full saturation of capacity and therefore require flatter utility function.

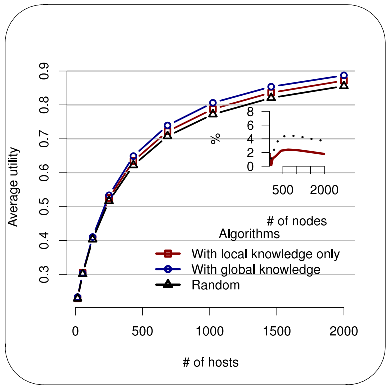
3.3 Results

First, we discuss the results of the configuration with prevailing elastic traffic (*Configuration 1*). We expect that this simulation will unveil potential (in terms of average utility) of the three algorithms that we study. All three show similar results (Figure 2(a)). For a network comprising switches with as much as 20 ports (2000 nodes) the algorithm with global knowledge performs just $\sim 2.5\%$ better than random flow scheduling algorithm and just 1% better than algorithm that relies on local knowledge only. And although the relative difference between the blocking probability for all three algorithms is big – algorithm with global knowledge has $> 100\%$ and $\sim 80\%$ smaller blocking probability comparing to random algorithm and algorithm with local knowledge only – the absolute difference is marginal. However, we think that the trend will remain for networks with higher loads giving all the advantage of global and even local information over the random forwarding strategy. To confirm this hypothesis, in Figure 2(b) we plot the average utility for *Configuration 2* in which number of flows was increased by almost 60% in comparison to *Configuration 1.* There is clear evidence that although in many settings algorithm with local knowledge is worse than algorithm with global knowledge it still performs better than random flow scheduling algorithm.

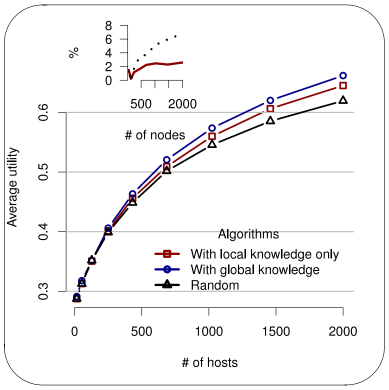
In our next experiment (*Configuration 3*) we let the deadline flows prevail over elastic flows. With this experiment we test how well each algorithm can avoid flow blocking. As expected, the blocking probability of the algorithm with global knowledge is lower than for the two other algorithms. Specifically, the blocking probability for the algorithm with global knowledge is $> 100\%$ and around 40% smaller than for random algorithm and algorithm with



(a) Configuration 1

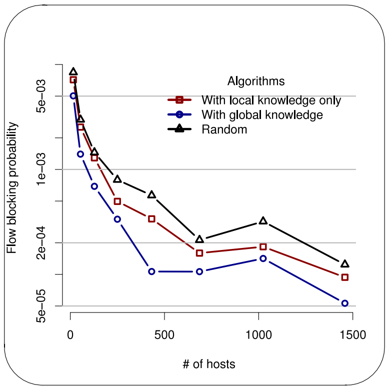


(b) Configuration 2

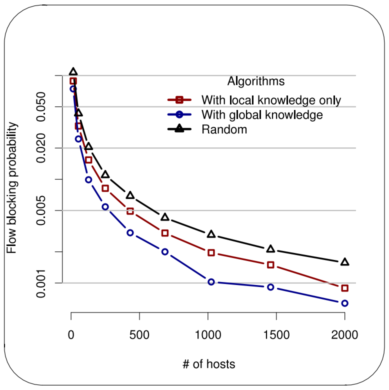


(c) Configuration 4

Figure 2: Average utility. Subplot shows improvement in percent of algorithm with global knowledge over random algorithm (dotted black curve) and over algorithm with local knowledge (solid red curve).



(a) Configuration 1



(b) Configuration 3

Figure 3: Flow blocking probability

local knowledge correspondingly. We demonstrate these differences in blocking probability in Figure 3(b).

Finally, we have measured the average utility for all algorithms using *smooth* utility function, *i.e.* the results for the *Configuration 4*. We present the results in Figure 2(c). There are two important observations that we can make directly from the plot. First, if the utility function is flatter, then the difference between *algorithm with local knowledge* and *oblivious or random strategy* becomes more prominent. Second, this difference tends to increase as the network size increases (this is expected as more paths are available and local information can aid in making better forwarding decisions). For instance, for the network with just 686 and 2000 hosts the difference between algorithms is 1.6% and 4.5% respectively, and should increase for larger network size ($\#hosts > 2000$).

Even though improvements of algorithm with local knowledge over random algorithm shown in Figure 2 may not seem to be big, we must note that exact values of average utility strongly depend on the choice of type and shape of utility function. It would possible for us to choose a utility function that would make the improvement more pronounced. We refrained from doing so, because we believe that since there is no reliable criterion of choosing the true utility function, it suffices to show relative performance of the three algorithms.

We must also point out that even if under the chosen utility function algorithm with local knowledge doesn't offer big improvement in average utility over random algorithm, it nevertheless significantly lowers flow blocking probability. This feature is very important for ensuring SLA in data centers.

4. DESIGN

Our simulation findings indicate that data center networks would benefit from a fully distributed mechanism that can fairly distribute resources. In this section we elaborate on the design principles of demand aware fixed (deadline) and limited rate (elastic) TCP variants and a switch forwarding logic. The first mechanism allows to allocate flows to proper outgoing ports and converge the rate of each flow to its max-min share in a fully distributed manner. The second mechanism (implemented on the end points) allows to achieve fair resource sharing among all the communicating peers without requiring additional support from the forwarding elements (*i.e.*, we do not require the forwarding elements to implement priority queues or similar mechanisms). And although the ideas we present here are similar in spirit to those presented in [9], the key difference is that we differentiate between flow types and take into account multiple paths available for the same destination.

4.1 Demand aware forwarding fabric

We begin with the description of the demand aware forwarding logic which is implemented at each switch in the network. There are two main design goals that we bear in mind. First, we want

Algorithm 1 Forwarding logic

```
function OnUpdateRoutingTable
    $FetchRoutingTableFromCentalController()$
end function
function ForwardFlow($flow_j$)
    if $Is\ new\ flow_j$ then
        if $flow_j.Rate > 0$ then
            if $AllocateFlow(flow_j) == success$ then
                $RecalculateRates()$
            else
                $flow_j.Rate = 0$
                if $flow_j\ is\ elastic$ then
                    $Send\ to\ destination\ using\ port\ with$
                    $least\ number\ of\ elastic\ flows$
                else
                    $The\ flow\ was\ blocked$
                    $Send\ to\ destination\ using\ any\ port$
                end if
            end if
        else
            if $flow_j\ is\ elastic$ then
                $Send\ to\ destination\ using\ port\ with$
                $least\ number\ of\ elastic\ flows$
            else
                $The\ flow\ was\ blocked$
                $Send\ to\ destination\ using\ any\ port$
            end if
        end if
    else if $IsFlagSet(flow_j.RateChanged)$ then
        if $flow_j.Type == elastic$ then
            $RecalculateRates()$
        end if
    end if
    if $AFlowComplete()$ then
        $RecalculateRates()$
    end if
end function
function RecalculateRates
    $SortDescendingByRate(ElasticFlows)$
    $ElasticFlowsCount \leftarrow Length(ElasticFlows)$
    $AvailableCapacity \leftarrow Capacity - \sum_{\forall i} DeadlineFlow_i.Rate$
    $FairShare \leftarrow \frac{AvailableCapacity}{ElasticFlowsCount}$
    for all $flow_j \in ElasticFlows$ do
        if $flow_j.Rate < FairShare$ then
            $AvailableCapacity \leftarrow AvailableCapacity + (FairShare - flow_j.Rate)$
            $ElasticFlowsCount \leftarrow ElasticFlowsCount - 1$
            $FairShare \leftarrow \frac{AvailableCapacity}{ElasticFlowsCount}$
        else
            $flow_j.Rate \leftarrow FairShare$
            $SetFlag(flow_j.RateChanged)$
        end if
    end for
end function
```

the system to be full distributed. In particular, such functionality as flow allocation to an outgoing port and update of the maximum possible sending rate for each flow are local responsibility of each forwarding element. Second, we try to minimize the amount of per-packet processing tasks performed by each forwarding element, to avoid our mechanisms becoming a bottleneck. In our architecture a centralized controller only computes the directed acyclic graph (*DAG*) on longer time intervals (*e.g.*, on order of several minutes) and populates the routing table of each switch in the network.

Formally, the forwarding logic can be described with the pseudo-code of Algorithm 1. Initially, whenever a new flow arrives at the forwarder, the flow's desired rate which we assume to be a field in the packet header, is checked. If it is zero and flow is deadline then the flow was blocked on one of the previous hops. In such case, nothing has to be done and the packet is forwarded towards the receiver using any port through which the destination is reachable. If the flow was elastic it is sent to a port with least number of elastic flows to avoid future congestion. Otherwise, the forwarder attempts to allocate the flow to a port which maximizes the local utility of the switch. This is achieved with $AllocateFlow()$. If the allocation is successful all elastic flows are assigned a new sending rate, which is the fair share of link capacity excluding the amount of bandwidth allocated to hard deadline flows. Note that if the rate of the flow on a prior link is less than a new fair share, the rate of the flow remains unchanged. Otherwise, the flow gets assigned with the new fair share and the packets are marked with a bit flag indicating that any upstream forwarder should recalculate the rates for its flows as well.

In practice, $AllocateFlow(flow_j)$ can be either *random* algorithm or *greedy algorithm with local knowledge only*. However, as our simulation results suggest, algorithm with local knowledge is better and therefore preferable flow allocation mechanism. Moreover, as we have demonstrated, the improvement of algorithm with local knowledge over random strategy tends to increase as the network utilization increases.

4.2 Rate control mechanisms

In this section we sketch the design of the two modifications to TCP (which we denote as TCP*) that allow the nodes to use the sending rates assigned to the flows by the forwarding elements (assuming nodes in data centers are not selfish). But before we present these changes, we give several useful and more formal definitions.

DEFINITION 3. R_i Fixed rate TCP*: *A TCP* flow is said to have fixed rate if the hosts transmit packets with constant rate R_i which is not changed during the connection lifetime.*

DEFINITION 4. R_i Limited TCP*: *A TCP* flow is said to be limited if its sending rate cannot surpass requested rate R_i.*

We present the modifications in the Algorithm 2. Note that we assume that such parameters as flow type and requested rate are passed from higher layer applications through socket options, *e.g.*:

```
setsockopt(socket, SOL_SOCKET, SO_FLOW_TYPE, type)
setsockopt(socket, SOL_SOCKET, SO_FLOW_RATE, rate)
```

Similarly to normal TCP a handshake is performed initially. This is where the forwarders assign the maximum allowed sending rate for the flow (this information is conveyed back from the receiver inside a TCP option). If the rate is greater than 0 and flow is of deadline type, the flow was admitted and can start its normal operation. Otherwise, the connection must be closed. Elastic flows can tolerate 0 sending rates and wait until rate is reallocated. The update procedure is similar.

The key idea behind this simple changes, is to not allow the flows to exceed their allocated sending rate. For the elastic flows (limited TCP*) this is achieved by not increasing the congestion window if the number of bytes sent was larger than the number of bytes allowed to be sent during some unit of time. For deadline flows (fixed rate TCP') the hosts simply must queue the packet if the number of bytes sent within some time unit surpasses the maximum allowed. These two simple modifications allow to maintain the network-wide fair resource share in a distributed way.

Algorithm 2 Sending rate adaptation mechanisms

```
$OpenConnection(flow_i)$
if $flow_i.Rate == 0$ and $flow_i.Type == deadline$ then
    $CloseConnection(flow_i)$
end if
while $IsOpenConnection(flow_j)$ do
    if $flow_i.Type == elastic$ then
        if $Need\ to\ increase\ TCP\ CWND$ then
            if $Bytes\ sent\ within\ T < \frac{flow_i.Rate}{T}$ then
                $CWND{+}{+}$
            end if
        end if
    else
        if $Bytes\ sent\ within\ T \geq \frac{flow_i.Rate}{T}$ then
            $Queue\ packet$
        else
            $Send\ Packet$
        end if
    end if
end while
```

4.3 Evaluation

In reality, given the high network loads and as a result the number of flows each forwarder must handle per unit time, flow allocation must be efficiently computed. Therefore, we have evaluated the performance (duration in microseconds) of Algorithm 1 (for a single switch) on an old 700MHz computer. We present the results in Figure 4. Note that even when there are 100 flows per port which sums up to 4800 flows per 48-ports switch (a realistic load) it takes on average $123\mu sec$ and $14\mu sec$ to allocate a new flow or update existing flows correspondingly. Assuming that there are at most 6 hops from any source to any destination, $700\mu s$ is the maximum time required to allocate the flow throughout the entire path. This is comparable with the typical deadline flow duration (1.5ms) in our simulations. For this reason we suggest that flow allocation time should be included in SLAs.

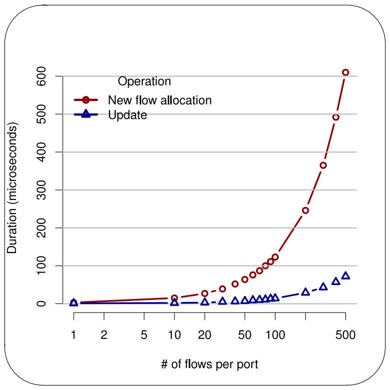


Figure 4: Computational overhead for allocating new or updating existing flows on 48-port switch

5. CONCLUSIONS

In this paper we considered the problem of demand aware flow allocation in a data center network. Although there were several studies that consider similar problem, we unlike other researchers differentiate between several classes of traffic and measure the goodness of flow allocation using the perceived utilities. We find that although there is difference between algorithms with and without global knowledge, it is rather insignificant for all simulation settings we try. This together with the fact that algorithm with local knowledge performs better than naive random algorithm, enables us to propose demand aware algorithms for per-hop load balancing and several modifications to transport protocol that undoubtedly can improve the network performance. For the future, we plan to extend this to other topologies and introduce more complex utility functions taking into account additional indicators of link congestion such as packet loss.

Acknowledgments

We would like to thank Scott Shenker, Dmitry Korzun, Barath Raghavan and Ilya Nikolaevskiy for their feedback. This work was supported in part by Academy of Finland project SEMOHealth.

6. REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 63–74, 2010.

[3] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[4] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12, New York, NY, USA, 2011. ACM.

[5] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 202–208, 2009.

[6] W.-H. Kuo and W. Liao. Utility-based resource allocation in wireless networks. *IEEE Transactions on Wireless Communications*, 6:3600–3606, 2007.

[7] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 39–50, 2009.

[8] S. Shenker. Fundamental Design Issues for the Future Internet. *IEEE journal on selected areas in communications*, 13:1176–1188, 1995.

[9] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 50–61, New York, NY, USA, 2011. ACM.

[10] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data center networks. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 13:1–13:12. ACM, 2010.