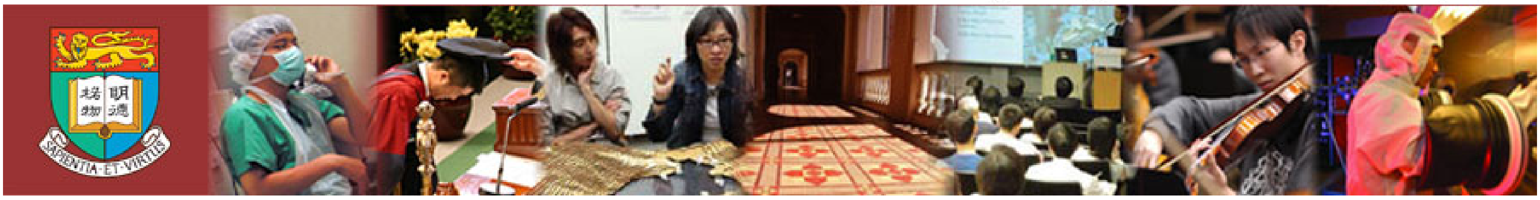


Responsive Multipath TCP in SDN-based Datacenters

Jingpu Duan[*], Zhi Wang[†], Chuan Wu[*]
[*]Department of Computer Science, The University of Hong Kong, Email: {jpduan,cwu}@cs.hku.hk
[†]Graduate School at Shenzhen, Tsinghua University, Email: wangzhi@sz.tsinghua.edu.cn

***Abstract*—A basic need in datacenter networks is to provide high throughput for large flows such as the massive shuffle traffic flows in a MapReduce application. Multipath TCP (MPTCP) has been investigated as an effective approach toward this goal, by spreading one TCP flow onto multiple paths. However, the current MPTCP implementation has two major limitations: (1) a fixed number of subflows are used without reacting to the actual traffic condition; (2) the routing of subflows of a multipath TCP connection relies heavily on the ECMP-based random hashing. The former may lead to a waste of both the server and network resources, while the latter can cause throughput degradation when multiple subflows collide on the same path. This paper proposes a responsive MPTCP system to resolve the two limitations simultaneously. Our system employs a centralized controller for intelligent subflow route calculation and a monitor running on each server for actively adjusting the number of subflows. Working in synergy, the two modules enable MPTCP flows to respond to the traffic conditions and pursue high throughput on the fly, at very low computation and messaging overhead. NS3-based experiments show that our system achieves satisfactory throughput with less resource overhead, or better throughput at similar amounts of overhead, as compared to common alternatives.**

I. Introduction

Datacenters are extensively employed to carry out “heavy” tasks such as masssive data processing, storage and distribution. Many large flows exist in a datacenter network, for content replication, virtual machine migration, and data shuffling in MapReduce-like computational workloads, which constitute the majority of datacenter traffic [1]. Efficient transfer of these large flows is crucial to the performance of a datacenter network. To improve the throughput of large flows, significant efforts have been devoted to flow routing and transportation designs.

From the routing aspect, datacenter networks are typically built on a fat-tree topology [2], providing redundant paths between different hosts. Flows are usually routed using ECMP [4], which spreads the flows onto different paths by hashing flow data in the packet headers. However, existing studies have pointed out that multiple flows may collide on the same path using ECMP hashing, causing significant throughput degradation [3] [4]. To alleviate such throughput degradation, more active flow path scheduling should be undertaken, which responds to the traffic conditions. Based on Software Defined Network (SDN) [5], Hedera [3] employs a centralized controller to monitor the network condition and periodically replace congested flow paths with less congested ones for data transfer. However, replacing the flow path for established flows will cause potential packet reordering and loss. In contrast, our work tries to establish new subflows for established flows, which totally avoids packet reordering and loss. We also design our own route calculation strategy that fits our subflow adding scheme better.

From the transportation aspect, TCP is not efficient in datacenter networks, and different variants of TCP-like protocols have been proposed to improve large flow transportation. DCTCP [6] and D2TCP [7] are two representatives. DCTCP provides a guaranteed RTT by actively controlling queue lengths in switches, and D2TCP guarantees that flow transfers meet their deadlines. The limitation is that both of them cannot provide a near-optimal throughput for large flows, since they adopt a single transport path per flow and fail to grab extra throughput by exploiting path diversity in a datacenter network.

To achieve higher throughput, a large flow can be split and transmitted using multiple paths. Sen et al. [8] proposes to dynamically split flows at the switches. However, splitting flows at a switch requires the switch to be able to match the TCP sequential number, which is an advanced new feature and not widely deployed. MPTCP [4] [9] splits a flow into multiple subflows at the end host, each to be sent on a different path. Though theoretically better to use more subflows [10], maintaining more subflows incurs more overhead. With MPTCP, a flow’s packets are scattered onto multiple paths, and hence the receiver needs to maintain a large buffer to assemble the out-of-order packets, causing increased memory footprint. In addition, each subflow is maintained as a sub-socket in the kernel and additional system calls need to be made to manipulate the sub-sockets, consuming CPU capacity [11]. The current Linux kernel implementation of MPTCP hence limits the maximum number of available subflows per MPTCP session to 8 [12]. What’s more, in a software defined network, adding subflows leads to installing more forwarding rules at the switches, consuming the precious TCAM resource. It is therefore crucial to achieve the maximal MPTCP throughput using as few subflows as possbile. Most existing MPTCP proposals employ a fixed number of subflows, and route the subflows using ECMP [4], which cannot achieve high flow throughput as well as the minimal resource consumption at the switches and the end hosts. A better approach is in need.

This paper proposes a responsive MPTCP system for SDN-based datacenters, which decides the number of subflows for each MPTCP session and the best subflow paths in responding to the current traffic condition. The system consists of two main modules: (1) a *controller* that carries out light-weighted algorithms to compute the additional number of subflows to

This work was supported in part by a grant from RGC under the contract HKU 717812 and NSFC under the grant No. 61402247.

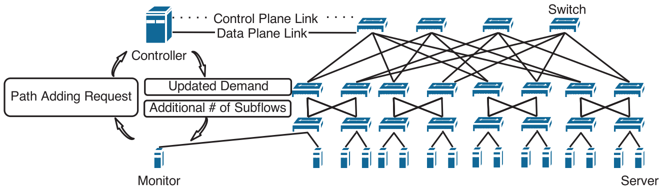


Fig. 1: An Overview of the Responsive MPTCP System

add and the best subflow paths, and (2) a *monitor* on each server for actively adjusting the number of subflows. The two modules work in concert through light-weighted message passing, enabling MPTCP flows to pursue high throughput on the fly, at very low computation, memory and bandwidth overhead. We carry out extensive NS3-based experiments and the results show that our system achieves satisfactory throughput with less resource overhead, or better throughput using the same amount of resource, as compared to common alternatives.

II. System Architecture

We consider a datacenter network built on the k-Ary fat-tree topology with three layers of k-port switches interconnected using homogeneous links to provide non-blocking switching ability for $\frac{k^3}{4}$ servers [2]. An SDN controller is connected to the switches as well as the data plane of the datacenter network, and is able to communicate with each server. Each server is configured to use MPTCP when transmitting large flows (*e.g.*, flow with a total data size larger than 256KB), and runs an active monitor module, which is responsible for communicating with the controller.

In the data plane, the controller constantly estimates the maximally achievable throughput for each MPTCP flow (referred to as *demand* of the flow) and delivers the updated demand of the existing flows to the monitors of the respective source servers. The monitor records the flow's throughput and issues a path adding request when it detects a significant gap between the achieved throughput and its achievable demand. When the controller receives a path adding request from a monitor, it notifies the monitor the number of subflows to add, in order to achieve the demand.

In the control plane, a switch notifies the controller about new flow arrivals and expiration through OpenFlow messages. Upon receiving such a message, the controller updates flow entries in demand table and configures the flow paths for new subflows by installing forwarding rules in the involved switches. The overall architecture of the proposed system is presented in Fig. 1. The detailed design of the controller and the monitor will be presented in what follows.

III. Controller

Fig. 2 illustrates interfaces and function modules of the controller. Upon receiving the path adding request from a monitor, the controller runs its route calculation algorithm, to compute the additional flow paths the respective MPTCP session can add, saves the computed paths, and sends the number of additional subflows back to the monitor. For communication with

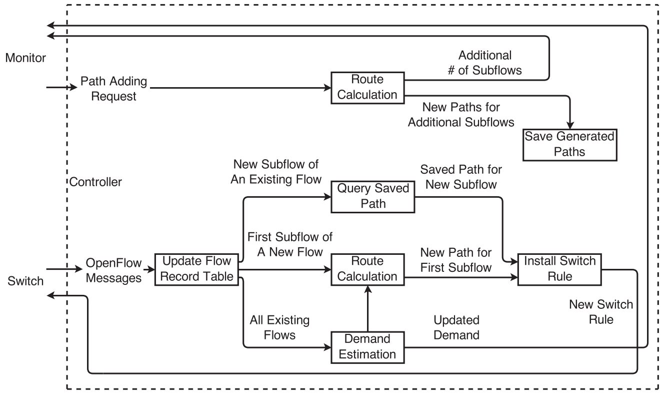


Fig. 2: Controller Architecture

the switches, two types of Openflow messages are of particular interest to the controller, the Packet-In messages and Flow-Removed messages. A Packet-In message is generated when the first packet of a new (sub)flow arrives at a switch, which notifies the controller about the arrival of a new (sub)flow. A Flow-Removed message is sent by a switch to notify the controller about the departure of an existing flow, when the respective flow rules expire in the switch. The controller maintains a demand table that records all the existing flows with their demand. Upon receiving a Packet-In or a Flow-Removed message, the controller updates the corresponding flow entries in the demand table, and proceeds following one of the two cases. (1) The arrival of a new subflow of an existing MPTCP flow: the controller finds one saved path from its previous route computation (which led to subflow addition notification to the monitor and hence the arrival of this new subflow), and then installs routing rules on the switches along the path. (2) The arrival of the first subflow of a new MPTCP flow: the controller runs its route calculation algorithm to identify the path for this new subflow, and then dispatches the rules to the involved switches. Upon (sub)flow arrival or departure, the controller also runs its demand estimation algorithm to update the maximally achievable throughput target (*i.e.*, the demand) for all the existing flows, and dispatches updated demand to the respective monitors whenever the demand change is significant.

We next present the two algorithmic modules of the controller, demand estimation and route calculated, in details.

A. Demand Estimation

Fat-tree datacenter networks with uniform link bandwidth are fully non-blocking networks [10][3]. In such a network, the bandwidth bottleneck along a flow's path can only happen at the sender's access link or the receiver's access link, and a flow's demand is the throughput achieved in TCP equilibrium state. We employ the same algorithm as in Hedera [3] to estimate a flow's demand in a non-blocking network, which proportionally increases flows' transmission rates at the senders' access links and proportionally decreases excessive rates at the receivers' access links until all the flow rates stablize. The rationale is that the TCP congestion control

Algorithm 1 Demand Estimation and Dispatching

```
Input: demand record $d\_t$, new flow $f\_n$ or expired flow $f\_e$
save $d\_t$ as an old demand record $old\_d\_t$, $old\_d\_t \leftarrow d\_t$ ;
update $d\_t$ to include $f\_n$ or exclude $f\_e$;
run Hedera demand estimation algorithm with $d\_t$ as input;
for each flow $f$ in $d\_t$ do
    if $f$ is not in $old\_d\_t$ then
        dispatch $d\_t[f]$ to the monitor at sender of $f$;
    else
        if $| d\_t[f] - old\_d\_t[f] | > \delta_{DDT} * old\_d\_t[f]$ then
            dispatch $d\_t[f]$ to the monitor at sender of $f$;
        end if
    end if
end for
```

process strives for max-min fairness among flows sharing the same link when the network enters an equilibrium state.

Our complete demand estimation algorithm is summarized in Alg. 1. The controller maintains a demand table d_t that records all the existing flows with their current demand. After receiving a Packet-In message or a Flow-Removed message, the Hedera demand estimation algorithm is invoked to reestimate the demand for all the existing flows. If the change of a flow f's demand exceeds a factor δ_{DDT} times the old demand or if f is a new flow, the updated demand for this flow will be dispatched to the flow sender's monitor.

B. Route Calculation

The route calculation algorithm is invoked in two cases: (i) when a monitor detects that the throughput of one flow is significantly lower than its current demand received from the controller and issues a path adding request to notify the controller about the demand gap (*i.e.*, demand minus current throughput), and (ii) when a switch notifies the arrival of a new MPTCP flow, in which case the new flow's demand is estimated through the demand estimation module and used as the demand gap. Using the demand gap as input, the route calculation algorithm computes a set of new paths for the flow, whose aggregate throughput can cover as much the demand gap as possible, and notifies the monitor to add additional subflows on these paths. We next propose an approach for estimating the expected throughput for a new subflow on a given path, which serves as the base for the route calculation algorithm. This method relies on the following two observations.

(1) Aggregate MPTCP Flows Fairly Share Link Bandwidth: When multiple subflows of a MPTCP flow are traversing the same link, they can be viewed as one aggregate flow. The MPTCP congestion control enables fair sharing among aggregate MPTCP flows. So when n MPTCP flows are contending for a link with bandwidth B, the expected throughput of each MPTCP flow can be calculated as B/n, which is an upper bound to the expected throughput of individual subflows in the MPTCP flow.

(2) Subflows at most Share Links in the Aggregation-ToR Layer but not in the Core-Aggregation Layer: Due to the fairness constraint, it would be meaningless if multiple subflows of the same MPTCP flow take the same paths. In a fat-tree network as shown in Fig. 3, subflows of a intra-pod flow should never share links on their paths (except access links), and subflows of an inter-pod MPTCP flow may share links in the aggregation-ToR layer, as long as they use different links in the core-aggregation layer. If multiple subflows of an inter-pod flow share the same aggregation-ToR uplink, then they must also share the same aggregation-ToR downlink, and vice-versa.

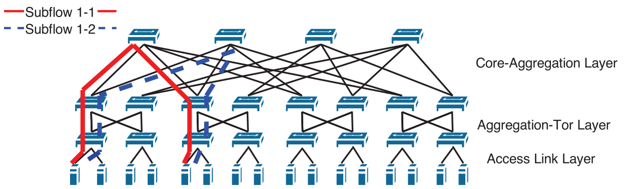


Fig. 3: Two Subflows Share Links in Aggregation-ToR Layer

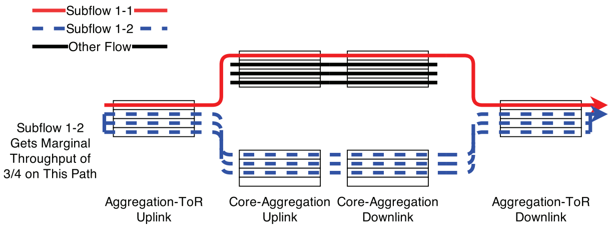


Fig. 4: An Illustration of Marginal Throughput

Algorithm 2 Route Calculation

```
Input: flow $f$, demand gap $d\_g$, number of existing subflows $n\_sf$
set number of additional subflows $n\_new\_sf = 0$;
while $n\_new\_sf + n\_sf < M$ do
    find out a subflow path $p$ with the largest expected throughput $e\_t$;
    if $f$ is a new flow then
        install switch rules on $p$;
        return;
    end if
    if $e\_t == 0$ then
        return;
    end if
    save path $p$;
    $n\_new\_sf += 1$;
    if $e\_t > d\_g$ then
        notify the sender monitor of $f$ of $n\_new\_sf$;
        return;
    else
        $d\_g = d\_g - e\_t$;
    end if
end while
notify the sender monitor of $f$ of $n\_new\_sf$;
```

Based on Observation (1), we can calculate an upper bound of a subflow sf's throughput on path p, by finding the link along p shared by the largest number of aggregate MPTCP flows, n_{max}, and obtaining the upper bound as B/n_{max}. If there is no other subflow of the same MPTCP flow f on p, this B/n_{max} is exactly the expected throughput of sf.

If there are other subflows sharing one or more links on p, we seek to compute a *marginal throughput* of p, which quantifies the achievable throughput of subflow sf on p. Suppose that there are m existing subflows of MPTCP flow f that use the same aggregation-ToR uplink as on p. According to Observation (2), the m subflows constitute an aggregate flow f_{aggre} on both p's aggregation-ToR uplink and aggregation-ToR downlink. Suppose that there are n_u aggregate MPTCP flows on p's aggregation-ToR uplink and n_d aggregate MPTCP flows on p's aggregation-ToR downlink. Then the expected throughput of aggregation flow f_{aggre} is at most $\min(B/n_u, B/n_d)$. Also suppose that u_i is the upper bound of the expected throughput of flow f's ith existing subflow that uses p's aggregation-ToR uplink, calculated based on Observation (1). Then the marginal throughput on path p is computed as $[\min(B/n_u, B/n_d) - \sum_{i=1}^{m} u_i]^+$. Note that this marginal throughput represents an estimation of achievable throughput on path p (instead of accurate computation), which nevertheless is shown to perform well in our experiments.

Fig. 4 gives an illustration of the marginal throughput on a path. Each box represents a link with the same unit bandwidth. Subflow 1-1 is an existing subflow of flow 1, and subflow 1-2 is a new subflow of flow 1. Subflow 1-1 shares two links in the core-aggregation layer with three other aggregate MPTCP flows. The upper bound of the expected throughput of subflow 1-1 is $1/4$. Subflow 1-2 shares the same aggregation-ToR uplink and aggregation-ToR downlink with subflow 1-1. The expected throughput of their aggregate flow on the two aggregation-ToR links is 1. So the marginal throughput on subflow 1-2's path is $3/4$.

We apply the following two rules to calculate the expected throughput of a new subflow sf of flow f on a given path p.

Rule 1: If there is no other subflow of f that uses p's aggregation-ToR uplink, find out the largest number of aggregate flows sharing the same link on p, n_{max}. The expected throughput of sf on p is B/n_{max}.

Rule 2: If there are m existing subflows of f that use p's aggregation-ToR uplink, calculate p's marginal throughput m_t, as well as the upper bound of sf's expected throughput B/n_{max}. The expected throughput of sf on p is $\min(m\_t, B/n_{max})$.

Employing these two rules, our route calculation algorithm is given in Alg. 2. The algorithm generates n_new_sf additional subflows for input flow MPTCP f in order to fill up its demand gap as much as possible, by exploring new subflow paths as long as the total number of subflows does not exceed a cap M. When the algorithm exits, it notifies flow f's sender monitor to add n_new_sf more subflows. All the generated paths are saved. When the controller receives a Packet-In message for a new subflow of f, the controller picks up one of the paths and installs corresponding rules on switches along this path.

IV. Monitor

The monitor on each server executes three main functionalities: (1) keeping track of the achieved throughput for each existing flow sent out from the server, (2) deciding whether to issue a path adding request for a flow upon receiving updated demand of the flow, and (3) instructing flow f to add additional subflows after receiving a message containing the additional number of subflows from the controller.

Algorithm 3 Monitor Loop

Input: monitored flows $f[n]$, previous flow rates $p\_r[n]$, counters $count[n]$, flow demand $demand[n]$

```
for $i = 1 : n$ do
    obtain instant throughput $i\_r$ for flow $f[i]$;
    $p\_r[i] = 0.2 * p\_r[i] + 0.8 * i\_r$;
    if $| i\_r - p\_r[i] | < \delta_{RVT} * p\_r[i]$ then
        $count[i] = count[i] + 1$;
        if $count[i] == R$ then
            $count[i] = 0$;
            if $p\_r[i] < (1 - \delta_{DGT}) * demand[i]$ then
                $gap = demand[i] - p\_r[i]$;
                issue path adding request for $f[i]$ with $gap$;
                return;
            end if
        else
            return;
        end if
    else
        $count[i] = 0$;
        return;
    end if
end for
```

(1) and (2) are achieved through the monitor loop, periodically executed every *monitoring interval* t_m. Its detailed algorithm is summarized in Alg. 3. Here n is the number of MPTCP flows being monitored. The monitor samples the instant throughput i_r of each flow $f[i]$ in each monitor loop and maintains a weighted average historical throughput $p\_r[i]$ for each flow, with a higher weight for new rate samples. The difference between the instant throughput and the historical rate is compared against $\delta_{RVT} * p\_r[i]$, where δ_{RVT} is a rate-varying threshold to evaluate whether the flow rate changes drastically. If the instant rate i_r does not exhibit any significant change after R (R is set to 5 in our simulation) consecutive monitor loops, the average historical throughput $p\_r[i]$ is examined to determine if the demand gap is large enough for issuing a path adding request. We employ a *demand gap threshold* δ_{DGT}: if $p\_r[i]$ is smaller than $(1-\delta_{DGT})$ times the demand $demand[i]$ received from the controller, a path adding request with demand gap $demand[i] - p\_r[i]$ is sent to the controller.

V. Performance Evaluation

A. Experiment Setup

We evaluate our responsive MPTCP system using a NS3 simulator, where we implement our customized SDN controller and the monitor. According to the datacenter network configuration in [2], we simulate a datacenter network with a 8-Ary fat-tree topology which contains 128 servers connected using $1Gbps$ links. This topology is large enough to carry out our simulation because it contains 16 different paths for each intra-pod flow. The link delay is $20us$, $30us$ and

TABLE I: Summary of Experiment Results

	Random Traffic		Permutation Traffic		Shuffling Traffic	
	NS	TP(Mbps)	NS	TP(Mbps)	JCT(ms)	NS
$t_m = 2ms, \delta_{DGT} = 0.15$	3.72	521	3.88	766	1402	1.92
$t_m = 4ms, \delta_{DGT} = 0.25$	3.40	513	3.82	767	1441	1.28
$t_m = 8ms, \delta_{DGT} = 0.35$	2.47	506	3.75	760	1454	1.03
4 subflows+ECMP	4	530	4	734	1332	4
2 subflows+ECMP	2	493	2	643	1394	2
1 subflow+Hedera	1	438	1	747	1652	1

NS: Average number of subflows per flow.
TP: Average throughput per flow.
JCT: Average shuffle completion time per MapReduce job.

$40us$ for links in the access link layer, aggregation-ToR layer and aggregation-core layer, respectively. XMP [9] is used as the default MPTCP congestion control algorithm, which can provide a stable transmission rate and maintain lower queue occupancy. We configure each server to have two different IP addresses such that it allows up to 4 subflows per MPTCP connection. The maximum subflow number $M = 4$ is set due to the following: In an 8-Ary fat-tree topology, there are only four shortest paths between two servers in different pods that do not share links from aggregation-ToR layer; adding more than 4 subflows does not help much in increasing the MPTCP flow's throughput. We set $\delta_{DDT} = 0.15$ and $\delta_{RVT} = 0.15$ throughout the simulations because these two parameters do not directly determine whether the monitor issues path adding requests and have only a minor influence on the performance. For comparison, we also implement ECMP routing and Hedera flow scheduling to work with MPTCP, instead of our algorithms.

All the messages exchanged between the controller and a server (see Fig. 1) are encapsulated into 64-$byte$ packets, which only impose negligible communication overhead. Suppose that a large flow issues 10 path adding requests per second during its life time and large flows arrive and depart at a $100ms$ interval. Then an average of 10 large flows per server generate $150Kbps$ communication traffic at the server, less than 0.1% of the server bandwidth.

Metrics. In our experiments, we use the following metrics to benchmark the performance of different approaches: (1) the throughput of the MPTCP flow, (2) the number of subflows used by each MPTCP flow, and (3) the shuffle completion time in experiments with MapReduce shuffling traffic.

B. Experiment Results

Our experiments are carried out under three types of typical datacenter traffic: (1) random traffic, (2) permutation traffic and (3) shuffling traffic in MapReduce. A summary of the experiment results is given in Table I, and the details follow.

1) Random Traffic: In this set of experiments, each server randomly picks a destination server and transmits a MPTCP flow whose size follows a uniform distribution within the range of $32MB$ to $400MB$, under the restriction that each server receives no more than 4 flows. When a server finishes one flow transmission, it immediately starts a new flow. The experiment stops when the number of transmitted flows reaches 500.

We investigate the impact of the monitoring interval (t_m) and the demand gap threshold (δ_{DGT}). Fig. 5(a) and the first column in Table I show that with the increase of the monitor loop interval and the demand gap threshold, the number of subflows used by the flows decreases. This is because when the monitor loop interval and the demand gap threshold increase, the frequency that a monitor issues path adding requests is decreased, leading to few subflows on average.

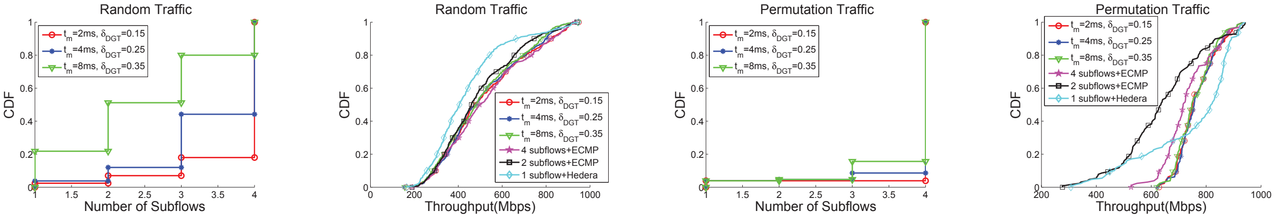
We also make comparison of our system with two existing approaches: MPTCP+ECMP routing, and MPTCP with single subflow+Hedera flow scheduling. Fig. 5(b) gives the CDFs of flow throughput achieved with our design under three different settings of t_m and δ_{DGT}, as well as those achieved with MPTCP+ECMP with 4 subflows per flow, MPTCP+ECMP with 2 subflows per flow, and MPTCP+Hedera with 1 subflow per flow. From this figure and the second column in Table I, we can make the following observations: (1) The 4-subflow ECMP scheme achieves the highest flow throughput, and the 1-subflow Hedera scheme achieves the lowest throughput; (2) Our design under all 3 settings achieves similar performance with the 4-subflow ECMP scheme, indicating that our responsive MPTCP system can achieve high throughput using simple algorithms and less resource consumption (reflected by the fewer number of subflows needed per flow from Fig. 5(a)).

2) Permutation Traffic: In this set of experiments, each server randomly selects a destination server and transmits only one flow, while each server also only serves as the destination of one flow (emulating the permutation traffic pattern). The flow size follows a uniform distribution within $64MB$ to $512MB$. All servers start flow transmission simultaneously. The experiments stops after the 128 flows from each server have been transmitted.

Fig. 5(c) and the third column in Table I show that in our system, most flows under the permutation traffic pattern use 4 subflows. The reason is that each flow has a demand (maximally achievable throughput) of $1Gbps$, and tends to use all four paths to achieve the demand.

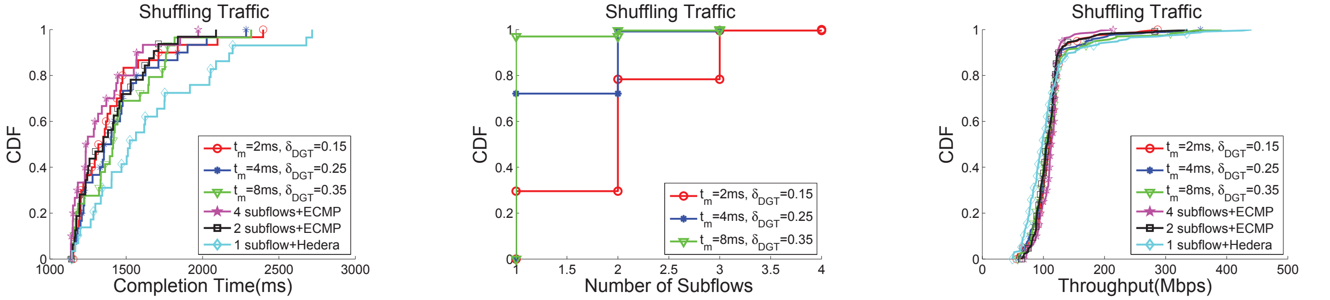
Fig. 5(d) and the fourth column in Table I demonstrate that our design with the setting $t_m = 4ms, \delta_{DGT} = 0.25$ achieves the best average throughput, among all different schemes. This result shows that when the number of subflows used by each flow in our system approaches the upper bound, the throughput of the flows benefits because of our better route selection mechanism.

3) Shuffling Traffic in MapReduce Jobs: In the next set of experiment, we simulate the shuffling traffic in the shuffling stage of MapReduce jobs, *i.e.*, the traffic incurred when mappers send intermediate data to each reducer. We choose 8 servers as the reducers. Each reducer reads from 8 different mapper servers concurrently by sending a read request ($2KB$) to them; then each of the 8 mappers transmits a data flow of $16MB$ to the reducer. We define the shuffle completion time as the duration from when the reducer sends the first read request to the time when it has completely received the last data flow from the mappers. The reducers repeat the above procedure until 30 shuffle jobs are completed. We also



(a) (b) (c) (d)

Fig. 5: (a) Number of subflows used by flows with random traffic. (b) Throughput of flows with random traffic. (c) Number of subflows used by flows with permutation traffic. (d) Throughput of flows with permutation traffic.



(a) (b) (c)

Fig. 6: (a) Completion time of shuffle jobs. (b) Number of subflows used by flows under shuffling traffic. (c) Throughput of flows under shuffling traffic.

schedule background random large flows to transmit during the execution of the shuffle jobs and the size of large flows follows a uniform distribution within the range of $32MB$ to $400MB$.

Fig. 6(a) and the fifth column in Table I illustrate that the average shuffle completion time with our design is slightly larger (5%) than that of the 4-subflow ECMP routing scheme. Nevertheless, our design utilizes a much smaller number of subflows (1-2 subflows for most flows), as illustrated in Fig. 6(b), leading to lower overhead. As compared to the 1-subflow Hedera scheme, our design under the setting $t_m = 2ms, \delta_{DGT} = 0.35$ employs an average number of subflows 1.03, similar to that in the Hedera scheme (see the last column in Table I), while achieving a 12% smaller average shuffle completion time. The reason to this good performance can be explained by Fig. 6(c) as the throughput of MPTCP flows achieved by our design is larger than that of the Hedera scheme and close to that of the 4-subflow ECMP scheme.

Results in this subsection indicate that our design can achieve comparably good performance using smaller numbers of subflows, or a better performance when a similar number of subflows are used.

VI. Conclusion

This paper presents a responsive MPTCP system. In our system, subflows are dynamically added to each MPTCP connection in responding to the actual traffic condition and are intelligently routed under the government of a centralized controller. NS3-based experiments show that compared with the existing approaches, our system achieves high throughput for large flows with less resource consumption, or even better throughput using similar amounts of resource.

References

[1] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proc. of ACM IMC*, 2010.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proc. of ACM SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Usenix NSDI*, 2010.

[4] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *Proc. of ACM SIGCOM*, 2011.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, 2008.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proc. of ACM SIGCOMM*, 2011.

[7] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware Datacenter TCP (D2TCP)," in *Proc. of ACM SIGCOMM*, 2012.

[8] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, "Scalable, Optimal Flow Routing in Datacenters Via Local Link Balancing," in *Proc. of ACM CoNEXT*, 2013.

[9] Y. Cao, M. Xu, X. Fu, and E. Dong, "Explicit Multipath Congestion Control for Data Center Networks," in *Proc. of ACM CoNEXT*, 2013.

[10] X. Yuan, W. Nienaber, Z. Duan, and R. Melhem, "Oblivious Routing for Fat-tree Based System Area Networks with Uncertain Traffic Demands," in *Proc. of ACM SIGMETRICS*, 2007.

[11] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, M. Handley *et al.*, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," in *Usenix NSDI*, 2012.

[12] *Multipath Linux Kernel Implementation*, http://www.multipath-tcp.org/.