Acceleration of Multipath TCP by Letting ACKs Take the Shortest Path

Jiangnan Cheng, Jaehyun Hwang, and Ao Tang
School of Electrical and Computer Engineering
Cornell University, Ithaca, NY
Email: {jc3377, jaehyun.hwang, atang}@cornell.edu

***Abstract*—Multipath TCP (MPTCP) can improve overall throughput of an end-to-end connection by leveraging different network paths. However, the heterogeneity of these paths can significantly hamper MPTCP's performance. In this paper, we propose to send acknowledgments (ACKs) along the lowest-latency path. This can help improve the performance of MPTCP when subflow throughput is constrained by packet loss by reacting to loss events faster. An active probing module is also developed to dynamically select the lowest-latency path against the potential change of path condition. Experiments demonstrate that overall throughput improvement generally ranges from 10% to 50%.**

I. Introduction

Single-path TCP (SPTCP) is a widely used transport layer protocol in today's Internet. However, it is often unable to provide sufficient throughput especially for long distance transfer. With this background, multipath TCP (MPTCP) [1] emerges, which can improve overall throughput of an end-to-end connection by leveraging multiple network paths [2].

It is well known that the heterogeneity (especially the RTT difference) of available paths can significantly hampers the performance of MPTCP. For example, when different paths have different RTTs, data packets delivered on different paths can be out of order, leading to receiver-side buffer blocking problem [3]–[7].

Interestingly, such latency heterogeneity can also bring opportunities. More specifically, we propose to send acknowledgments (ACKs) of all subflows of a MPTCP connection along the shortest path. It helps increase throughput by reducing RTTs of subflows that do not use the shortest path. Furthermore, an active probing module is also developed to select the shortest path when path condition changes due to link congestion or failure. Theoretically, this method can improve the overall throughput by as much as 100% for certain cases. Experiments demonstrate that overall throughput improvement generally ranges from 10% to 50% in practice.

II. Sending ACKs on the Shortest Path

Consider a sender-receiver pair connected by several heterogeneous paths with different latencies. In the standard MPTCP protocol, different subflows are built upon different paths. Hence ACKs of different subflows go through different paths with different latencies. We propose that MPTCP receiver uses the shortest path to deliver all the ACKs. Since ACKs are much smaller packets compared to normal data packets, our method will have only a small overhead for the lowest-latency path, as long as the number of ACK packets shifted to the lowest-latency path at one time is not too large. Besides, when an ACK is lost, follow-up ACKs can still acknowledge the data packets associated with the lost ACK. So occasional loss of ACKs is not a problem even if the lowest-latency path has a higher (but still reasonable) loss rate.

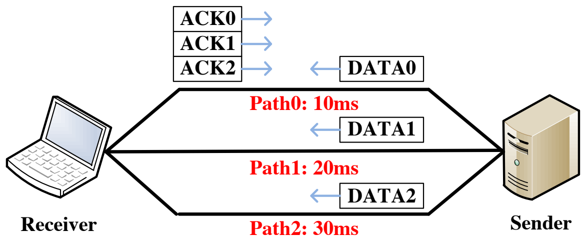

Fig. 1. ACKs on Lowest-Latency Path

Fig. 1 shows an example. The MPTCP connection has three available paths. A distinct subflow is built upon each available path. Same as the standard MPTCP, data packets of each subflow are sent on its associated path. But since path 0 has the lowest one-way latency, the ACKs of all the three subflows are delivered on the path 0 from the receiver to the sender.

The main benefit of the proposed method is that RTTs of subflows built on high-latency paths will decrease. For example, in Fig. 1, the RTTs of subflow 1 and 2 (which are built upon path 1 and 2) decrease from 40 and 60ms to 30 and 40ms, respectively.

III. Analysis

In this section we analyze the overall throughput improvement by sending ACKs on the shortest path.

Consider a sender-receiver pair connected by $N+1$ heterogeneous paths, which are indexed as $0, 1, \cdots, N$, respectively. Let d_i denote the one-way latency of path i, $\forall i$. Without loss of generality, let path 0 be the one with the smallest one-way latency, i.e.,

$$d_0 \leq d_i, \forall i \tag{1}$$

And subflow i is built upon path i, $\forall i$. Let p_i denote the loss rate of subflow i.

Now suppose that a large file is transferred from sender to receiver. Let T_i and T_i' be the average throughput of subflow i when using standard MPTCP protocol and ACKs on lowest-latency path method, respectively; and let T and T' be the average overall throughput when using standard MPTCP protocol and ACKs on lowest-latency path method, respectively. Let I denote the relative throughput improvement by adopting ACKs on lowest-latency path method, i.e.,

$$I = \frac{T'}{T} - 1 = \frac{\sum T_i'}{\sum T_i} - 1 \tag{2}$$

If we only consider the effect of RTT decrement for those subflows built on high-latency paths, theoretically, on average we should have $I \geq 0$. This is because, other conditions being equal, RTT decrement for some subflows is at least not harmful (and should be helpful in most cases).

However, how much we can improve also depends on path condition (including the value of RTT and loss rate) and the congestion control algorithm we are using. In the remainder of this section, the throughput improvements of TCP-Reno and MPTCP-Lia are analyzed in details, and some other congestion control algorithms are also discussed.

A. TCP-Reno

If we specify TCP-Reno as our MPTCP congestion control algorithm, it actually means that each subflow uses TCP-Reno as its subflow-level congestion control algorithm independently. So the throughput improvement of each subflow is also independent.

Consider a TCP-Reno connection whose RTT is rtt and loss rate is p. We increase window size by $1/w$ packet when a data packet is correctly delivered (where w is the current window size), and we decrease window size by half when a data packet is lost. If there is no timeout event, the long-time average window size $\bar{w}$ is determined by

$$(1-p) \cdot \frac{1}{\bar{w}} = p \cdot \frac{\bar{w}}{2} \tag{3}$$

and hence we have $\bar{w} \approx \sqrt{2/p}$ (since p is generally small). Thus in the long time, the average throughput is

$$\frac{C}{rtt}\sqrt{\frac{1}{p}} \tag{4}$$

where C is a constant determined by the TCP maximum segment size (MSS).

So for ACKs on lowest-latency path of MPTCP, if each subflow has the same MSS, the throughput improvement I is:

$$I = \frac{\sum T_i'}{\sum T_i} - 1 = \frac{\sum \frac{C}{d_0+d_i}\sqrt{\frac{1}{p_i}}}{\sum \frac{C}{2d_i}\sqrt{\frac{1}{p_i}}} - 1 = \frac{\sum \frac{1}{d_0+d_i}\sqrt{\frac{1}{p_i}}}{\sum \frac{1}{2d_i}\sqrt{\frac{1}{p_i}}} - 1 \tag{5}$$

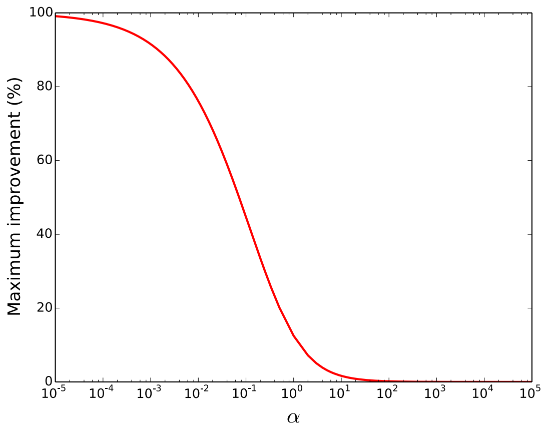

Fig. 2. Maximum improvement under different α (TCP-Reno)

In order to determine the maximum improvement when the loss rates $p_0, p_1, \cdots, p_N$ are fixed, we calculate the partial derivatives of I with respect to d_k, $\forall k \in \{1, 2, \cdots, N\}$:

$$\frac{\partial I}{\partial d_k} = \frac{-\frac{1}{(d_0+d_k)^2}\sqrt{\frac{1}{p_k}}\sum \frac{1}{2d_i}\sqrt{\frac{1}{p_i}} + \frac{1}{2d_k^2}\sum \frac{1}{d_0+d_i}\sqrt{\frac{1}{p_i}}}{(\sum \frac{1}{2d_i}\sqrt{\frac{1}{p_i}})^2} \tag{6}$$

Let the numerator be zero, we have:

$$\frac{2d_k^2}{(d_0+d_k)^2} = \frac{\sum \frac{1}{d_0+d_i}\sqrt{\frac{1}{p_i}}}{\sum \frac{1}{2d_i}\sqrt{\frac{1}{p_i}}} \tag{7}$$

$\forall k \in \{1, 2, \cdots, N\}$. This implies that we have maximum improvement when $d_1 = d_2 = \cdots = d_N$. Let $d_1 = d_2 = \cdots = d_N = \beta d_0$, where $\beta \geq 1$, then we have

$$I = \frac{\frac{1}{2d_0}\sqrt{\frac{1}{p_0}} + \sum_{i>0} \frac{1}{d_0+d_i}\sqrt{\frac{1}{p_i}}}{\frac{1}{2d_0}\sqrt{\frac{1}{p_0}} + \sum_{i>0} \frac{1}{2d_i}\sqrt{\frac{1}{p_i}}} - 1 \tag{8}$$

$$= \frac{\sqrt{\frac{1}{p_0}} + \sum_{i>0} \frac{2}{1+\beta}\sqrt{\frac{1}{p_i}}}{\sqrt{\frac{1}{p_0}} + \sum_{i>0} \frac{1}{\beta}\sqrt{\frac{1}{p_i}}} - 1 \tag{9}$$

$$= \frac{\sqrt{\frac{1}{p_0}}/\sum_{i>0}\sqrt{\frac{1}{p_i}} + \frac{2}{1+\beta}}{\sqrt{\frac{1}{p_0}}/\sum_{i>0}\sqrt{\frac{1}{p_i}} + \frac{1}{\beta}} - 1 \tag{10}$$

Let $\alpha = \sqrt{1/p_0}/\sum_{i>0}\sqrt{1/p_i}$, we have

$$I = \frac{\alpha + \frac{2}{1+\beta}}{\alpha + \frac{1}{\beta}} - 1 = \frac{\beta - 1}{\alpha\beta^2 + (\alpha+1)\beta + 1} \tag{11}$$

When $\beta = 1 + \sqrt{2}\sqrt{1 + 1/\alpha}$, I has maximum value $1/(3\alpha + 1 + 2\sqrt{2}\sqrt{\alpha(\alpha+1)})$.

Fig. 2 shows the maximum improvement under different α's. 1) When $\alpha \to 0$, i.e., either path 0 has much higher loss rate or N is large enough, path 0 has trival effect on overall throughput. In the meantime, $\beta \to \infty$, i.e., path 0 has a much smaller one-way latency, which implies the RTT of path i ($\forall i > 0$) will decrease by nearly one half and throughput will increase by nearly 100%. So the maximum improvement can be as large as 100%. 2) When $\alpha \to \infty$, path 0 has much smaller loss rate and it dominates the overall throughput. So the maximum improvement can be as small as 0.

B. MPTCP-Lia / MPTCP-Olia

MPTCP-Lia (Linked increase algorithm) [2] and MPTCP-Olia (Opportunistic linked increase algorithm) [8] are two congestion control algorithms designed specifically for MPTCP, for the purpose of satisfying the following two requirements:

- An MPTCP flow should have at least as much throughput as an SPTCP flow on the best path;
- An MPTCP flow should take no more capacity on any collection of paths than an SPTCP flow on the best of these paths.

So when a data packet associated with a subflow is delivered, unlike TCP-Reno, MPTCP-Lia/MPTCP-Olia increases the window size by a factor depending on not only the status of this subflow, but also the statuses of other subflows. Ideally, if these two requirements are satisfied, we can expect an MPTCP connection to have the throughput as same as an SPTCP flow on the best path under our assumptions in the beginning of this section. So

$$T = \max_i \frac{C}{2d_i}\sqrt{\frac{1}{p_i}} \tag{12}$$

$$T' = \max_i \frac{C}{d_0 + d_i}\sqrt{\frac{1}{p_i}} \tag{13}$$

if each subflow has the same MSS. Thus

$$I = \frac{T'}{T} - 1 = \frac{\max_i \frac{1}{d_0+d_i}\sqrt{\frac{1}{p_i}}}{\max_i \frac{1}{2d_i}\sqrt{\frac{1}{p_i}}} - 1 \tag{14}$$

Suppose $m \in \operatorname{argmax}_i \sqrt{1/p_i}/(d_0 + d_i)$. Then we have

$$I = \frac{\frac{1}{d_0+d_m}\sqrt{\frac{1}{p_m}}}{\max_i \frac{1}{2d_i}\sqrt{\frac{1}{p_i}}} - 1 \tag{15}$$

$$\leq \frac{\frac{1}{d_0+d_m}\sqrt{\frac{1}{p_m}}}{\frac{1}{2d_m}\sqrt{\frac{1}{p_m}}} - 1 < 2 - 1 = 1 \tag{16}$$

So here the maximum improvement is also upper bounded by 100%. When the best path for SPTCP is always the same one (other than path 0), and d_0 is small enough, we have $I \to 100\%$. And if the best path for SPTCP is always path 0, or always has the same one-way latency as path 0, we have $I = 0$.

C. Other congestion control algorithms

Some TCP congestion control algorithms are designed carefully in order to leverage all the available bandwidth and packets rarely get dropped. This holds true for nearly all the rate control algorithms, such as TCP-Vegas [9]. So they may not be the congestion control algorithms we want to consider under the context of this section.

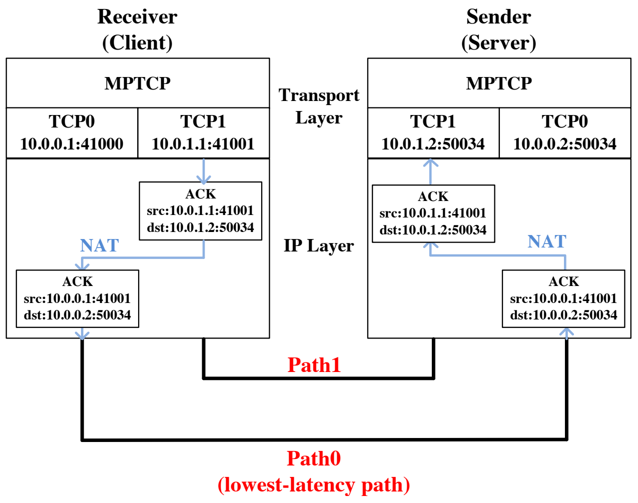

Fig. 3. Network address translation

For some other congestion control algorithms, it is not easy to determine the improvement mathematically. For example, TCP-Cubic [10] increases window size by a factor based on a cubic function where the time since last data packet loss is the argument. In this case, the long time average throughput for fixed RTT and loss rate is hard to be determined mathematically, hence the theoretical improvement is also unclear.

IV. Design

To design a system that always sends ACKs through the shortest path, there are two major functions need to be enabled. In order to shift ACKs to another path, the source and destination IP address of ACK packets should be modified before being sent, and port numbers can be used to determine the subflows they originally belong to; In order to ensure that lowest-latency path is always selected, the RTT of each subflow need be sampled periodically. In our design, network address translation (NAT) and real-time active probing are used to realize these two functionalities.

A. Network address translation

Fig. 3 illustrates the principle of our NAT program. The standard MPTCP protocol itself is not changed at both receiver and sender side. However, MPTCP receiver needs to launch an NAT program to monitor all the packets passing through its IP layer. As shown in Algorithm 1, when the NAT program detects a packet that belongs to our MPTCP connection, and it is a normal ACK packet (not SYNACK or FINACK), it will modify its source and destination IP address to match with the lowest-latency path. This way, the lower layer can deliver this ACK packet on lowest-latency path, instead of its original path. In Fig. 3, ACK packet spawned by receiver-side TCP

Algorithm 1 MPTCP Reciver-Side NAT program

```
1: // Subflow info of MPTCP connection
2: $SFs$ = $[(sip\_0, dip\_0, sport\_0, dport\_0, prot\_0),$
          $(sip\_1, dip\_1, sport\_1, dport\_1, prot\_1),$
          $\cdots,$
          $(sip\_N, dip\_N, sport\_N, dport\_N, prot\_N)]$
3: // index of lowest-latency path
4: $llp\_idx = \cdots$
5: // the following function is called whenever an IP packet $p$ passes IP Layer
6: ReceiverNATFunc($p$):
7: if $(p.sip, p.dip, p.sport, p.dport, p.prot) \in SFs$ and $(p.SYN == 0$ and $p.FIN == 0$ and $p.ACK == 1)$ then
8:     $p.sIP = sip\_\{llp\_idx\}$
9:     $p.dIP = dip\_\{llp\_idx\}$
10: end if
```

Algorithm 2 MPTCP Sender-Side NAT program

```
1: // Subflow info of MPTCP connection
2: $SFs$ = $[(sport\_0, dport\_0, prot\_0) : (sip\_0, dip\_0),$
          $(sport\_1, dport\_1, prot\_1) : (sip\_1, dip\_1),$
          $\cdots,$
          $(sport\_N, dport\_N, prot\_N) : (sip\_N, dip\_N)]$
3: // the following function is called whenever an IP packet $p$ passes IP Layer
4: SenderNATFunc($p$):
5: if $p.ACK == 1$, and $(p.sip, p.dip) == (sip\_k, dip\_k)$ for some $k$, and $SF[(p.sport, p.dport, p.prot)]! = null$ then
6:     $p.sip, p.dip = SF[(p.sport, p.dport, p.prot)]$
7: end if
```

socket of subflow 1 are detected by the NAT program, and its source and destination IP address are reset to 10.0.0.1 and 10.0.0.2, respectively. Similarly, MPTCP sender also needs to launch an NAT program. As shown in Algorithm 2, when the NAT program detects that an ACK packet belongs to our MPTCP connection, it will set its source and destination IP address back, based on the subflow information (more specifically, source and destination port numbers) we saved earlier. This way, the ACK packet can be delivered to the correct TCP socket. In Fig. 3, for the ACK packet received on the lowest-latency path at the sender side, the NAT program sets its source and destination IP address back to 10.0.1.1 and 10.0.1.2, respectively.

The correctness of our design is guaranteed by a simple fact: MPTCP client uses different port numbers for different subflows. So even though we may change the source and destination IP address of an ACK, we can still recognize it by its source (when client is receiver) or destination port number (when client is sender).

B. Real-time active probing

In practice, the shortest path llp_idx may change when network link fails in real time or some paths suddenly become congested. So active probing is needed to measure the real-time RTT of all paths and further select the shortest path among them.

The active probing module is designed as follows. The lowest-latency path is updated per time interval τ. At each time interval, MPTCP receiver sends out several probing packets, and response packets are sent back by the remote side to measure RTT. In our design, we use ICMP protocol for this functionality. Once we discover the lowest-latency path changes from one to another, and the latency difference between the old one and the new one is greater than a threshold value λ, we will update llp_idx in Algorithm 1, so that following ACK packets can be sent on the up-to-date lowest-latency path as soon as possible. At MPTCP receiver side, however, we don't necessarily need to know which one is the lowest-latency path. We passively apply the NAT rule to all the ACK packets that belong to our connection.

If the lowest-latency path fails, the communications of other subflows may suspend for some time. At the receiver side, the active probing module needs some time ($\leq \tau$) to discover the failure of path 0 and then update the lowest-latency path; at the sender side, we also need to wait one RTO ($\leq 2\tau$) for the next retransmission. In different tests the suspension time may vary, but it should be upper bounded by 3τ.

There is a tradeoff between responsiveness and stability when choosing the value of τ and λ: small τ and λ means module is sensitive to small changes and the lowest-latency path is updated quickly, while large τ and λ means the lowest-latency path is updated in a stable yet slow way.

V. Experiments

Our experiments are based on Multipath TCP Linux kernel implementation (kernel version 4.14.70 and MPTCP version 0.94) [11]. We implement MPTCP receiver-side/sender-side NAT program as two kernel modules. We set up three independent paths: path 0, path 1 and path 2 between the sender and the receiver. The bandwidth of each path is set to 1Gbps full duplex.

In the first experiment, to verify our analysis in Section III, we let network condition be static. In the second experiment, we change network condition in real time, to test the effectiveness of real-time active probing. In the last experiment, to test the robustness of real-time active probing, we let lowest-latency path fail in real time and measure the time it takes for recovery.

A. Static network condition

We first measure the performance of our ACKs on lowest-latency path idea when network condition is static. For path 0, we let $p_0 = 1\%$, $d_0 = $ 10ms; for path 1, we let $p_1 = 0.02\%$ and change the value of d_1 ($\geq$ 10ms) in different tests (latency and loss are simulated by netem functionality [12] provided by Linux tc tool). And we consider three different

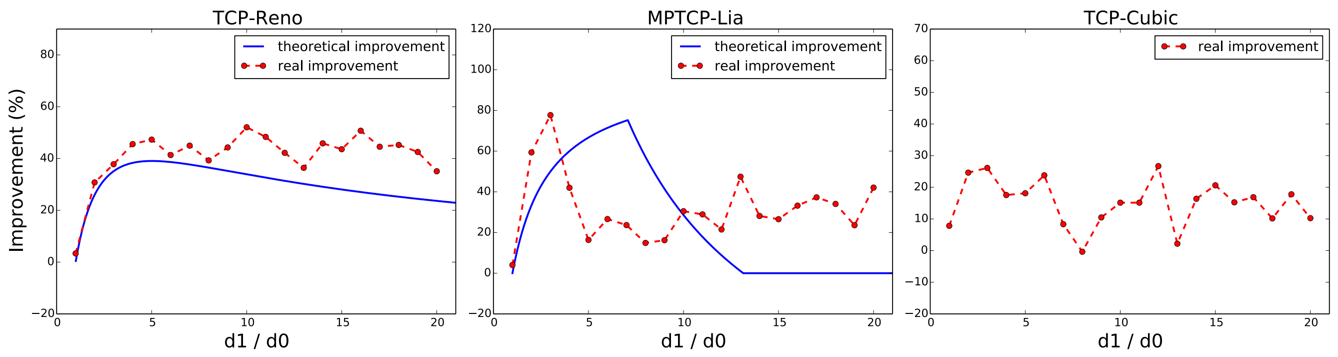

Fig. 4. Improvements for different congestion control algorithms ($p_0 = 1\%, p_1 = 0.02\%$)

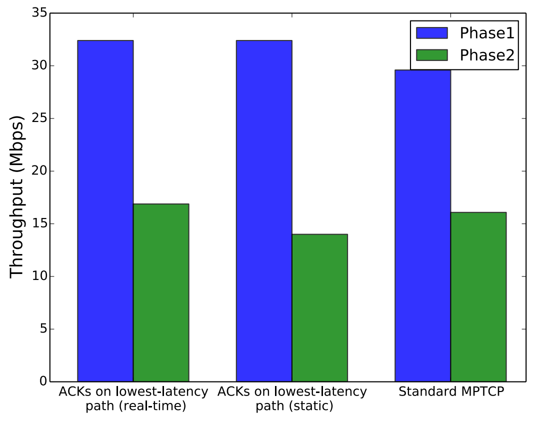

Fig. 5. Throughput under dynamic network condition (TCP-Reno)

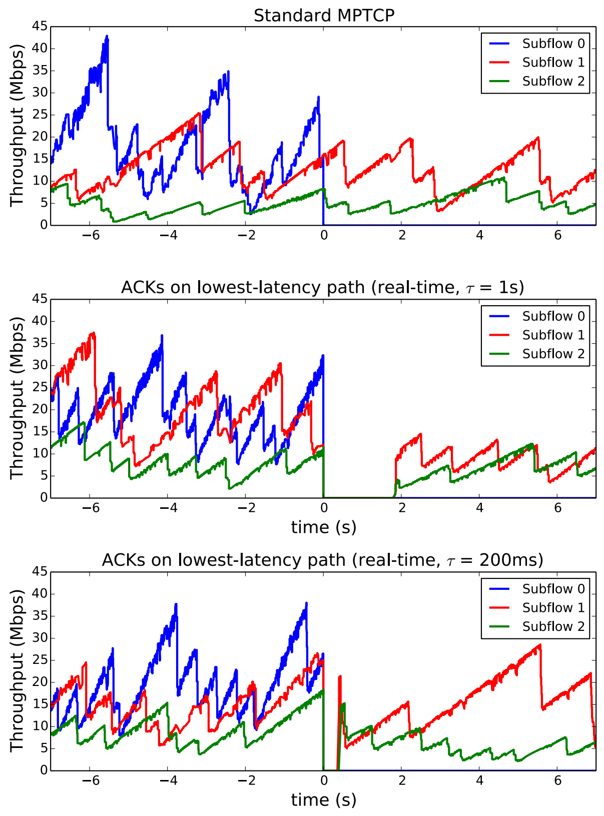

Fig. 6. Real-time throughput before/after link failure (TCP-Reno)

congestion control algorithms in our experiment: TCP-Reno, MPTCP-Lia and TCP-Cubic. The throughput improvements for different congestion control algorithms are shown in Fig. 4, where blue curves and red curves show the theoretical and real improvements, respectively.

1) TCP-Reno: The average real improvements are a little bit higher than theoretical computations. When $2 \leq d_1/d_0 \leq 20$, the average real improvement ranges from 30% to 50%, while the corresponding theoretical improvement should be 20% to 40%. The gap is believed to be due to the randomness of our tests and the small discrepancy between mathematical model and reality.

2) MPTCP-Lia: The average real improvements do not match with theoretical computations. When $d_1/d_0 = 3$, the average real improvement reaches its maximum value 77%; when $4 \leq d_1/d_0 \leq 20$, the average real improvement is around 15-50%. On the other hand, the theoretical improvement reaches its maximum value 75% when $d_1/d_0 = 7.07$. This gap is believed to result from MPTCP-Lia algorithm itself, who has the following two deficiencies in practice:

- it makes approximations when tries to satisfy the two requirements (in Section III-B);
- its average throughput converges much slower than SPTCP congestion control algorithms, since multiple congestion windows are changing in real time simultaneously.

So in practice, when we transfer a file of hundreds of MBs or several GBs, MPTCP-Lia's throughput deviates from the two requirements significantly, and hence (14) can be quite inaccurate.

3) MPTCP-Cubic: Generally, when $2 \leq d_1/d_0 \leq 20$, the average real improvement is within 10-25%.

On average, we generally have a positive real improvement ranges from 10 to 50% in most cases. This shows the effectiveness of sending ACKs along the shortest path.

B. Dynamic network condition

Next, we measure the performance of real-time active probing. Our experiment contains two phases: in phase 1, we let $d_0 = 10$ms; in phase 2, we let $d_0 = 30$ms. And we let $p_0 = p_1 = 0.1\%$ and $d_1 = 20$ms for both of these two phases. We measure the throughput when three different MPTCP versions are used: standard MPTCP, MPTCP with ACKs on static lowest-latency path (which is fixed to path 0), MPTCP with ACKs on real-time lowest-latency path (which is determined by active probing). And the congestion control algorithm used in this experiment is TCP-Reno.

Fig. 5 shows the throughputs of different versions during different phases. In phase 1, MPTCPs with ACKs on lowest-latency path have 10% higher throughputs than standard MPTCP. However, when d_0 increases from 10ms to 30ms, fixed lowest-latency path performs worse than standard MPTCP, since the real lowest-latency path is no longer path 0. On the other hand, our real-time active probing is able to detect this change and correctly update the lowest-latency path to path 1, so that in phase 1 we still have a throughput improvement of 5% compared to standard MPTCP.

This experiment verifies the effectiveness of real-time active probing.

C. Link failure

In the end, we let lowest-latency path fail in real time to test the robustness of real-time active probing. In this experiment, we let $d_0 = 10$ms, $d_1 = 20$ms, $d_2 = 30$ms, and $p_0 = p_1 = p_2 = 0.1\%$. We let path 0 fail in the middle of the communication (for convenience, the time that path 0 fails is set as zero), and observe the changes of real-time throughput of each subflow. Two different MPTCP versions are compared: standard MPTCP, MPTCP with ACKs on real-time lowest-latency path. For ACKs on real-time lowest-latency path, we consider two different real-time active probing intervals: $\tau =$1s and $\tau =$200ms. The congestion control algorithm used in this experiment is TCP-Reno.

Fig. 6 shows the real-time throughputs of different tests before/after link failure. For standard MPTCP protocol, the failure of path 0 doesn't affect the other two subflows at all, since each subflow operates independently. However, for MPTCP with ACKs on real-time lowest-latency path, when $\tau =$1s and $\tau =$200ms, the failure of path 0 makes the communications of the other two subflows suspend for 1.8s and 600ms, respectively. After the resumption of transmissions, subflow 2 has a higher average throughput (8.4Mbps) than its counterpart in standard MPTCP protocol (7.4Mbps), which benefits from ACKs on lowest-latency path (which is path 1 after failure).

This experiment verifies the robustness of real-time active probing: our module is able to discover the new lowest-latency path after link failure and resumes the transmissions of other subflows.

VI. Conclusion

In this paper, we propose to send all MPTCP's ACKs along the lowest-latency path to increase throughput. The throughput improvement is examined analytically which depends on path condition and the congestion control algorithm used with its maximum approaching 100%. Experiments then show that the average improvement generally ranges from 10% to 50% for usual conditions.

There are several directions along which we can extend this study. For example, in Section II, it is assumed that the path lantency will not be affected by ACK traffic. However, This may not hold when there is enough ACK traffic along a certain link which can become congested and introduce significant latency. It is therefore worthwhile characterizing the condition under which such a problem can happen and the corresponding consequences. Another direction concerns the complexity of NAT in Section IV which grows linearly in the number of packets that pass through the IP layer. It is conceivable that such overhead can be significantly reduced by adding our functionality directly into the MPTCP protocol.

References

[1] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How hard can it be? designing and implementing a deployable multipath tcp," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 29–29.

[2] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath tcp." in *NSDI*, vol. 11, 2011, pp. 8–8.

[3] N. Kuhn, E. Lochin, A. Mifdaoui, G. Sarwar, O. Mehani, and R. Boreli, "Daps: Intelligent delay-aware packet scheduling for multipath transport," in *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2014, pp. 1222–1227.

[4] H. Shi, Y. Cui, X. Wang, Y. Hu, M. Dai, F. Wang, and K. Zheng, "Stms: Improving mptcp throughput under heterogeneous networks," in *2018 USENIX Annual Technical Conference (USENIXATC 18)*. USENIX Association, 2018.

[5] F. Yang, Q. Wang, and P. D. Amer, "Out-of-order transmission for in-order arrival scheduling for multipath tcp," in *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. IEEE, 2014, pp. 749–752.

[6] T. Kurosaka and M. Bandai, "Multipath tcp with multiple acks for heterogeneous communication links," in *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*. IEEE, 2015, pp. 613–614.

[7] C. Xu, P. Wang, C. Xiong, X. Wei, and G.-M. Muntean, "Pipeline network coding-based multipath data transfer in heterogeneous wireless networks," *IEEE Transactions on Broadcasting*, vol. 63, no. 2, pp. 376–390, 2017.

[8] R. Khalili, N. Gast, M. Popovic *et al.*, "Opportunistic linked-increases congestion control algorithm for mptcp," 2013.

[9] L. S. Brakmo and L. L. Peterson, "Tcp vegas: End to end congestion avoidance on a global internet," *IEEE Journal on selected Areas in communications*, vol. 13, no. 8, pp. 1465–1480, 1995.

[10] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[11] C. Paasch, S. Barre, and et al., "Multipath tcp in the linux kernel." [Online]. Available: https://www.multipath-tcp.org

[12] "Netem." [Online]. Available: https://wiki.linuxfoundation.org/networking/netem