OCPS:Offset Compensation based Packet Scheduling Mechanism for Multipath TCP

Dan Ni, Kaiping Xue*, Peilin Hong, Hong Zhang, Hao Lu
The Department of EEIS, University of Science and Technology of China, Hefei, Anhui 230027 China
*kpxue@ustc.edu.cn

***Abstract*—As terminals are equipped with multiple interfaces and allowed to access heterogeneous networks, transferring data simultaneously through all the available paths becomes possible and also brings many benefits. Multipath TCP (MPTCP) distributes an application stream over different TCP connections. Since different paths have disparate latencies, out-of-order packets problem occurs at receiver. Large number of these packets exhaust the limited receive buffer and make the receive window stall, which greatly degrade the throughput. Thus, a scheduling mechanism plays an important role to keep in-order delivery. Previous intelligent scheduling mechanisms schedule data independently each time and doesn't utilize the feedback carried in acknowledgements, which lose flexibility. Our Offset Compensation based Packet Scheduling (OCPS) mechanism gets feedback information from SACK options and gains the knowledge of whether last scheduling round still causes out-of-order problem. Then it modifies the scheduling next round accordingly by using an offset. The simulation results illustrate our mechanism enhance throughput and reduce cache occupancy at receiver.**

I. Introduction

Nowadays, mobile equipments always have more than one network interfaces, such as WiFi and 3G/4G and so on. When the various radio access technologies (RATs) overlap in a place, simultaneously exploring different interfaces becomes possible, which can aggregate path's capacity, improve robustness and balance load. Regarding to multipath transmission, a new module to manipulate different paths should be implemented to the kernel. Theoretically, it can be at any layer only if it is transparent to the application layer. In the context, we'd like to use transport layer solutions [1] [2] [3] [4]. Because it is the lowest layer to keep end-to-end semantics among peers, any other lower layer solutions [5] [6] will confuse TCP, making it hard to distinguish out-of-order from packet loss. Application layer solutions [7] [8] need modifications to applications and let them aware of multiple interfaces. Moreover, TCP can react to the congestion on different paths more instantly while compared with any other transport layer protocols.

Since the latency of each path differs, there is a high probability that the packets with lower sequence numbers sent over a slower path arrive at the sink later than the packets with higher sequence numbers sent over a faster path. Thus, there exists holes in sequence numbers. The receiver has to store large number of out-of-order packets, which will exhaust the limited receive buffer and occupy the receive window. Employing a large buffer at receiver is always a solution, but it wastes the memory. Instead, the problem can be resolved more intelligently at the sender with a little computing by implementing a scheduling mechanism..

IETF's MPTCP working group has been working hard to standardize a multipath protocol on transport layer, which is Multipath TCP (MPTCP) [2]. The networking stack is in Fig.1. It adds MPTCP layer above TCP, which helps manage multiple paths between endpoints, a scheduling function is also implemented to keep in-order delivery. An original data stream is then divided into several segments and sent over multiple TCP connections, which is named as "TCP subflows" in MPTCP. MPTCP introduces dual sequence numbers. Subflow sequence numbers (SSNs) are used at the subflow level to reorder data within each TCP subflow, while data sequence numbers (DSNs) are used at the connection level to reorder data collected from all subflows. Moreover, MPTCP provides coupled congestion control to compete friendly with regular TCP flows.

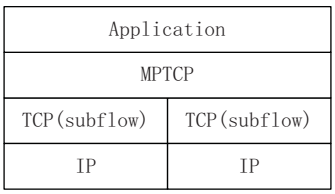

Fig. 1. MPTCP Internet Stack

MPTCP uses a connection-level acknowledgment, DATA ACK, to act as a cumulative ACK for the connection as a whole. In the context, we may also name DATA ACK as CumAck. The DATA ACK indicates how much data has been successfully received without holes. This is in comparison to the subflow-level ACK, which acts analogous to TCP SACK [9], given that there may still be holes in the data stream at the connection level. Besides, TCP SACK options carry related information about the consequence of the scheduling last round, telling sender whether the data arrive in order. In this paper, we propose an Offset Compensation based Packet Scheduling (OCPS) mechanism. The sender gets feedback information from SACK options and has the knowledge whether it schedules too many or too few packets last round. Then, it will modify the value in next scheduling round accordingly.

The remainder of this paper is organized as follows. Section II is the state of art, where many scheduling algorithms

978-1-4673-6432-4/15/$31.00 ©2015 IEEE

for multipath TCP will be introduced. Section III elaborates our mechanism, giving how sender utilizes the feedback in the acknowledgements. In section IV, we compare OCPS with some other scheduling mechanisms by NS-3 simulation. Section V concludes the paper.

II. Related Work

In this section, some scheduling algorithms for multipath TCP will be introduced. Stream Control Transport Protocol (SCTP) is another multipath transport layer protocol, which is proposed earlier than MPTCP, which has already been implemented into the kernel. Even though, SCTP has drawbacks, such as it can't compatible with regular TCP and it is aggressive to other regular TCP flows at bottleneck. That's why MPTCP is being proposed and received much attention. Most of the scheduling mechanisms introduced in this section are based on SCTP. Moreover, they can be applied to MPTCP with only a few modifications.

Load Sharing for SCTP (LS-SCTP) [10] supports weighted round robin and distributes data to each path in proportion to the ratio $cwnd/RTT$(congestion window/round trip time). However, it is coarse-grained and can't ensure in-order delivery for each packet. WestwoodSCTP (W-SCTP) [11] performs a more intelligent Bandwidth Aware Scheduling at sender, which is named as BAS. It scores for each path, and the path with the lowest score has the highest priority to transmit packets. It tries to provide in-order delivery but still suffers from serious performance degradation if the paths in an association have significantly different latencies.

Forward Prediction Packet Scheduling (FPS) for multi-interface terminals with disparate latencies is introduced in [12], [13], which is verified in SCTP. When a path under scheduling frees congestion window space to pull new data from sending buffer, it estimates the duration of this new transmission. Then it predicts the number of packets (N) that can be delivered simultaneously in other paths during this given duration. Then the under-scheduling path chooses $(N+1)$th packet and the following ones from the sending buffer to fill its congestion window.

MPTCP scheduler [14] is similar with FPS, which is implemented in Linux MPTCP kernel [15]. The amount of data scheduled on each TCP subflow is in proportion to the estimated bandwidth of the path, calculated by $BW = cwnd/RTT$. In addition, it has the intelligence to choose which packet to allocate from the shared sending buffer. However, these two mechanisms schedules every round independently without utilizing the feedback of the previous predictions. Meanwhile, they all ignore the packet loss when scheduling.

We have already proposed a scheduling mechanism for MPTCP, which is more suitable in lossy networks, named F^{2}P-DPS [16]. It schedules packets as the same way in FPS, however the algorithm to estimate N is different. F^{2}P-DPS is more adaptive and suitable for wireless networks with inevitable packet loss during transmission. It utilizes the idea of TCP modeling. Since the estimation is in a statistical sense, it may not instantly react to the varying of path condition.

All the scheduling algorithms ignore the feedback carried in the acknowledgements, which should have clues about previous scheduling. Thus, the sender has no prior knowledge to correct scheduling of the next round and has to do the scheduling each round independently.

III. Offset Compensation based Packet Scheduling Mechanism(OCPS)

In this section, we present Offset Compensation based Packet Scheduling (OCPS) for MPTCP. Scheduling function at MPTCP layer collects the path conditions while feedback module collects the knowledge of previous predictions. Here, the basic scheduling mechanism can be FPS,F^{2}P-DPS or any other mechanisms, OCPS provides modification for these basic scheduling algorithms.

When using FPS or F^{2}P-DPS, the path with larger latency doesn't take the bottommost packets to send. Instead, scheduling function firstly predicts the arrival time (i.e, t') of the packets on this subflow in a new round and calculates the total data amount (i.e, N) that can be sent on other subflows before t'. Then this under-scheduling subflow skips the first N packets in the send buffer and chooses the $(N+1)$th packet with the following ones to send.

However, the estimation of N in each round isn't always precise. It may be larger or smaller than the true value due to the varying of path conditions.

A. Overview: Two Specific Situations

We assume there are $subflow_i$ and $subflow_j$, where $RTT_i << RTT_j$. Dual space sequence numbers are used in the form (DSN, SSN), where the former item means connection level sequence number and the latter one means subflow level sequence number. (DSN, SSN) contained in DATA segments indicates the mapping of DSN and SSN. Meanwhile, (DSN, SSN) contained in ACK segments indicates the next expected sequence number at the connection level and the subflow level. We assume DSN starts with 0, and $SSNs$ on $subflow_i$ and $subflow_j$ starts with 10831, 566 respectively. In this way, the spaces of DSN and SSN will not overlap. The packet size is equal to 1400bytes. Each time $subflow_j$ sends packets, it needs scheduling. $subflow_i$ always takes the bottommost packets to send, while $subflow_j$ takes $(N+1)$th packet and the following ones to send. At the scheduling time all the transmission events haven't occurred yet, the sender estimates N by using scheduling algorithms (e.g, FPS, F^{2}P-DPS), estimated N is always different from the true value. Thus, two different situations may occur:

1) ***Situation1***: the estimation of N is too large.

The sender distributes too many packets on $subflow_i$. Then, the packets sent on the $subflow_j$ arrive at receiver too early and have to wait for reordering at receive buffer. It is just the example showed in Fig.2, we assume at scheduling time the congestion window ($cwnd$) on $subflow_i$ and $subflow_j$ are 2 and 3 separately. And $cwnd$ is increased by 1 every round. In the figure, packets on $subflow_j$ arrive at receiver after the first 5 packets sent on $subflow_i$. Thus, if $N = 5$ and $subflow_j$

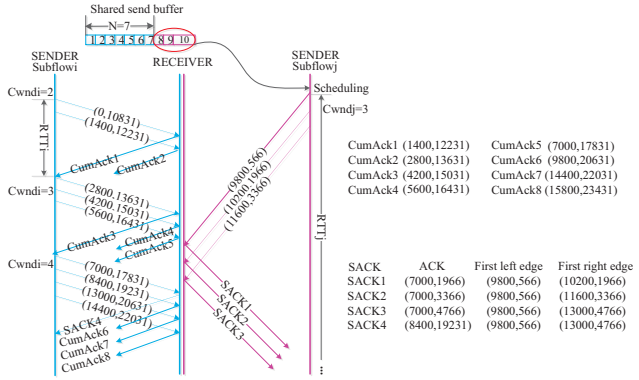

Fig. 2. Situation1:The Estimation of N is Too Large

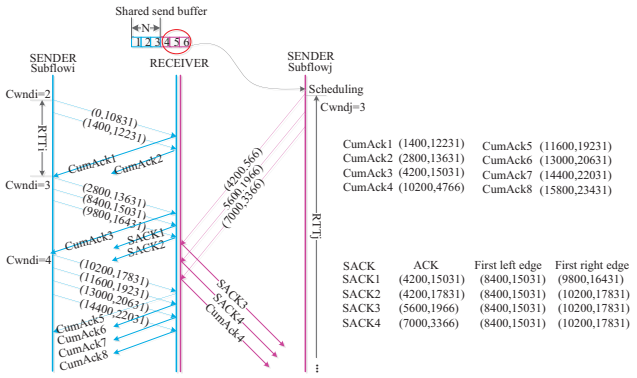

Fig. 3. Situation2:The Estimation of N is Too Small

takes the 6, 7, 8th packets to fill its congestion window, all the packets will arrive at the receiver in order.

However, in this situation, the sender predicts $N = 7$ and takes 8, 9, 10th packets from the shared sending buffer to send on $subflow_j$. The corresponding sequence numbers are (9800, 566), (10200, 1966), (11600, 3366), which are in the form $(DSN,\ SSN)$. The first 5 packets sent on $subflow_i$ arrive at the receiver in order and return cumulative ACKs($CumAcks$). But the following 3 DATA segments sent on $subflow_j$ arrive at the receiver too early and trigger SACK. We take $SACK1$ as a example to explain how SACK works. The ACK field of $SACK1$ is (7000, 1966), where 7000 is the next expected sequence number on the connection level and 1966 is the next expected sequence number on $subflow_j$. First left edge (9800, 566) and first right edge (10200, 1966) are two boundaries of the first block storing continuous out-of-order packets. These out-of-order packets are the packets that arrive at the receiver earlier than expected time while the packets with lower sequence numbers still haven't arrive yet. In additions, more blocks need more fields to carry the boundaries. When the 8th segment (9800, 566) arrives at the receiver, the sequence numbers at the connection level have holes because 7000 and 8400 haven't arrived. Thus, the segment (9800, 566) is stored and the first block boundaries are carried in the SACK options. Then, there are another 3 SACKs. When the data segment (8400, 19231) arrives at the receiver, the packets stored in receive buffer are all reordered and a $CumAck$ is returned again.

2) ***Situation2***: the estimataion of N is too small.

The sender schedules too few packets on $subflowi$. Then, the packets sent on $subflow_j$ arrive late and the receiver has to store packets from $subflow_i$ for a while until the packets are in order. It is the example showed in Fig.3. In this situation, $subflow_j$ predicts $N = 3$ and takes 4, 5, 6th packets to fill its congestion window at scheduling time, while the true value is still 5. The corresponding sequence numbers of these three data segments are (4200, 566),(5600, 1966),(7000, 3366). The first 3 packets sent on $subflow_i$ arrive at the receiver in order and $CumAcks$ are returned consequently. When the 4th packet (8400, 15031) on $subflow_i$ arrives at the receiver, there are holes at the connection level because the DATA segments with DSN being 4200, 5600, 7000 which scheduled to $subflow_j$ haven't arrived yet. Thus $SACK1$ is triggered and sent back on $subflow_i$, which contains the boundaries of the first continuous out-of-order packets, the first left edge is (8400, 15031) and first right edge is (9800, 16431). After that, there are another 3 SACKs. When data segment (7000, 3366) arrives the receiver, the packets stored in the receive buffer are all reordered and a corresponding $CumAck$ is returned again.

The more the estimation deviates from the true value, the worse the performance is. The main issue of our algorithm is how to distinguish ***Situation1*** from ***Situation2*** and then set an offset for each subflow to modify its estimated N.

B. Terminology

For each $subflow_k$, we introduce 5 parameters, N_k, a_k, $athresh_k$, b_k, $bthresh_k$. In order to explain the algorithm more clearly, we introduce two concepts, master-subflow and slave-subflow, which will be explained below.

N_k: the amount of packets distributed to other subflows by using scheduling algorithms(e.g, FPS, F^{2}P-DPS...) at $subflow_k$'s scheduling time.

a_k: the offset for $subflow_k$ to modify N_k when N_k last round is too large. It is negative.

$athresh_k$: the threshold for $subflow_k$ to determine whether to exponentially or linearly decrease a_k.

b_k: the offset for $subflow_k$ to modify N_k when N_k last round is too small. It is positive.

$bthresh_k$: the threshold for $subflow_k$ to determine whether to exponentially or linearly increase b_k.

Two concepts (master-subflow and slave-subflow): for each subflow, any subflow with longer RTT is its master-subflows, and the subflow with shoter RTT is its slave-subflow.

The master-subflow (e.g, $subflow_k$) calculates N_k by estimating the number of the packets that can be sent simultaneously on all its slave-subflows.

C. The Detailed Algorithm

Since each TCP subflow uses TCP SACK, when packets arrive at the receiver out of order, SACK options will be contained in the acknowledgements. SACK information can specify the blocks of continuous out-of-order segments stored at receiver. There are two cases. In the first case, the out-of-order is within each TCP subflow, which means holes are at the subflow level. Each TCP subflow may operate loss recovery algorithm according to standard TCP when receiving an acknowledge. In the latter case, the out-of-order is among different subflows, which means holes are at the connection level. In this case, which is also the emphasis we will talk about in detail later, the SACK submitted to MPTCP layer is an indication that the prediction deviates from the true value. Then, the sender determines whether it is in ***Situation1*** or ***Situation2***, which subflow was wrongly estimated last round and which subflow should update its offset. Now, let's rethink those two situations in the two-subflow MPTCP scenario mentioned above.

In ***Situation1***, the sender receives SACK from $subflow_j$ and finds out no loss indication within $subflow_j$. Hence, it infers that the out-of-order are between subflows, which is caused by wrongly estimating in last scheduling round. After comparing the DSNs in the hole with the DSNs in the send buffer, the sender finds out the packets in the hole were scheduled on $subflow_i$. Since $subflow_j$ is master-subflow of $subflow_i$, this SACK means the estimated N_j last round is too large. When $subflow_j$ is ready to send packets next scheduling round, the sender modifies N_j as $N'_j = N_j + a_j$, where N_j is the theoretical value calculated by using FPS or F^{2}P-DPS and a_j is a negative offset, which makes theoretical value approximates the real value step by step. Here, we define another parameter $asthresh_j$, which is a threshold to decide whether a_j is exponentially decreased or linearly decreased. If continuous SACK indicates it is in ***Situation1***, the offset should be decreased according to the algorithm. Otherwise, when a SACK indicates it changes from ***Situation1*** to ***Situation2***, the offset should be cleared and $athresh_j$ should be halved.

In ***Situation2***, the sender receives SACK from $subflow_i$ finds out no loss indication within $subflow_i$. Thus, it indicates the out-of-order are between subflows. Then the sender discovers the packets in the hole were scheduled to $subflow_j$. Since $subflow_j$ is master-subflow of $subflow_i$, this SACK means the estimated N_j last round is too small. When $subflow_j$ is ready to send packets in the next round., the sender modifies N_j as $N'_j = N_j + b_j$, where b_j is a positive offset. Also $bthresh_j$ is needed to determine whether b_j is exponentially increased or linearly increased. If continuous SACK indicates it is in ***Situation2***, the offset should be increased. Otherwise, when a SACK indicates it changes from ***Situation2*** to ***Situation1***, the offset should be cleared and $bthresh_j$ will be halved.

OCPS algorithm description is showed in Algorithm.1. If the number of subflows is more than 2, the algorithm is also applicable. When SACK is returned from $subflow_k$ and it can't handle the out-of-order problem by simply retransmitting, which means the packets in the holes are sent on other subflows, it operates in two folds:

- ***Step1***: The sender finds whether the packets in some holes were scheduled to $subflow_k$'s slave-subflows. If so it means at $subflow_k$'s scheduling time, the sender is supposed to distribute too many packets to its slave-subflows, estimated N_k of $subflow_k$ is larger than the true value. Thus, when $(N_k + 1)$th packet on $subflow_k$ arrives at receiver, some of the first N_k packets are still on the way, which causes the holes. The offset a_k for $subflow_k$ is modified accordingly and N_k for the next scheduling round should be increased by a_k, where a_k is negative.
- ***Step2***: The sender checks whether there are packets in the holes were scheduled to $subflow_k$'s master-subflows. If so for $subflow_k$'s each master-subflow (e.g, $subflow_l$) the estimated N_l of last scheduling round is smaller than the true value, the offset b_l is modified accordingly and N_l for the next scheduling round should be increased by b_l, where b_l is positive.

When $subflow_k$'s N_k is continuously indicated too large, a_k is decreased step by step. Once a returned SACK indicates

Algorithm 1 OCPS Algorithm Description

Require: The sender receives $subflow_k$'s SACK and updates the offset.
if some holes belong to $subflow_k$'s slave-subflows **then**
  if Continuously indicating N_k is too large **then**
    if $a_k > athresh_k$ **then**
      exponentially decrease a_k ($a_k = 2a_k$);
    else
      linearly decrease a_k ($a_k = a_k - 1$);
    end if
  else
    halve $bthresh_k$ ($bthresh_k = b_k/2$) ;
    clear b_k ($b_k = 0$);
  end if
end if
if some holes belong to $subflow_k$'s master-subflows **then**
  for each $subflow_l$ in $subflow_k$'s master-subflows **do**
    if Continuously indicating N_l is too small **then**
      if $b_l < bthresh_l$ **then**
        exponentially increase b_l ($b_l = 2b_l$);
      else
        linearly increase b_l ($b_l = b_l + 1$);
      end if
    else
      halve $athresh_l$ ($athresh_l = a_l/2$);
      clear a_l ($a_l = 0$);
    end if
  end for
end if

$subflow_k$'s N_k is too small, a_k is cleared and $athresh_k$ is halved. Similarly, when $subflow_k$'s N_k is continuously indicated too small, b_k is increased step by step. Once a returned SACK indicates $subflow_k$'s N_k is too large, b_k is cleared and $bthresh_k$ is halved.

IV. Performance Evaluation

In this section, we evaluate our scheduling mechanism (OCPS) proposed in this paper on NS3 simulator [17]. The MPTCP NS3 code is provided by google mptcp group [18]. Another two scheduling mechanisms, FPS and F^{2}P-DPS are implemented as comparisons. The main difference between FPS and F^{2}P-DPS is the algorithm to estimate N. Compared with FPS, F^{2}P-DPS is more adaptive and suitable in wireless networks, in which packet loss is inevitable. Sometimes when the path conditions vary a lot, they may still suffer. On the contrary, OCPS is fit for the condition where the path conditions vary a lot. It modifies the estimation by utilizing the feedback in SACK instantly.

1) Simulation Setup

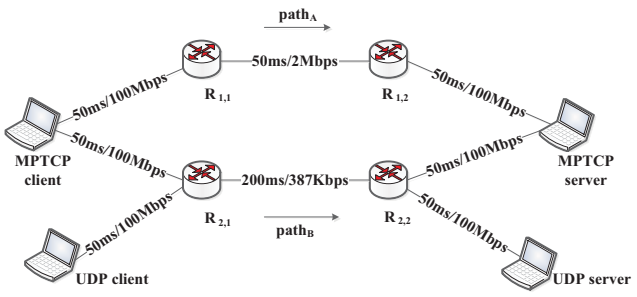

Fig. 4. Simulation Setup

In the simulation, MPTCP client establishes two subflows with MPTCP server. As showed in Fig.4, two subflows are through different paths: $path_A$ and $path_B$. Another pair of UDP client and UDP server produce UDP traffic and compete with MPTCP flow at bottleneck. $R_{i,j}$ is the router on each path, $i = 1$ means it is the router on $path_A$ while $i = 2$ means it is the router on $path_B$. There are two routers on each path, and the link between the $R_{i,1}$ and $R_{i,2}$ is the bottleneck. The path $path_A$'s bottleneck has 2Mbps bandwidth and 50ms latency, which represents a WIFI link. The path $path_B$'s bottleneck has 387Kbps and 200ms latency, which represents a 3G link. The Link queue limit and type on bottleneck are set 100 packets and Droptail, respectively. The loss rate on $path_A$ is set to 0.5%. The ratio of the latency on these two paths is equal to 1:2.

Maximal Segment Size(MSS) in our simulation is set to 1400 bytes, which is also the packet size at TCP layer. The receive buffer of MPTCP is equal to 100MSS(136K). The congestion control algorithm adopts RTT-compensation, which is used widely in MPTCP and provides more friendliness. The simulation time of MPTCP flow is 80s, during which UDP traffic starts at 20s and ends at 40s. The UDP traffic generator produces traffic with uniform distribution and the interval between continuous packets is set to 0.01s. The packet length of each UDP packet is 1024bytes.

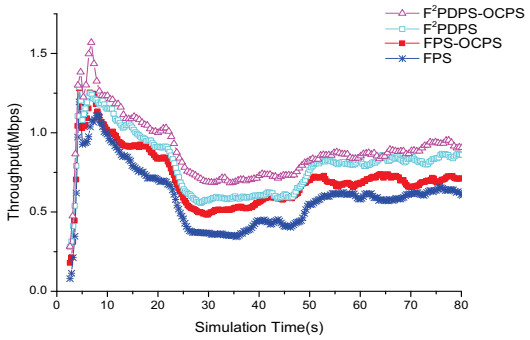

Fig. 5. Comparison of Global Throughput

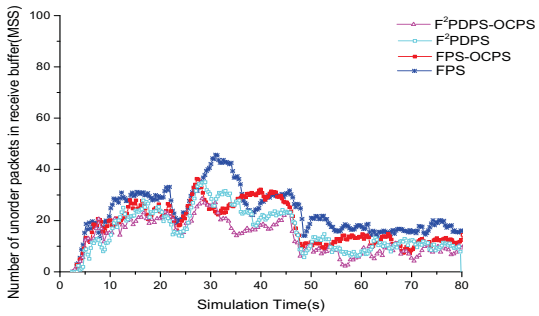

Fig. 6. Comparison of Out-of-order Packet Number

2) Simulation results

Since the path conditions vary a lot, the result of a single sample is unconvincing. We run 100 times and obtain 100 samples in the simulation. The simulation results are the average of all the samples for each scheduling scheme, and mainly presented in two aspects, the global throughput and the number of out-of-order packets.

Fig.5 compares the the global throughput of different mechanisms. For OCPS is an additional algorithm modifying the basic scheduling algorithms, the two curves named FPS-OCPS and F^{2}PDPS-OCPS represent that OCPS is modified from the basic scheduling algorithms being FPS and F^{2}P-DPS respectively. Thus, there are 4 curves in Fig.5, from which we can clearly see that OCPS has larger throughput than FPS and F^{2}P-DPS. During 20s $\sim$ 40s, UDP flow competes with MPTCP flow at bottleneck and the latency on $path_B$ grows heavily. Thus, the throughput of MPTCP degrades. As we know, in lossy networks F^{2}P-DPS works better than FPS, so the throughput is much higher. The throughput improvement by using OCPS which is modified from F^{2}P-DPS is smaller as showed in Fig.5.

In Fig.6 the number of out-of-order packets is presented. Since the results are averaged over all the samples, the maximum number of out-of-order packets doesn't reach 100MSS, which is the size of receive buffer. From Fig.6, we can see the number of out-of-order packets is much fewer when adopting

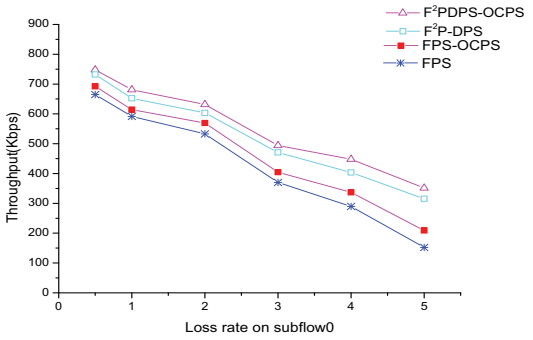

Fig. 7. Comparison of Average Throughput with Varying Loss Rate

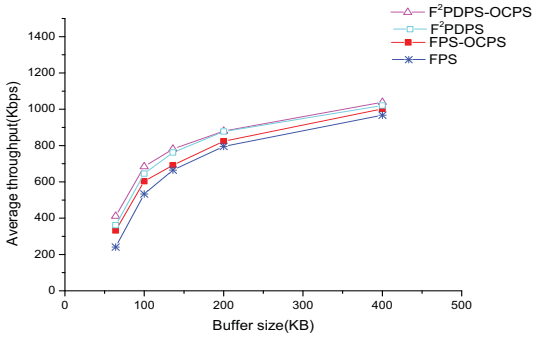

Fig. 8. Comparison of Average Throughput with Varying Buffer Size

OCPS. Since OCPS provides an offset for each scheduling round, the time of the packets stay in the receive buffer as out-of-order ones is much shorter and the number of out-of-order packets measured each time is fewer. From 20s $\sim$ 40s, UDP flow competes with MPTCP flow at bottleneck and the latency on $path_B$ grows heavily. So the number of out-of-order packets grows large and the end-to-end delay increases accordingly, which also corresponds to the decrease of throughput showed in Fig.5.

Fig.7 shows the average throughput with the loss rate on $path_A$ varying from 0.5% to 5%. The average throughput heavily degrades when the loss rate increases. OCPS improves the throughput for MPTCP no matter which basic scheduling algorithm is used. Since F^{2}P-DPS is more adaptive to lossy networks, the average throughput of F^{2}P-DPS is always larger than that of FPS. So the increase of throughput by using OCPS modified from F^{2}P-DPS is smaller. Fig.8 presents the average throughput with the buffer size varying. As the buffer size grows, the throughput of different scheduling mechanisms improves. When the receive buffer size increases from 64KB to 100KB, the throughput nearly doubles. Then with the receive buffer size keep on growing, the gradient decreases.

V. Conclusions

Multipath TCP(MPTCP) can exploit heterogeneous paths by implementing an intelligent packet scheduling mechanism at sender. The segments in a connection should be carefully scheduled to multiple TCP subflows on different paths with minimal occurrence of reordering at receiver. However, some previous scheduling mechanisms, FPS, F^{2}P-DPS schedule each round independently and ignore the feedback information from acknowledgements. In this paper, we propose OCPS, which uses the feedback carried in SACK to modify the next round scheduling. The comparison is among basic scheduling mechanisms (FPS, F^{2}P-DPS) and our proposed one. The results verify that OCPS outperforms the other two basic scheduling mechanisms. Moreover, OCPS can be applied to not only MPTCP, but also any multipath transmission based on TCP which uses SACK.

Acknowledgment

This work is partly supported by the National Natural Science Foundation of China under Grant No. 61170231, No. 61379129, and Intel ICRI MNC.

References

[1] R. Stewart and Q. Xie. Stream Control Transmission Protocol. *RFC 2960*, October 2000.
[2] A. Ford. TCP Extensions for Multipath Operation with Multiple Addresses. *RFC 6824*, January 2013.
[3] Y. Hasegawa, I. Yamaguchi, T. Hama, H. Shimonishi, and T. Murase. Improved Data Distribution for Multipath TCP Communication. In *Proceedings of Global Telecommunications Conference 2005 on (Globecom'05)*, volume 1, pages 5–pp. IEEE.
[4] L. Magalhaesand R. Kravets. MMTP: Multimedia Multiplexing Transport Protocol. *ACM SIGCOMM Computer Communication Review*, 31(2 supplement):220–243, 2001.
[5] J. Kim, T. Ueda, and S. Obana. MAC-level Measurement based Traffic Distribution over IEEE 802.11 Multi-radio Networks. *Consumer Electronics, IEEE Transactions on*, 54(3):1185–1191, 2008.
[6] K. Chebroluand R. Rao. Communication Using Multiple Wireless Interfaces. In *Proceedings of IEEE WCNC*, volume 1, pages 327–331, 2002.
[7] D. Kaspar, K. Evensen, P. Engelstad, and A.F Hansen. Using HTTP Pipelining to Improve Progressive Download over Multiple Heterogeneous Interfaces. In *Proceedings of IEEE International Conference(ICC)*, pages 1–5. IEEE, 2010.
[8] P. Sharma, S. Lee, J. Brassil, and K.G. Shin. Aggregating Bandwidth for Multihomed Mobile Collaborative Communities. *Mobile Computing, IEEE Transactions on*, 6(3):280–296, 2007.
[9] M. Mathis and J. Mahdavi. TCP Selective Acknowledgment Options. *RFC 2018*, October 1996.
[10] P. Amer. Load Sharing for the Stream Control Transmission Protocol (SCTP). *draft-tuexen-tsvwg-sctp-multipath-09*, October 2014.
[11] C. Casetti and W. Gaiotto. Westwood SCTP: Load Balancing over Multipaths Using Bandwidth-aware Source Scheduling. In *Proceedings of Vehicular Technology Conference Fall (VTC2004-Fall)*, volume 4, pages 3025–3029. IEEE, 2004.
[12] F. H Mirani, N. Boukhatem, and M.A. Tran. A Data-scheduling Mechanism for Multi-homed Mobile Terminals with Disparate Link Latencies. In *Proceedings of Vehicular Technology Conference Fall (VTC 2010-Fall), 2010 IEEE 72nd*, pages 1–5. IEEE, 2010.
[13] F. H. Mirani, M. Kherraz, and N. Boukhatem. Forward Prediction Scheduling: Implementation and Performance Evaluation. In *Proceedings of Telecommunications (ICT), 2011 18th International Conference on*, pages 321–326. IEEE, 2011.
[14] S. Barré. *Implementation and Assessment of Modern Host-based Multipath Solutions*. PhD thesis, 2011.
[15] http://www.multipath-tcp.org/.
[16] D. Ni, K. Xue, P. Hong, and S. Shen. Fine-grained Forward Prediction based Dynamic Packet Scheduling Mechanism for Multipath TCP in Lossy Networks. In *Proceedings of International Conference on Computer Communications and Networks(ICCCN2014)*. IEEE, 2014.
[17] www.nsnam.org/.
[18] http://code.google.com/p/mptcp-ns3/.