Multipath IP transmission: Motivation, Design, Performance

ABSTRACT

Multipath Internet transmission has become available in recent years. Most devices have more than one interfaces that can connect to Internet. Especially the booming of mobile devices introduces the greatest chance for multipath communication.

In 2011, multipath TCP was introduced as RFC by IETF. This draft defines the architecture and guideline of multipath TCP development. Also, a popular multipath TCP implementation[1] has been in process for several years. Most recent research on multipath is based on this implementation. But, as the name refers, multipath TCP is only designed for TCP traffic. Inside the current Internet infrastructure, it is still an accepted assumption that most Internet traffic is transmitted via the TCP protocol. However, the rise of new streaming applications and P2P protocols that try to avoid traffic shaping techniques will likely increase the use of other protocols like UDP as a transport protocol.

In this paper, we introduce multipath communication at network layer instead of transmission layer. By introducing multipath capability at network layer, almost all traffic on Internet can benefit from multipath. Also, in the kernel of any operation system, because the huge difference of complexity between TCP and IP, we can expect a much lighter weighted implementation of multipath at network layer. To evaluate the performance of MPIP, we implement our proposition in the latest Linux kernel under Ubuntu system. We prove that besides having the same even better performance as MPTCP for TCP traffic, MPIP also has good support to UDP traffic, and provides potential optimized routing decision for specific applications.

1. INTRODUCTION

Multipath has become available in recent years. Also, IETF proposed RFC 6182 specifically for multipath TCP in 2011. By introducing multipath between both ends for one connection, not only higher throughput can be achieved, different characters on different paths can be complementary to satisfy different user requirement under volatile Internet congestion situations.

In data center network, almost every two nodes are connected by multiple physical paths. Because of this unique structure, multipath has been deployed for a long time to increase throughput and improve reliability.

Most current devices (Mainly mobile devices) have more than one internet interface (4G, WiFi), it is possible to make use of this facility to improve the quality of Internet transmission. In scenarios that end users want high throughput, like user is watching HD movies, parallel multipath transmission can greatly improve throughput. In scenarios that end users have intermittent internet connection on one interface, multipath connection can provide smooth switching between connections.

Current work on multipath is mainly on TCP. In MPTCP, if the user has more than one internet interface, there will be more than one subflow in one TCP connection while each subflow is an independent TCP connection. In this way, the user does not need to re-establish the connection when switching connection.But on ther other hand, MPTCP also has some problems. MPTCP maintains multiple TCP flows. Normally, the number of connections is all the possible composition of all interfaces between the client and server. If clients and the server both have 2 interfaces, it means that there will be 4 subflows for each connection. This can be of high workload for servers that have large number of parallel clients. Also, in multipath TCP, as designed, there will be two levels of sequence number. There will be a mapping between the overall sequence number and independent sequence number of each subflow. If the difference of delay among subflows is huge, MPTCP has to maintain a large size of buffer to deal with out-of-order problem. Because every subflow is a regular TCP connection, TCP slow-start happens whenever switching happens which will pull down the overall performance of MPTCP. Of the most importance, MPTCP can only be used in TCP connection given that there is still large amount of non-TCP traffic on the Internet although TCP traffic is dominating.

Based on all weaknesses of MPTCP and feasibility study of MPIP, we propose our multipath implementation at network layer. Our contribution is three-fold.

1. We propose the overall design and architecture of MPIP. By comparing our design with MPTCP, we see that implementing multipath at network layer has much lighter weight and more straightforward than transportation layer.

2. We implement our design in the latest Linux kernel under Ubuntu system. Also, we evaluate the implementation in different Internet environments. We show that our implementation can match MPTCP in TCP protocol, and also, other protocols like UDP can fit perfectly with multipath IP.

3. For investigation purpose, we combine the implementation of MPTCP and MPIP together to prove multipath feature at both layer. It turns out that this combination can provide

better and more consistent performance over some Internet conditions.

The rest of the paper is organized as follows. Section 2 describes the related work. The design of our implementation is introduced in Section 3. In Section 4, we report the experimental results for our multipath IP design. We conclude the paper with summary and future work in Section 5.

2. RELATED WORK

todo...

3. THE DESIGN

Based on Linux kernel 3.12, the implementation is mainly at the network layer and targets to IP protocol. To make the implementation transparent to users and other layers, we try to keep all kernel modifications inside network layer. Later in the paper we will merge MPTCP and MPIP together, this good modularization makes the two features work together seamlessly. To keep the simplicity of IP protocol, we keep the connectionless feature of IP protocol while maintaining some feedback information of different paths. As will be shown later, we achieve this goal by simply by keeping track of several tables. This can make sure that we don't add excessive overhead to the system.

As a fundamental problem, NAT devices are commonly used in the current Internet. For a single connection, multiple NAT devices can exist on the path. With the existence of NAT devices, many IP addresses are exposed to the Internet as a combination of IP address and port number. So in the following sections, when referring the address of a node, we will use the combination of IP address and port number.

3.1 IP Layer Control Communication

IP protocol is a connectionless protocol. In TCP protocol, the ACK packet is mainly used as the feedback information from the receiver. Then the sender is able to know the real time status of the transmission. But IP protocol doesn't have this built-in feedback loop. To maintain multiple paths at IP layer, we need to add control communication functionality like the ACK packet in TCP at IP layer, then two ends of one connectionless connection can talk to each other. Instead of constructing new control packets between connection ends, we use piggyback technology to implement the feedback message in IP layer. This can avoid excessive control packets and reduce overhead of the MPIP.

For each MPIP enabled packet that goes out of the system, we add an additional control message(CM) data block at the end of user data. Sometimes the size of the packet will exceeds MTU after attaching the data block, in this case, we reduce the amount of user data to fit the CM block.

The content in the control message is shown in Table 1.

Node ID is the globally unique identification of one node. The combination of IP address and port number is not a qualified candidate because during connection switching, IP address can change on the specific NIC. To have a static and consistent node ID, we choose the MAC address of one of NICs to be the unique ID of a node. The value of node ID is initiated when the system starts and keep it unchanging until the system exits. Every time the node sends out a packet, it fills the field of *node ID* with the previously extracted value into the control message.

Local Address List carries all local IP addresses. This list will be used to construct new MPIP paths.

CM Flags notates the functionality of the packet. With different value of *CM Flags*, different action will be taken when the packet is received.

Table 1: Control Message Structure

Control Message
Node ID,
Session ID,
Path ID,
Feedback Path ID,
Packet Timestamp,
Path Delay
Local Address List
CM Flags
Checksum

Checksum is used to verify the validation of the CM data block. This value is assigned by simply adding up the value of all other field in the CM data block. This will be recalculated when received to judge whether a CM data block is attached in this packet. The packet will be treated as a normal packet instead of a MPIP-enabled packet if the checksum verification doesn't get through.

Other fields of the CM block will be explained in following sections.

3.2 MPIP availability handshake

As a new feature in Linux, MPIP needs to be backward compatible. For one specific connection, before enabling MPIP functionality at both end, the two sides need to synchronize with each other to make sure both are MPIP enabled. Locally, every node maintains Table 2 to identify the availability of MPIP at its opposite nodes. Before the handshake finishes, all communication on the connection is normal traffic without MPIP enabled.

Table 2: MPIP availability

IP Address	Port Number	MPIP Availability	Query Count
IP_1	P_1	True	2
IP_2	P_2	False	5

When a node sends out a packet, it checks locally whether the target node is MPIP enabled in Table 2. If not, it copies the current packet. Besides sending out the original packet, the system inserts the CM block into the copied packet with *CM Flags* of $Flags\_Enable$. This value is used for MPIP query. When this packet is received by a MPIP enabled node, the receiver adds the sender's IP address and port number into Table 2 with value of $True$, then sends back the confirmation to the sender with *CM Flags* of $Flags\_Enabled$ at a proper time. For TCP and non-TCP connection, we send out the confirmation packet at different times. For any protocol rather than TCP, we populate a new packet, fill all header fields with the correct information, and insert the CM block at the end, then send back to the sender right away. But for TCP, we add the confirmation request into a waiting list, and piggyback the confirmation when next TCP message is sent out to that specific node. There are two reasons to have this different process.

1. For protocols rather than TCP, like UDP, because they don't have built-in feedback loop, which means that all traffic can be one direction. In this scenario, we can't wait until there are packets available to piggyback the confirmation.

2. For TCP protocol, we don't simply populate a new TCP packet and send back because the sequence numbers of each direction are way different. Through our experiments, some NAT

devices will refuse to transfer this packet if the sequence number messes up for the same TCP connection. To solve this problem, we add the request into a waiting list. When there are TCP packets available to send , we make a copy of the packet and insert the CM block with *CM Flags* of $Flags\_Enabled$. In this case, there will be two consecutive TCP packets with the same sequence number. The NAT devices will simply consider them as retransmissions instead of dropping them.

For both the MPIP query packet and MPIP confirmation packet, we populate new packets based on the original packets by making copies. All copied packets won't be passed to higher layers because they are generated at the network layer and stop there too, they don't mean anything to higher layers. The network layer drops all MPIP packets with *CM Flags* of $Flags\_Enable$ and $Flags\_Enabled$ after processing them.

With the handshake flow above, the smallest number of query packet that will be sent for the whole process is only one. But sometimes because of packet loss or synchronization issues, there can be multiple query packet sent out at both ends. This is not a problem because the design of the system allows receiving more than one query packet and confirmation packet. But on the other hand, for nodes that don't support MPIP, we can't send out query messages forever. In Table 2, the column *Query Count* maintains the number of query messages that have been sent out to relative IP addresses. If the number is larger than a threshold value, it assumes that IP address doesn't support MPIP. Considering that packet loss is rare event in Internet nowadays, we set this value to 5 in our system, this is more than enough to make sure a MPIP enabled node can receive and reply this message with successful transmissions. In a connection with only one-way traffic, if the receiver is MPIP enabled while the sender isn't, then no query packet will be sent out at all.

Generally, for MPIP enabled nodes, they may have more than one IP address. In this situation, multiple synchronization messages will be transmitted for the same node.

3.3 Path Management

Given M and N interfaces at each end of one connection, there are totally $M * N$ possible paths on the connection. In our system, we maintain all available paths through Table 3.

Node ID is used to identify the sender globally. In all packets sent out from this node, the same value will be filled into the field of *node ID*.

Path ID is used to identify one path. It is a locally unique Arab numbers starting from 1. This value is generated only when adding new paths into Table 3.

Because of NAT mapping, the same node can have different IP address and port number that will be seen on the other side at different time or even for different application. That's why column *Session ID* needs to be added into Table 3. When sending out one packet, the final path will only be chosen from the ones that have the correct session id. The detail of session is explained in Section 3.7. Here we just consider a session is the synonym of one connection. As the names show, source IP, source port, destination IP and destination port show the address information of the path. The IP addresses here are the addresses that are seen by the node that maintain the instance of Table 3. This value might be different from local IP addresses because of NAT mapping.

One instance of Table 3 is maintained at each side. Initially, Table 3 is empty. Every time one new combination of IP address and port number is flagged as MPIP enabled in Table 2, it adds new paths to the table with every available local IP address as the source IP address, and the value of Node ID is extracted from Table 4 which is maintained to conveniently map node ID with IP address and port number.

Table 4 contains all working IP addresses and port number of one specific node ID. When paths in Table 3 are obsolete and need to be removed, relative entries in Table 4 will also be removed.

Table 4: Node ID vs IP address and Port

Node ID	IP Address	Port Number
ID_1	IP_{11}	P_{11}
ID_1	IP_{12}	P_{12}
ID_2	IP_{21}	P_{21}
ID_2	IP_{22}	P_{22}

Given M and N NICs at each side, a full mesh will have $M * N$ different paths. Actually, in this mesh,if both M and N are larger than 1, every NIC is reused at least once. Some times it is not necessary to reuse them because of system overhead consideration; In this use case, we can limit the number of available paths for one session to be N, then only the fastest N paths will be used, other paths will be abandoned.

3.4 Feedback Loop

We try to maintain the metrics of paths in Table 3. Packet loss and delay are the most important two characters of one path. Given that packet loss in current Internet has become rare events, we only focus on delay of one path. Furthermore, when choosing a proper path to send out packets, the way back is not considered, so the delay here only means one-way delay. With the addition of CM block to existing packet, we implement IP layer's feedback loop with one-way delay as the feedback variable.

In Table 3, all fields referring to network delay will be filled or calculated by feedback one-way delay. Here we need to go back to Table 1. In Table 1, the fields *Path ID* and *Packet Timestamp* are used to measure network delay. When node A sends out a packet, it chooses a path from Table 3, fill the field *Path ID* with the chosen path ID, and fill the filed *Packet Timestamp* with local system time T_1 in the CM block. After node B receives this packet, it updates records in Table 5. Node B extracts node ID, path ID and timestamp from the CM block. The node ID and path ID are directly used to identify records in Table 5, and node B uses $T_2 - T_1$ as the one way delay from node A to node B where T_2 is the local system time of node B when receiving the packet. Node B checks whether the path that identified by the node ID and path ID already exists in Table 5, if yes, it updates the path's delay with $T_2 - T_1$, otherwise, it adds a new record into Table 5.

Table 5: Path Feedback information

Node ID	Path ID	Path Delay	Feedback Time
ID_{11}	PID_{11}	D_{11}	T_{11}
ID_{12}	PID_{12}	D_{12}	T_{12}
ID_{21}	PID_{21}	D_{21}	T_{21}
ID_{22}	PID_{22}	D_{22}	T_{22}

In practice, the value of path delay calculated here isn't the real delay value because of time difference between node A and node B, it can even be negative. But as we will see later, time difference between A and B doesn't have any influence on our algorithm.

When node B needs to send packet back to node A, it chooses the record with the earliest feedback time in Table 5, it fills the

Table 3: Path information

Node ID	Path ID	Session ID	Src IP	Src Port	Dest IP	Dest Port	Minimum Network Delay	Real-Time Network Delay	Real-Time Queuing Delay	Maximum Queuing Delay	Path Weight	Feedback Time
ID	PID_{11}	SID_1	SIP_1	SP_1	DIP_1	DP_1	D_{min11}	D_{11}	Q_{11}	Q_{max11}	W_{11}	T_{11}
ID	PID_{12}	SID_1	SIP_1	SP_1	DIP_1	DP_2	D_{min12}	D_{12}	Q_{12}	Q_{max12}	W_{12}	T_{12}
ID	PID_{21}	SID_2	SIP_1	SP_2	DIP_1	DP_1	D_{min21}	D_{21}	Q_{21}	Q_{max21}	W_{21}	T_{21}
ID	PID_{22}	SID_2	SIP_1	SP_2	DIP_1	DP_2	D_{min22}	D_{22}	Q_{22}	Q_{max22}	W_{12}	T_{22}

field *Feedback Path ID* and *Path Delay* in the CM block with relative value in Table 5, and updates the column *Feedback Time* with local system time. When node A receives this feedback packet, it extracts the path ID and path delay value, and fills the path delay value into the column *Real-Time Network Delay* in Table 3. To avoid outliers, the value of path delay is calculated by moving average algorithm. At meantime, the column *Feedback Time* is updated with current system time. If one path hasn't received feedback information from the receiver for some threshold value of time, it means this path is literally dead, then the system considers the path to be obsolete and remove the path from Table 3.

In practice, IP addresses of devices can be removed or added dynamically. Especially for mobile device, it can connect to different access points(WiFi hotspot/Cellular Tower) at different time, during this stage, its IP address can be changed, removed or added dynamically. Under this situation, the system supports dynamic addition and removal of paths from Table 3 because of IP address changes. When IP address change happens, the value of *CM Flags* is set to be $Flags\_IP\_Change$ in the CM block. After receiving packets of this flag, the receiver knows that IP address change happened on the sender, it will remove all entries of this connection in all tables, then add the combination of IP address and port number contained in the received packet into all the tables as if that combination is MPIP enabled. Also, the sender does the same reset for this connection. After all these resets, there is only one path left for this connection, all the other paths will be added into the system through the normal process. By doing this path reconstruction, we achieve smooth switching during IP address changes. The higher layers won't know the whole process at all because there is always an active path transmitting data for the application.

3.5 Periodical Heartbeat

For protocols like TCP, during the whole lifetime of the connection, both sides are sending packets to each other at a high frequency, then at both sides, column *Feedback Time* of Table 3 can be updated way before the path becomes obsolete. But there are protocols that don't have this built-in feedback mechanism, like UDP. In some UDP applications, all traffic is one way, there aren't any acknowledgements, which means that the sender can't get feedback information through piggybacked messages. Under this scenario, the sender won't be able to properly add new entries into Table 3, then multipath feature can't be applied at all.

To solve this problem, in our system, we add periodical heartbeat mechanism. At each side, when the node receives packets, the system checks Table 5 for the specific node, if it finds the feedback time of this path is close to the obsolete value, it makes a copy of the received packet, switches the source/destination address information, and sends back the packet with CM block attached. The path to send this heartbeat message will be chosen through the same algorithm as regular IP packet as in Section 3.6. Through the heartbeat message, we effectively maintain the active paths between two nodes, and obsolete paths can be safely removed from Table 3.

All heartbeat packets have a special flags value $Flags\_HB$ in the CM block. These packets will be dropped after being processed at network layer.

3.6 Path Selection

Every time one node needs to send out packets, it chooses the most suitable path from Table 3. Depends on the requirement of throughput and responsiveness, we have different considerations of the standard to choose the target path.

3.6.1 Delay-based Path Selection

In the current Internet, there are many high delay-bandwidth product connections. These connections generally have both high delay and high bandwidth. For applications that want to achieve high throughput, we propose following mechanism to achieve the requirement.

The criterion of choosing the best path that can achieve high throughput is on the column of *Path Weight*. Given certain values of this column, we don't simply choose the path that has the largest path weight which may overuse the path and starve other paths; this will definitely cause terrible fluctuations on this connection. Instead, we choose the path by random number. For one specific path k, the probability $P(k)$ it will be chosen is calculated in Equation 1. By balancing the percentage of packets on each path, system fluctuation can be effectively avoided.

$$P(k) = \frac{W_k}{\sum_{i=1}^{N} W_i} \tag{1}$$

From above, the path weight is the only criterion to choose the target path, so the calculation of path weight W is critical to the performance of the system. Wrong decision can result in catastrophic disaster. In our prototype, by introducing delay-based solution, we calculate the value of W in an incremental pattern.

Generally, there are two ways to notate the character of one path which are loss based and delay based. Nowadays, packet losses in networks are rare events, this makes loss rate estimation is difficult and coarse in accuracy. But delay estimation can be accurate enough with a proper measurement.

For network delay, it consists of following 4 parts.

1. Processing delay. Time routers take to process the packet header
2. Queuing delay. Time the packet spends in routing queues
3. Transmission delay. Time it takes to push the packet onto the link
4. Propagation delay. Time for a signal to reach its destination

Among the four parts above, processing delay, transmission delay and propagation delay are fixed value for one path, they can be treated as constant. But queuing delay changes as the traffic load on the path changes. Figure 1 shows the trend of network delay as of the traffic on the path changes.

We can see that the queue in the routers on the path starts to accumulate at point C, but still, there is no loss. Starting from point

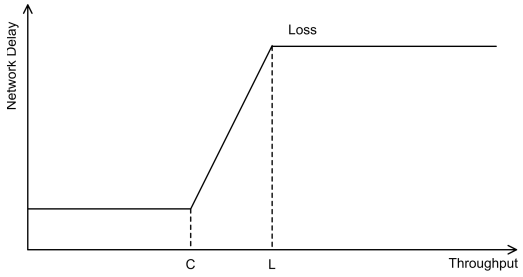


Figure 1: Network delay trend as throughput increases

L, the queue size overflows and packet loss happens. The minimum and maximum network delay for the path can be measured at point C and point L. The difference between the real-time delay and minimum delay can be treated as the approximate queuing delay which represents congestion situation on that path. By observing the real-time queuing delay, we can adjust the weight of each path to properly assign future packets.

In Table 3, except the column *Real-Time Network Delay*, other three delay related columns are calculated through real time delay D.

1. **Minimum Network Delay** D_{min}. Every time one node receives update of network delay, it update this column with the minimum of its current value and the new real-time network delay.

2. **Real-Time Queuing Delay** Q. According to Figure 1, the value of queuing delay is calculated as $Q = D - D_{min}$ which is the difference between real-time network delay and minimum network delay.

3. **Maximum Queuing Delay** Q_{max}. Maximum queuing delay is updated once the real-time queuing delay Q is larger than Q_{max}. When packet loss happens, this value notates the curve section after point L in Figure 1.

During our experiments, we found that calculating the weight of each path independently according to queuing delay can result in high fluctuation. So instead, we couple all the paths together and do micro adjustment to the weight of each path periodically. The detailed mechanism is shown in Algorithm 1.

Algorithm 1 Path Weight Incremental Adjustment.

```
1: Q_avg = (Σ_{i=1}^{N} Q_i) / N;
2: if Q_i ≤ Q_avg then
3:     W_i = W_i + S;
4:     if W_i > 1000 then
5:         W_i = 1000;
6:     end if
7: else
8:     W_i = W_i − S;
9:     if W_i < 1 then
10:        W_i = 1;
11:    end if
12: end if
13: return;
```

In Algorithm 1, N is the number of paths that belongs to one specific connection, Q_i is the queuing delay of path i, W_i is the path weight of path i, and S is the adjustment granularity. Initially, every path has the same path weight of $\frac{1000}{N}$. In each loop, the path weight increases or decreases by S. The maximum value of the weight is 1000. The minimum value is 1 because we try to keep all the paths alive in case the congestion on a bad path has huge relief.

Algorithm 1 is executed periodically, the length of of period is defined as a configurable system variable T. So here, we have two configurable parameters which are the granularity S and the period T. There is a trade-off here. Larger S and shorter T can generate higher fluctuation but faster convergence while smaller S and longer T can result in slow convergence and lower fluctuation. In our system, according to the path weight range (1 1000), we set S to 10 and T to 100 milliseconds. During our evaluation, this configuration can converge fast with low fluctuation.

3.6.2 Customized Path Selection Framework

In practice, maybe high throughput can represent most user requirements, but in some special cases, responsiveness can be of the first priority.

A typical example is Skype[**?**] calls. When we make a Skype call, beside audio calls, we can also have video calls by enabling the camera. For Skype, audio packets and video packets are transmitted independently. In most scenarios, with responsive audio quality, real-time video streaming can be a bonus. But large delay in audio streaming can be a nightmare for a Skype call. In this case, obviously, audio packets and video packets have different priorities. During our study of Skype packets, although video packets can also be small, audio packets generally have substantially shorter length comparing to video packet. This provides a chance to define specific rules for Skype audio packets.

Also, in a TCP connection with multipath, ACK packets are generally very small. On the other hand, delay in ACK packets can trigger TCP congestion control at the sender side. This will result in unnecessary degradation of performance. By sending small ACK packets on the path with lowest delay, unnecessary congestion control can be avoided, and further improve the overall performance of the connection.

To address the requirement above, we enable the users to define their own customized path selection policy based on the destination and length of packets. This customization provides a fundamental framework for more advanced path selection algorithms. A dedicate MPIP routing table is defined in Table 6.

Table 6: MPIP routing table

IP Address	Port Number	Protocol	Start Size	End Size	Routing Priority
$*$	23	TCP	0	200	P_Res
192.168.1.2	80	TCP	200	$*$	P_Tp
192.168.1.3	5221	UDP	0	500	P_Res

When sending a packet, the system checks the destination IP address, port number, protocol and length of the target packet to get relative routing priority from Table 6. Different routing priority has different path selection policy. In our system, only two priorities are supported. $P\_Throughput$ means throughput is the first priority, the delay-based path selection algorithm in Section 3.6.1 will be used. When $P\_Response$ is assigned, it means responsiveness is the first priority; In this scenario, the path that has the lowest delay will be chosen to send out the packet.

In the first row of Table 6, for any TCP connection with destination port 23, if the packet length is smaller than 200 bytes, the path with the lowest delay will be chosen, otherwise, delay-based algorithm is used. The second row actually is useless be-

cause all packets will use the delay-based algorithm even this row doesn't exist. The third row specifies that UDP packets sent to 192.168.1.3's 5001 port will be assigned to the lowest delay path, otherwise, delay-based algorithm is used.

The same as regular routing table, the content of the MPIP routing table is configurable by users. In MPTCP, because all congestion control algorithms inherit from traditional TCP, it can't make the best of multipath. On the other hand, MPIP maintains paths in a customized pattern, it is more feasible to have application specific routing decisions. As we mentioned above, we only provide a basic framework for customized routing in MPIP. It only has limited functionalities. For example, it only specifies applications by port number. For some applications like P2P software, they have arbitrary port numbers, we can't locate the specific application in this case. There can also be other scenarios that our prototype doesn't work. But based on this framework, a more powerful and smarter routing decision mechanism can be completed.

3.7 Session Management

For one specific connection, it can be identified by a socket at each end. Each socket pair is described by a unique 4-tuple consisting of source and destination IP addresses and port numbers of local and remote nodes. In our prototype, we call a connection's unique socket a session. To maintain all sessions, each MPIP enabled node maintains an instance of Table 7.

3.7.1 Addition/Removal of Sessions

Session ID is the unique identity of one session. Unlike path ID in Table 3 which only requires to be unique locally, session ID needs to be unique on both ends of one connections which means that the session ID needs to be the same on both ends for the same session. But from Table 7, we can see that different nodes can have the same session ID. Same as path ID, session ID is Arab numbers starting from 1. This value is generated only when adding new sessions into Table 7.

After the MPIP availability handshake has been successfully completed, when sending out a packet, the sender checks Table 7 to see whether a proper session entry has been generated. If not, one new record will be added. We extract the IP addresses, port numbers and protocol from packet headers, and gets the destination node ID from Table 4, then it generates a new session ID and adds one new entry into Table 7. After this, the new session ID will be filled into *Session ID* in the CM block whenever packets are sent out on this connection. The receiver extracts the session ID and inserts one entry into its own Table 7. This will make sure that for the same session, both sides of the connection have the same session id because only one side generates it.

Besides the session ID, all IP addresses and port numbers can be different in Tabel 7 of both sides because of NAT devices, but this doesn't have a problem because the IP addresses and port numbers only need to be the values that are seen locally.

Removal of sessions is decided by expiration. At each node, every time it sends or receives a non-heartbeat packet, it goes to Table 7 to get the session ID to fill the control message, meantime, it updates the column *Update Time*. For an active session, this time stamp should be updated frequently. If the timestamp expires a threshold value, the session is considered to be obsolete and removed from Table 7. In our system, this threshold value is set to 120 seconds.

For each connection, the system internally maintains a socket pair with source/destination IP addresses and port numbers, the system maintained socket information is the same as the pair maintained in Table 7. The socket information for a session entries won't be modified after they have been added into Table 7 even the IP address that initiates the session doesn't exist any more. As will be shown later, the socket information in Table 7 is used to communicate with higher layer to guarantee seamless connection switching. If these information is modified, higher layers will notify the mismatch between the system maintained socket amd MPIP maintained socket which finally causes failure of seamless connection switching.

Once session entry has been inserted into Table 7 at both side, different paths will be added into Table 3 and available to be chosen to transmit future packets that belong to this session according to path selection algorithms.

3.7.2 Workflow of Sending/Receiving Packets

For one regular packet, when being sent, it goes through application layer, transportation layer, then arrives at network layer where IP protocol resides in. Before the packet arrives at network layer, socket information inside the packet is the one that stores in a system maintained table. In regular scenarios, the system finds a proper interface to push out the packet according to the destination IP address and routing table. The opposite flow is executed for inbound packets. For a MPIP enabled system, all the other operations remain the same except that of network layer.

For outbound packets, when the packet arrives at network layer from transportation layer, the system looks into Table 7, and finds the session entry that matches socket information in the original packet, extracts the session ID to fill into the field *Session ID* in the CM block. To choose the proper path to send out the packet, the first thing is to locate which paths in Table 3 are eligible for this connection. Given the destination IP address and port number in the socket information, we can find the *Node ID* in Table 4, then in Table 3, all entries that have the correct node ID and session ID are eligible. Among all these paths, the most suitable path will be chosen out through the mechanism introduced in Section 3.3.

The chosen path's path ID will be assigned to the field *Path ID* in the CM block for delay measurement at the receiver as narrated above. Meantime, we modify the source and destination IP address/port number in the IP header and transportation layer header with the chosen path's source and destination IP address/port number. Then we route the packet with the new IP header information to a proper interface.

For inbound packets, when receiving a packet, the receiver extract the node ID and session ID in the CM block, with these two parameters, the receiver locates the original socket pair in Table 7, and modify the IP header and transportation layer headers with source and destination IP address/port number from Table 7. Fields related to path measurement will be processed as explained int Section 3.3. Now the packet is back to its original shape that can be recognized by higher layers, it is ready to be pushed up.

3.8 Multipath TCP Transmission

Through our experiments and previous studies[1], NAT devices have a lot of interferences to end-to-end connections, especially for TCP packets. The most straightforward limitation for TCP packets is that many NAT devices will drop TCP packets that don't have a connection related to them. MPTCP doesn't have this problem each subflow in MPTCP is a regular TCP connection. But in MPIP, if we transmit TCP packets on a path rather than the original one, NAT devices on the path will probably drop these packets before they arrive at the destination. To solve this problem, in our MPIP implementation, we provide two options to solve this problem.

3.8.1 Fake TCP connection

Table 7: Session Information

Node ID	Session ID	Source IP	Source Port	Destination IP	Destination Port	Update Time	Protocol	Next Sequence No
ID_1	SID_1	SIP_1	$SPORT_1$	DIP_1	$DPORT_1$	T_1	TCP	S_1
ID_1	SID_2	SIP_1	$SPORT_2$	DIP_1	$DPORT_2$	T_2	UDP	0
ID_2	SID_1	SIP_2	$SPORT_3$	DIP_2	$DPORT_3$	T_3	TCP	S_2
ID_2	SID_2	SIP_2	$SPORT_4$	DIP_2	$DPORT_4$	T_4	UDP	0

In NAT devices that drop TCP packets without relative connection information, we cheat them by constructing fake TCP connections. The construction of TCP contains three-way handshakes. Instead of constructing real TCP connections, we implement a simple three-way handshake at network layer which is similar as TCP handshake, but this connection information won't go up to TCP layer. All handshake packets have *CM Flags* value of $Flags\_Hs$, these packets are dropped after being processed by MPIP.

As shown in Table 1, the field *Local Address List* carries all local IP addresses. Also, the node that initiates the connection is considered as the client. When the client receives the IP address list of the server, it extracts its own IP address list, then sends out a SYN packet through each possible path to the server except the original one which is the one that was used to initiate the connection. When the server receives this SYN packet, it replies with a SYN-ACK packet through the same path. After the client sends out the final ACK packet to the server, the three-way handshake for our fake TCP connection is completed successfully. After this, the path can be used to transmit TCP packets without being dropped.

In one specific TCP connection, we assume that the server has at least one public IP address for initiating the connection. But it is possible that other IP addresses on the server are not public, then the SYN packet will never arrive at that address, in this case, this interface will not be used at all.

3.8.2 UDP wrapper

The other option we provide for multipath TCP traffic is UDP wrapper. Because there is no connection information in an UDP packet, during our experiments, most NAT devices don't have limitation on regular UDP traffic. We can make use of this feature to wrap our TCP packet with a UDP header to pass the NAT devices and unwrap it when received.

At the sender side, every time the network layer gets a TCP packet from transport layer, the system chooses a path to send the packet out as shown in Section 3.3. If the chosen path isn't the original path, we wrap the user data and TCP header into an UDP packet.

Also, if some addresses of the server in the IP address list are not public, the packet won't be received by the server and the path at the client will expire soon because of no feedback for that path and this path will be deleted.

If UDP wrapper is used to transmit TCP traffic, when receiving one packet, we are capable to know this UDP packet is a wrapper for a TCP packet instead of a regular UDP packet by checking the column *Protocol* in Table 7. After removing the UDP wrapper, socket information will be extracted from Table 7 and filled into the TCP and IP header.

3.8.3 Dealing with out of order TCP traffic

In modern devices, different interfaces can have totally different delay behaviour even if we connect them to the same service. Generally, cellular interface has much larger delay than WiFi; For PC devices, wired connection generally has smaller delay than WiFi interface. Under this situation, packets can be out of order frequently during the life time of a connection. This is not a problem for protocols like UDP, but for TCP, as shown below, packet out-of-order can result in catastrophic disaster for the overall performance.

In TCP Congestion Avoidance algorithm, a retransmission timer expiring or the reception of duplicate ACKs can implicitly signal the sender that a network congestion situation is occurring. The sender immediately sets its transmission window to one half of the current size. When a duplicate ACK is received, the sender doesn't know if it is because a TCP packet was lost or simply that a packet was delayed and received out of order at the receiver. If the receiver can re-order TCP packet, it shouldn't be long before the receiver sends the latest expected acknowledgement. Typically no more than one or two duplicate ACKs should be received when simple out of order conditions exist. If however more than two duplicate ACKs are received by the sender, it is a strong indication that at least one packet has been lost, the sender does not even wait for a retransmission timer to expire before retransmitting the segment and enter control avoidance stage, the sender's transmission window is cut off by one-half. This is the fast retransmit algorithm.

Considering multipath scenario, if the delay behaviour difference among different paths is not trivial, we can expect a lot of out-of-order packets. This will result in many fast retransmit events, even TCP can go back to slow start stage. Given that in modern Internet, real packet loss has become rare event, most of transmission window cut-off events are unnecessary at all. In our prototype, to solve the heterogeneous delay performance of different paths, we make specific process of TCP out-of-order packets.

For each session in Table 7, if it is TCP protocol, MPIP maintains a buffer B to store all out-of-order packets and the next expecting sequence number S. Every time one node receives a TCP packet, it takes out the sequence number, if it equals with the current expecting sequence number of that session, the packet is pushed up to transportation layer immediately, and also, the buffered packets will be checked to see whether there are qualified packet for delivery. If not, the packet will be buffered into B for future delivery. Every time one packet is pushed up to transportation layer, the next expecting sequence number S of this session is updated through Equation 2 where L is size of the whole packet, H_{ip} is IP header size, and H_{tcp} is TCP header size. TCP sequence number ranges from 0 to $2^{32} - 1$, Equation 2 can overflow S. But the data type of sequence number is unsigned integral, the number will loop back to 0 automatically after that.

$$S(k) = S(k-1) + (L - H_{ip} - H_{tcp}) \tag{2}$$

For the data structure that holds the TCP packets, a proper choice is to implement it as a binary tree, but according to our observations, most out-of-order packets come in an incremental order, and they are held in the buffer because of one late packet, when that late packet arrives, all the held packets will be pushed up. For this reason, we simply implement this structure a sorted list, every out-of-order packet will be inserted into this list in ascending order

according to the sequence number. The time complexity to insert this packet is almost $O(1)$ because most out-of-order packets arrive in an incremental order as mentioned. When the expecting packet arrives, then all the waiting packets can be cleared from this list with a simple loop. This reduces code complexity greatly without sacrificing any performance.

During our experiments, it happens that one specific packet can be late for a long time, like if the packet is lost. In this situation, holding all subsequent packets will halt the whole session because TCP layer will assume that all packets are lost, this will result in catastrophe for the connection. To address this problem, we set up the maximum size of the buffer, all the packets in the buffer will be forcefully pushed up once it is full. In our prototype, we set this maximum size to 10.

3.8.4 MPIP and MPTCP work together

As the first implementation of multipath, MPTCP gains huge attractions in research. During the development of our prototype, we try to do side-by-side comparison between MPTCP and MPIP when referring to TCP connection. Besides this, for investigation purpose, we merge them together to see how the system works. Because MPTCP is implemented at transportation layer, and MPIP is implemented at network layer, when trying to merge the two together, we found that the work can be done easily because of good modularization of the kernel source.

Assume that there are 2 NICs at each end, MPTCP will have 4 subflows for the session while each subflow is an independent TCP connection. MPIP will have four paths for all these four subflows which means that there will be totally 16 paths for one MPTCP session in this case. We will see how this overlapped feature works in Section 4.

4. PERFORMANCE EVALUATION

To evaluate the performance of our proposed system, we implemented our multipath IP in Linux kernel 3.12.1 under Ubuntu system and install the prototype on two desktops. Both desktops are connected directly to one router without any middle-box. Each desktop has two $100Mbps$ NICs which means that there are totally 4 paths and the capacity is $200MBps$ between the two nodes. We use Netem[**?**] to throttle the connection to evaluate our prototype under multiple scenarios. Wireless connection is also considered in our evaluation.

Besides the controlled lab experiments, we also evaluate MPIP on the Internet to verify NAT immunization and system robustness. We connect our client to a MPIP enabled node in Emulab[**?**] located in Utah . According to our test, the capacity between our client and the server in Emulab is about $5Mbps$, so we limit the bandwidth of both NICs on the client to be $2Mbps$. Because there is only one NIC that connects to Internet on the Emulab node, there are only 2 paths in this case.

In all experiments, we try to keep the configuration of each node unchanged after installation. We don't do any special configuration to the system, neither we do any optimization to squeeze out all possible throughput. Except specific experiments that can only be applied to MPIP like UDP experiment and customization MPIP routing, we try to do side-by-side comparison with MPTCP for TCP connections. We will figure out how these two features work independently and together as stated in Section **??**;

For all throughput related experiments, we use iperf3 to generate traffic between the client to the server.

4.1 Clock Offset

During our experiments, we found that the clock of each node has some small difference. In our experiment configuration, the clock of the server is slightly faster than the client. Even this error is very small, we still see the difference in a long experiment. Certainly, nowadays, most computers have NTP enabled and the system's local time synchronize with time server periodically, but we still think that this difference is worth to be shown here.

On our experiment plat, we turn off NTP on both nodes, do a TCP transmission with iperf3 for one day with consistent traffic. Inside MPIP, we record the one-way delay of each packet from the client to server. Because the traffic load is consistent, queuing delay roughly remains the same. But as shown in Figure 2(a), because of the clock offset between the two nodes, the trend of queuing delay exposes an linearly increasing curve even the trend is very slow. In Figure 2(a), we record the queuing delay every one minute. For the whole day(1400 minutes), we can see that the clock offset is about 350 milliseconds which means that the server's clock runs one millisecond fast than the client for each four minutes. We will be able to see this trend again in following results.

4.2 TCP Evaluation

As the dominating traffic, TCP plays a critical role in today's Internet. We try to evaluate the performance of MPIP over TCP connections in multiple network configurations.

4.2.1 TCP Out-of-order Process

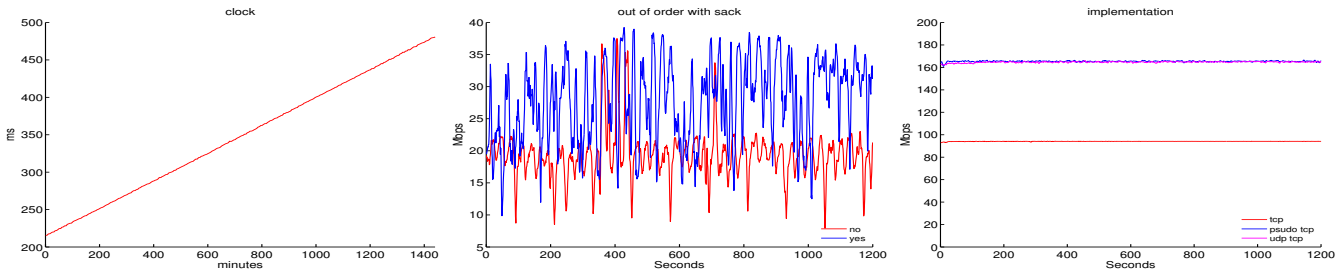
In Section 4.2.1, we explained why out-of-order is a problem that must be dealt with in multipath implementations, and we also proposed our solution to this problem. To verify our proposition, we replace one NIC card on the client with a wireless interface. In our experiment plat, the RTT of one path with two wired NICs is about 0.1ms while paths with one the wireless NIC is about 0.5ms which will generate enough out-of-order packets. Also, to make sure that there are heavy load of packets to be assigned to the wireless NIC card, instead of using the standard path selection algorithm in Section 3.6, we fix the weight of all the four paths to be the same. Then we make sure that 50% of outbound packets will be assigned to the wireless NIC. With this configuration, we do a regular TCP transmission that lasts for 20 minutes with out-of-order process enabled and disabled respectively. The result is shown in Figure 2(b).

With the same configuration, Figure 2(b) shows the improvement brought by the out-of-order process. The average throughput is XXMbps and XXMbps with/out out-of-order process respectively. The improvement maybe trivial if the delay on all the paths is the same because most packets will arrive at the receiver in the order of being sent out. But for multipath connections, it can be very often that each path goes through a totally different route, that is where out-of-order happens most. In all following experiments, we enable out-of-order process by default.

4.2.2 TCP Throughput Enhancement

As we mentioned in Section 4.2, there are two different implementation of multipath TCP in our system to solve NAT problem which are fake TCP connection and UDP wrapper. We do a specific experiment to evaluate the performance of each approach as shown in Figure 2(c). With regular TCP, the average throughput we can achieve is about 92Mbps. For MPIP, with either fake TCP or UDP wrapper, we both achieve an average throughput of about 165Mbps. This result shows that both implementations have roughly the same performance. In all following experiments, we use fake TCP by default, but UDP is still kept as an option for users.

Now we start to do side-by-side comparison between MPIP and MPTCP for TCP traffic. Table 8 shows the average throughput



(a) Clock offset (b) Out-of-order process (c) Fake TCP vs UDP Wrapper

Figure 2

Table 8: TCP Throughput Comparison

	MPTCP (Mbps)	MPIP (Mbps)	MPIP and MPTCP (Mbps)
No Limit	129.5	171.2	164.6
Bandwidth Limit	112.6	113.4	125.6
Delay Limit	129.7	182.0	165.0
Wireless	97.9	89.2	107.5

comparison results for multiple configurations.

We first do the transmission without any throttles which means the capacity of the connection is 200Mbps. We can see that MPIP achieves the highest throughput which is 171Mbps, MPTCP only gets 129.5Mbps. When we combine MPIP and MPTCP together as stated in Section **??**, we get a throughput of 164.6Mbps.

When we limit the outbound bandwidth of one NIC card on the client to 40Mbps, we get the result of the second row of Table 8. In this scenario, the capacity of the connection is 140Mbps. In this case, MPIP and MPTCP get roughly the same throughput, but when they work together, the highest throughput is achieved.

By replacing one NIC on the client to a wireless interface, we get the fourth row in Table 8. In this case, MPTCP achieves higher throughput than MPIP, but still, when they work together, the highest throughput is achieved.

Figure 5 shows the result of throughput comparison of MPIP, MPTCP and MPIP/MPTCP work together. We can see the throughput enhancement brought by MPIP from Figure 3(a). In our experiment plat, without any throughput squeezing configuration, MPTCP achieve an average throughput of 135Mbps, MPIP achieves an average throughput of 170Mbps, when MPIP and MPTCP work together,

4.3 Out-of-order evaluation for TCP

To evaluation the effect of out-of-order process in our prototype, we design following experiment. We replace one NIC with wireless NIC, then there will still be 4 paths. For all the paths, we assign the same path weight which means that every path is assigned the same amount of packets. In this scenario, 50% percent of packets will be sent throughput the wireless NIC. The difference between propagation delay of the wireless path and wired path is large enough to generate a substantial number of out-of-order packets. By enabling and disabling out-of-order functionality, we get Figure 2(b).

4.4 Skype voice call improvement

In Section 3.6.2, we mentioned that Skype can benefit from our customized routing for responsiveness. Skype calls has higher requirement on the responsiveness of audio packets. According to our experiments, Skype audio packets are generally less than 200 bytes. In Table 6, for packets that is smaller than 200 bytes, responsiveness consideration will be the first priority.

To show how this rule influence Skype audio streaming, we change the delay of NIC cards using the netem package in Ubuntu. On one side of the Skype call, we have two NIC cards as stated above. We ran the experiment for 360 seconds with 3 stages. In the first 120 seconds, there is no manually added delay on each NIC card, we add additional $120ms$ delay to the first NIC card in the following 120 seconds, finally for the last 120 seconds, we add $120ms$ delay to the second NIC card. We did our Skype audio experiment without video content while most packets on the connection are audio packets except some control packets. Figure **??** shows the packets routing adaptation as the delay value changes. By this adaptation, we can make sure that the audio streaming will always choose the path that has lower delay to keep the best user experience even other paths have the same throughput.

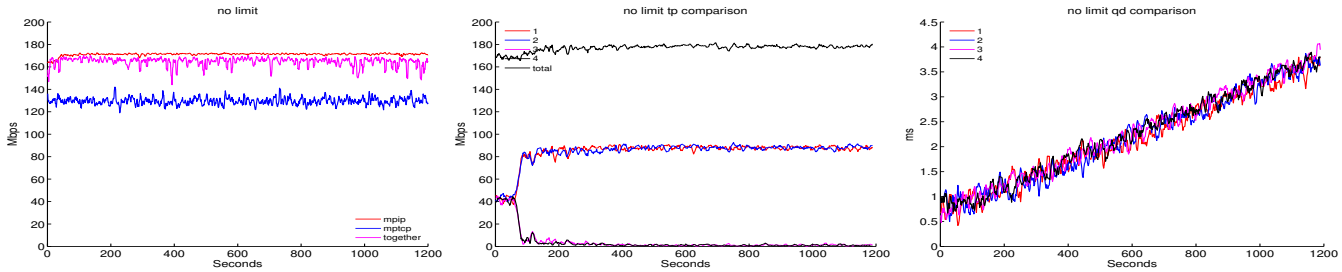
4.5 Smooth connection switch

In Figure **??**, we verify that smooth switch between different NIC cards works perfectly over our MPIP implementation by doing an IPERF TCP experiment. We do a side-by-side comparison between MPIP and MPTCP. We also divide the experiment into 3 sections with 120 seconds for each section. On the client side of the connection, there are two NIC cards. In the first 120 seconds, both NIC cards work synchronously, then we disable one of them for 120 seconds, and during the last 120 seconds, we enable back the NIC card. The result shows that our MPIP system can follow this on/off process perfectly with stable throughput. For MPTCP, as shown in previous experiments, MPTCP has higher fluctuation than MPIP even MPTCP can achieve higher throughput than MPIP. This happens because in MPTCP, there are more than one TCP connections (4 in this experiment), and each connection has its own congestion window. Although congestion control in MPTCP is coupled among different connections, fluctuation have more chances to happen with 4 relatively independent congestion windows. But in MPIP, one one TCP connection is constructed for each session which means that there is only one congestion windows. In this case, the throughput is more consistent than that of MPTCP.

5. CONCLUSIONS AND FUTURE WORK

6. REFERENCES

[1] C. Raiciu, C. Paasch, S. Barré, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How hard can it be? designing and implementing a deployable multipath tcp," in *USENIX Symposium of Networked Systems Design and Implementation (NSDI'12), San Jose (CA)*, 2012.



(a) Throughput Comparison (b) Path throughput Comparison for MPIP (c) Path Queuing Delay Comparison for MPIP

Figure 3: Side-by-side comparison without any limitation

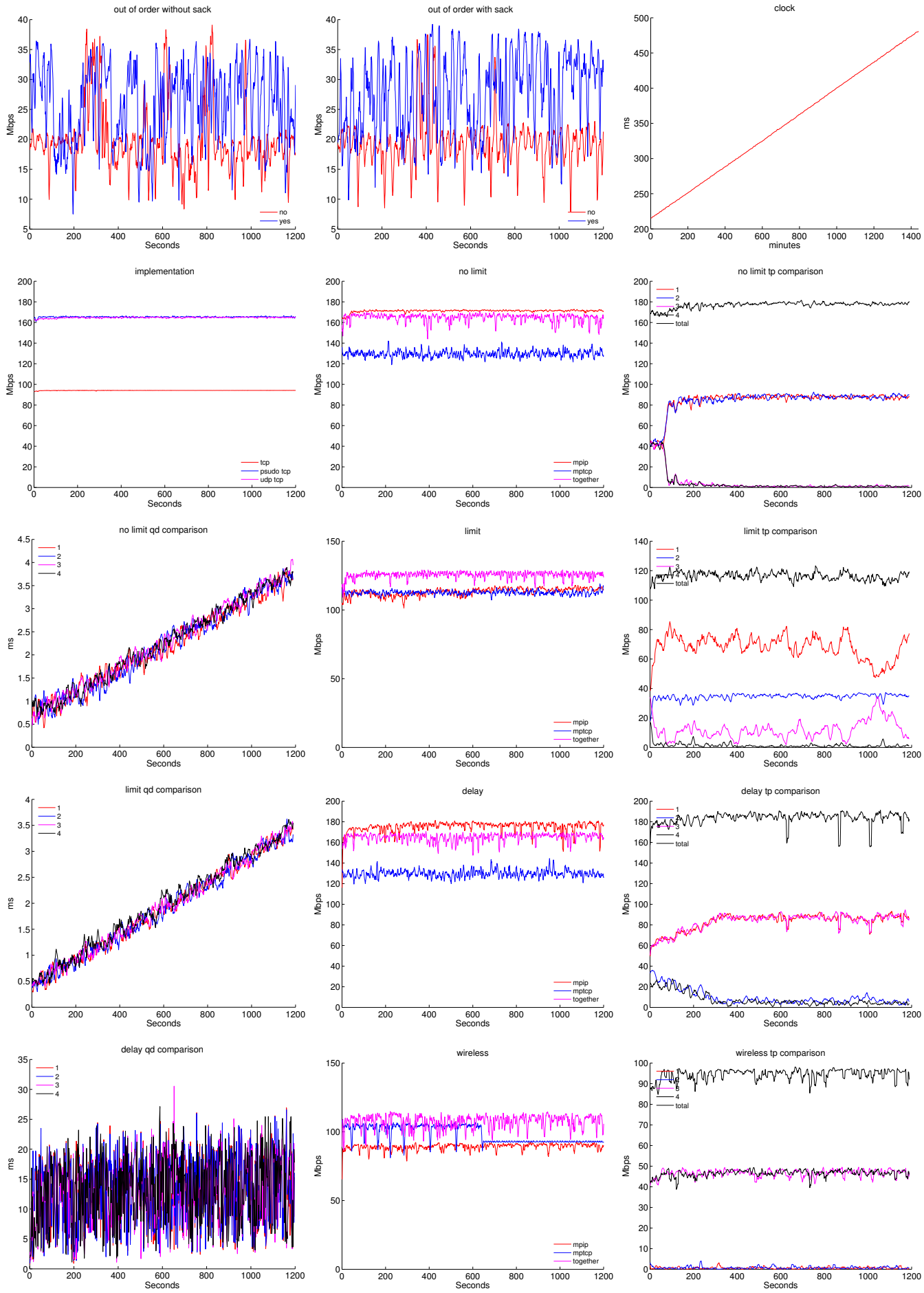


Figure 4: Side-by-side comparison for no limit

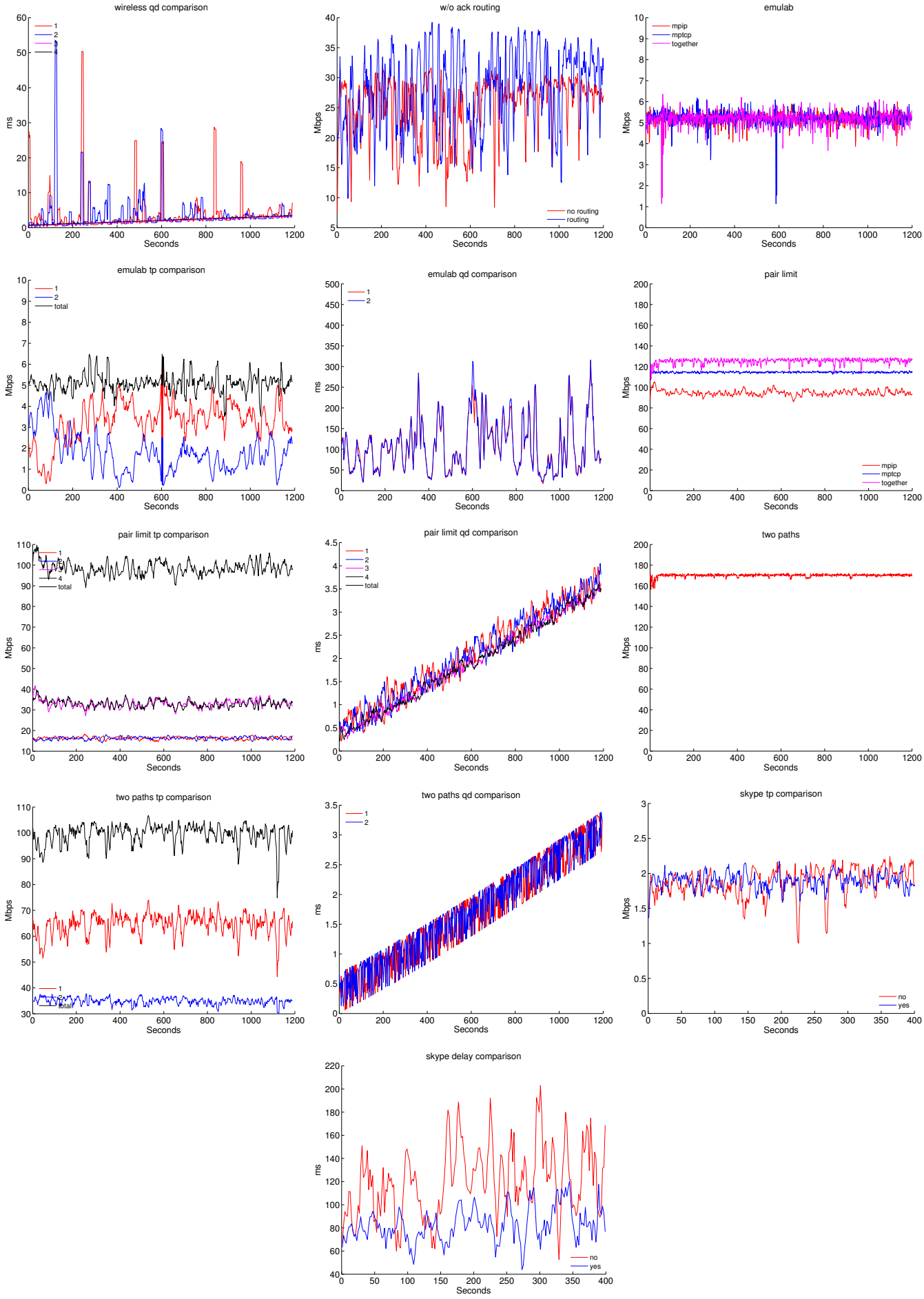


Figure 5: Side-by-side comparison for no limit