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 be connected 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 research work on multipath TCP 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 implementing multipath feature at network layer, almost all traffic on Internet can enjoy the convenience and performance improvement introduced by multipath. Also, because of the design and internal characters of multipath TCP and multipath IP, we can get a lighter weight implementation of multipath at network layer. We implement our proposition in the latest Linux kernel and evaluate its performance under multiple Internet scenarios.

1. INTRODUCTION

Multipath has become available in recent years. Also, IETF proposed RFC 6182 specifically for multipath TCP development in 2011. By introducing multipath between both ends for one connection, not only higher throughput can be achieve, 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 (3G, WiFi), it is possible to make use of this facility to improve internet transmission. In use cases that end users want high throughput, like user is watching HD movies, parallel multipath transmission can greatly improve throughput. In use cases that end users have intermittent internet connection on one interface, multipath connection can provide smooth switching between connections which improves user experience.

Current work on multipath is mainly on TCP. In multipath TCP, if the user has more than one internet interface, there will be more than one sub TCP sub-flow in one TCP connection. In this way, the user does not need to re-establish the connection again when switching connection. The most popular implementation is from [1], which maintains multiple sub-flows for a single TCP connection. But in multipath TCP also has some problems.

1. In multipath TCP, between the client and server, there will be multiple TCP connections. 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 sub-flows for each connection. This can be a very high workload for the server if the server has large number of parallel clients.

2. In multipath TCP, each sub-flow has its own congestion window, and also, to guarantee fairness, the congestion windows of all sub-flows are coupled. This makes the mechanism too complicated.

3. 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 sub-flow. Also, this will make things complicated. Complicated things are always vulnerable.

4. In multipath TCP, when switching connection, TCP slow-start will be done which may result in delays.

5. multipath TCP can only be used in TCP connection. For other transport layer protocol, multipath can not be used. Although TCP traffic is dominating the Internet nowadays, there are studies showing that other protocols like UDP still play important roles. For some specific applications, TCP is not the best choice.

Based on this, we propose our multipath implementation at IP layer.

Implementing multipath functionality at IP layer has following pros.

1. IP layer is relatively simpler than transport layer because it is connectionless. We do not need to deal with the complicated

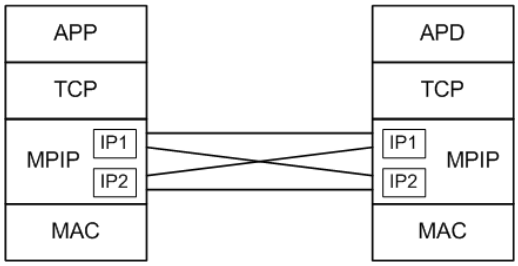


Figure 1: Overall design of MPIP

congestion window and flow control which resides in TCP protocol. Instead, we only do the implementation at IP layer and the implementation is also connectionless. This is totally transparent to other layers.

2. More straightforward to implement multipath at IP layer since multipath is in fact multiple IP addresses.

3. MPTCP can only be used for TCP connection while MPIP is eligible for all protocols above IP layer.

Our contribution is four-fold.

1. We propose the overall design and architecture of multipath IP transmission. By comparing our design with multipath TCP, we see that implementing multipath at IP layer has lighter weight than at TCP layer because of the internal simple character of IP layer.

2. We implement our design in the latest Linux kernel.

3. We evaluate our implementation in different Internet environments. We show that our implementation can match multipath TCP in TCP protocol, and also, other protocols like UDP can fit perfectly with multipath IP.

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. DESIGN OF MPIP

Based on Linux kernel 3.12, we implement our prototype in Ubuntu. Most work resides in the IP layer with some compatibility modification in other modules. Also, the addition of multipath IP feature is transparent to users, a.k.a, all user's applications won't be modified.

The framework of MPIP is shown in Figure 1.

3.1 NAT consideration

For each connection, probably there can be multiple NAT devices on the path. In this scenario, we can't only use the exposed IP address as the identification one node, instead, the combination of IP address and port number is unique for each node. So in the following sections, when referring the address of a node, we will use the combination of IP address and port number.

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

3.2 IP Layer Control Communication

As stated before, most implementation is done at IP layer. In the IP layer, everything is like UDP, there is no feedback information at IP layer. The kernel just sends out the packets through a proper path, then all the other things will be dealt with by upper layer. Like in TCP, reliable transmission is guaranteed by its own control protocol.

To support additional multipath feature at IP layer, we need to introduce feedback message at IP layer to enable the peers to talk with each other. Not like in TCP, which has specific control packets like ACK packets to support the protocol, 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 data block at the end of user data. As shown in Figure **??**.

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

Node ID is the globally unique identification of one node. Because IP address can change at different times, we use the MAC address of one of NICs at the local node. When the system starts, we initiate node ID's value with one of the node's NIC card's MAC address and keep it unchanged 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 has been received.

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.

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

3.3 MPIP availability handshake

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

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

Table 2: MPIP availability

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

packet is received by a MPIP enabled node, the receiver adds the sender's IP address and port number into Table 2, then sends back the confirmation to the sender with *CM Flags* of $Flags\_Enabled$.

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 do 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 of only one direction. In this situation, we can't wait until there are packets available to send back

2. For the protocol of TCP, we don't simply populate a new TCP packet and send back because the sequence numbers of each direction are different. Through our experiments, some NAT devices will refuse to transfer this packet if the sequence number messes up for the same TCP connection. We can manage to maintain the right sequence numbers to fill them into the confirmation packet, but by considering the whole design of our system, we choose to add the request into a waiting list, when there is TCP packet available to send back, 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 that go through the path with the same sequence number. The NAT devices will simply consider that they are retransmission 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 manually copied packets won't be passed to the upper layers because they are generated at the network layer and stop there too, they don't mean anything to upper 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. For nodes that don't support MPIP, we can't send out query messages forever. In Table 2, the column *Query Count* maintains how many query messages 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 transmission. In a connection will 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 can have more than one IP address. In this situation, multiple synchronization messages will be transmitted for the same node.

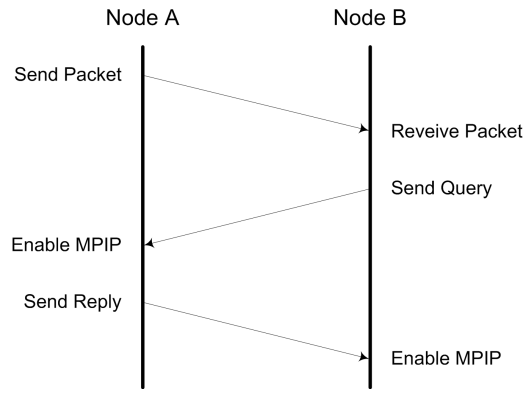


Figure 2: MPIP availability synchronization

3.4 Multipath TCP Transmission

Through our experiments and previous studies[1], NAT devices have a lot of interference 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. This means that if we transmit TCP traffic on a path rather than the original one, the NAT devices on the path will probably drop these packets before they arrive at the destination. There is no such problem in Multipath TCP[1] because MPTCP has a totally independent TCP connection on each path which will pass this category of NAT devices successfully. In our MPIP implementation, we provide two options to solve this problem.

3.4.1 Fake TCP connection

In NAT devices that drop TCP packets without relative connection information, we cheat them by constructing fake TCP connections on relative paths. 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 it only happens at network layer.

As shown in Table 1, the field *Local Address List* carries all local IP addresses. 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, then this path will be used to transmit MPIP TCP packet without being dropped. The whole process only happens at network layer, all packets for the three-way handshake have *CM Flags* value of 5, they will be dropped after being processed at network layer. By having fake TCP connections, we can make use of multiple feature for TCP traffic without the overhead of managing multiple TCP connections as in MPTCP.

If some addresses of the server in the address list are not public address, then the SYN packet will never arrive to that address, in this case, this interface will not be used at all.

3.4.2 UDP wrapper

The other option we provide for multipath TCP traffic is UDP wrapper. Because there is no connection information in an UDP packet, 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 packet to pass the NAT devices and unwrap it when received.

As shown in Figure **??**, 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 3.5. 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 address, the packet may go to some other node with the private IP address, in this case, the *node id* value verification will fail, then this packet will be dropped by that node. At the sender side, the path will expire soon because of no feedback for that path and this path will be deleted.

3.5 Path Management

As shown in Figure 1, given M and N interfaces at each end, there are totally $M * N$ possible paths on the connection. On each end of one connection, the same number of paths with the opposite direction will be maintained. 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 maintained at each end, we set it as 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 the 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 3.9. 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. Because of the existence of NAT devices, IP address is not enough to identify the address information. Also, the IP addresses can be different on the two nodes even if they represent the same physical path because NAT devices change IP addresses. For the same pair of nodes, one instance of Table 3 is maintained at each side, the IP addresses here are the addresses that are seen by the node that maintain the instance of Table 3.

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, 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.

Structure of Table 4 is very simple, it 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}

3.6 Feedback Loop

Unlike multipath TCP, which has built-in congestion control algorithms, in the Network layer, to be able to detect the capability of each path, we need some kind of feedback information like in TCP. But we don't need the same complicated congestion control algorithms. Instead, we just roughly estimate capability of each path to make a choice.

In Table 3, all fields referring to network delay will be filled or calculated by feedback information. In Table 1, the fields *Path ID* and *Packet Timestamp* are used to measure network delay. When a node A sends out a packet, it chooses a path from Table 3, fill the field *Path ID* in control message with the chosen path ID, also, get the current time with system's jiffies value and fill the filed *Packet Timestamp* in control message. After the receiver node B receives this packet, it updates records in Table 5. The receiver extracts node ID path ID and timestamp from the control message. The node ID and path ID are directly used to identify records in Table 5, for the timestamp T_1, the receiver calculate the timestamp T_2 with its local jiffies value and uses $T_2 - T_1$ as the one way delay from node A to node B. 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}	J_{11}
ID_{12}	PID_{12}	D_{12}	J_{12}
ID_{21}	PID_{21}	D_{21}	J_{21}
ID_{22}	PID_{22}	D_{22}	J_{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, even, it can 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 field *Feedback Path ID* and *Path Delay* of the control message with relative value in Table 5, and updates the column *Feedback Time* with system's current jiffies value. When node A receives this packet, it extracts the path ID and path delay value in the control message, 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 the current local jiffies value.

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 addition and removal of paths from Table 3.

There are two scenarios that new paths will be added into Table 3. The first scenario is when new packets are received, new paths will be added as narrated above. The second scenario is when new IP addresses are assigned to the device, the system will check Table 3 to added new path entries for all destination IP addresses. Every time the system sends out packets, it checks the value of the column *Feedback Time* in Table 3, if the the latest update time for the path has exceeds a threshold value, then the system considers the path to be obsolete and remove the path from Table 3. Also, if there are changes of IP addresses locally, like up/down of NICs and change of IP address, there will also be addition and removal

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}

of paths.

If there is any changes of local IP addresses, when sending out packets, the value of *CM Flags* in CM block will be 1 to notify the other node to make changes to relative tables.

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

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 3, 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 3.

3.7 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 control messages. Under this situation, 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 message on one connection. At each side, when the node receives packets, the system checks Table 5 for the specific node, if it finds that the entry with the earliest feedback time 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 3.8. Through the heartbeat message, we can effectively maintain the active paths between two nodes, and safely remove obsolete paths 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.

3.8 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.

3.8.1 Delay-based Path Selection

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

As mentioned above, the candidate paths should have the correction session id for the target connection. The criterion of choosing the best path is on the column of *Path Weight*. Given certain values of the is column for each path, we don't simply choose the path that has the largest path weight which may overuse the path and starve other paths at meantime, instead, we choose the path by random number. For one specific path k, the probability P 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 most suitable 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.

When the system tries to send a packet, it will have to choose the most suitable one among all the available paths. To achieve this goal, we need to have some parameters to notate the capability of each path. Generally, there are two ways to notate the character of one path, the first is loss based and the second is 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 4 parts above, processing delay, transmission delay and propagation delay are fixed value for one path, they can be treated as constant C. But for queuing delay Q, it will change as the traffic that passes through the router changes. igure 3 shows the trend of network delay as of the traffic on the path changes.

We can see that the queue in the router starts to accumulate at point C, but still, there is no loss. Starting from point L, the queue size overflows. 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 can be treated as the approximate queuing delay Q which represents congestion situation

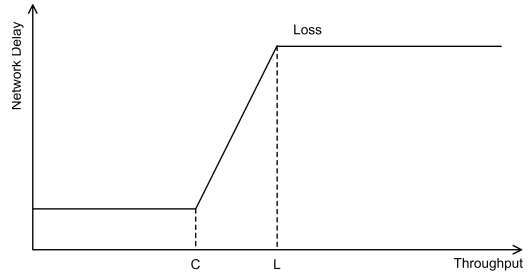


Figure 3: Network delay trend as throughput increases

on the related path. By observing the real-time queuing delay of each path, we can adjust the weight of each path to properly assign future packets.

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.

$Q_{avg} = \frac{\sum_{i=1}^{N} Q_i}{N}$;
if $Q_i \leq Q_{avg}$ **then**
  $W_i = W_i + S$;
  if $W_i > 1000$ **then**
    $W_i = 1000$;
  end if
else
  $W_i = W_i - S$;
  if $W_i < 1$ **then**
    $W_i = 1$;
  end if
end if
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 the granularity value 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 10 milliseconds. In our evaluation, this configuration can converge fast with low fluctuation.

3.8.2 Customized Path Selection Framework

In practice, maybe high throughput can represent most user requirements, but in some special use 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 customized path selection policy based on the destination and length of packets. This customization provides a fundamental framework for more advanced path selection algorithms.

In our prototype, we define a dedicate MPIP routing table as shown in Table 6.

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 prototype, only two priorities are supported. $P\_Throughput$ means throughput is the first priority, the delay-based path selection algorithm in Section 3.8.1 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 port 23, if the packet length is smaller than 200 bytes, the lowest delay path will be chosen, otherwise, delay-based algorithm is used. The second row actually is useless in our prototype because 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 5221 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. In MPIP, because we maintain the path in a customized pattern, it is more feasible to have application specific routing decision. As we mentioned above, we only provide a basic framework for customized routing in MPIP. It only has limited functionalities. It can only specify the application 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.

3.9 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 socket addresses. In our prototype, we call a connection's unique socket a session. Since multiple IP paths is introduced into the system, a session module is development to manage the mapping between different IP addresses and session. To maintain all sessions, each MPIP enabled node will maintain an instance of Table 7.

Table 6: MPIP routing table

IP Address	Port Number	Protocol	Start Length	End Length	Routing Priority
$*$	23	TCP	0	200	$P\_Response$
192.168.1.2	80	TCP	200	$*$	$R\_Throughput$
192.168.1.3	5221	UDP	0	500	$P\_Response$

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

3.9.1 Addition/Removal of Sessions

Session ID is the unique identity of one session in local system, but globally, it still needs node ID to identify the node it belongs to. 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. At both sides of one connection, the session ID needs to be the same to locate the correct entry in Table 7.

After the MPIP availability handshake has been successfully completed on one connection, 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 with an unique session id. Until now, no other paths have added into Table 3 for this session because there was no session id. 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. The receiver extracts these information and inserts one entry into its own Table 7. For the same session, both sides of the connection have the same session id.

Now, both sides of the connection has the entry that identify the same session. Besides the session ID, all IP addresses and port numbers can be different 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 will go 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.

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. The only operation that can apply to them is removal.

Once session entry has been inserted into Table 7 at both side, different paths can be chosen to transmit future packets that belong to this session. For each connection, the system internally maintains a socket pair with source/destination IP addresses and ports, the system maintained socket information is the same as the pair maintained in Table 7.

3.9.2 Workflow of Sending/Receiving Packets

For one regular packet, when being sent, it goes through application layer, transportation layer, then arrives at network layer which is IP layer. Before the packet arrives at IP layer, socket information inside the packet is the one that stores in a system maintained table. In regular scenario, the system will find a proper interface to push out the packet according to the destination IP address in the IP header and routing table. But since we need to apply MPIP, additional operations need to be done after the packet arrives.

When sending a packet, the system looks into Table 7, and finds the session entry that matches the original socket information, extracts the session ID to fill into the field *Session ID* in the control message. 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 in the socket pair, we can find the *Node ID* in Table 4, then in Table 3, all entries that have the correct node ID are eligible. Among all these paths, the most suitable path will be chosen out through the mechanism introduced in Section 3.5.

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

When receiving a packet, the receiver extract the node ID and session ID in the control message, with these two parameters, the node locates the original socket pair in Table 7, and modify the IP header with source and destination IP address, modify TCP header with source and destination port. For the fields related to path measurement, they will be processed as explained int Section 3.5. Now the packet is back to it shape that can be recognized by upper layer, it is ready to be pushed up.

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. The protocol field of IP header needs to be modified to TCP too.

3.9.3 Dealing with out of order TCP traffic

In modern devices, different interfaces can have totally different delay behaviour. For mobile devices, the WiFi interface and cellular interface can have huge difference in 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

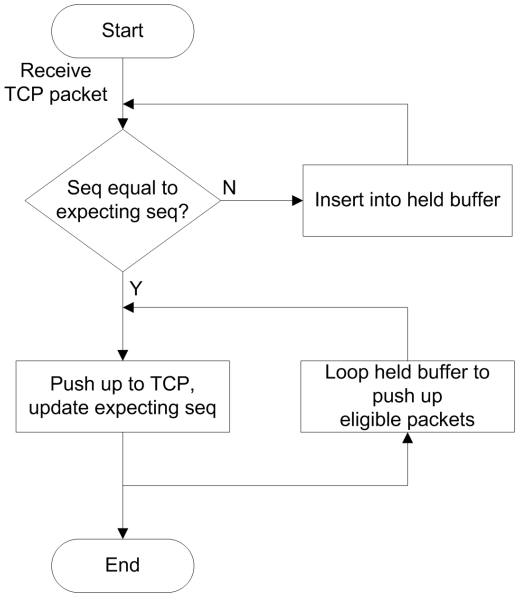


Figure 4: TCP Out-Of-Order Packet Process

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 does not know if it is because a TCP segment was lost or simply that a segment was delayed and received out of order at the receiver. If the receiver can re-order segments, it should not 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 segment 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 imagine a lot of out-of-order packets will happen. 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. The overall process is shown in Figure 9(a).

For each session in Table 7, if it is TCP protocol, the system will maintain 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 the related 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.

3.10 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 side-by-side comparison between MPTCP and MPIP when referring to TCP connection. Besides this, we also 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 in a straightforward pattern 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.

3.11 Andriod implementation

4. PERFORMANCE EVALUATION

To evaluate the performance of our proposed system, we implemented our multipath IP in Ubuntu under Linux kernel 3.12.1 and install the prototype on two desktops. Both desktops are connected directly to one router without any middlebox. At each desktop, two NICs working with $100Mbps$ capacity are installed which means that there are totally 4 paths and the throughput upper bound is $200MBps$ between the two nodes. Also, to evaluate our prototype under multiple scenarios, for some experiments, we limit the capacity of one NIC card on the client to $30Mbps$ with built-in netem feature in Ubuntu. Also, we replace one NIC card on the client with a wireless NIC to evaluate our system for wireless environment.

Besides the controlled lab experiments, we also evaluate MPIP on the Internet. We connect the client to a customized node in Emulab. According to our test, the capacity between our client in the lab 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 card, 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 any special configuration to the default TCP congestion control algorithm, neither we do any optimization to TCP parameters to squeeze out all possible throughput. Beside enabling MPIP feature at he kernel source tree, to make MPIP feasible, we need to configure IP routing rules to route incoming packets to a proper NIC as in MPTCP.

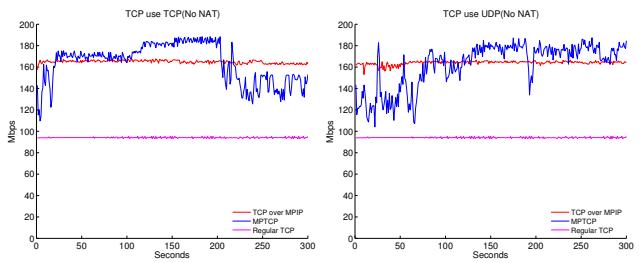
Except specific experiments that can only be applied to MPIP like UDP experiment and customization MPIP routing, we try to do side-by-side comparison with MPTCP for TCP connections. We will figure out how these two features work independently and together as stated in Section **??**;

For all throughput related experiments, we used iperf3 to transmit packets from the client to the server.

4.1 TCP/UDP throughput enhancement

In this section, we try to verify that our system can achieve high throughput in both TCP and UDP scenarios.

In Figure 5,



(a) TCP throughput with psudo TCP connection

(b) TCP throughput with UDP Wrapper

Figure 5: Side-by-side comparison between different MPIP TCP implementations

MPIP TCP uses fake TCP connection for all experiments below.

4.2 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 9(a).

4.3 Skype voice call improvement

In Section 3.8.2, 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 9(b) 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.

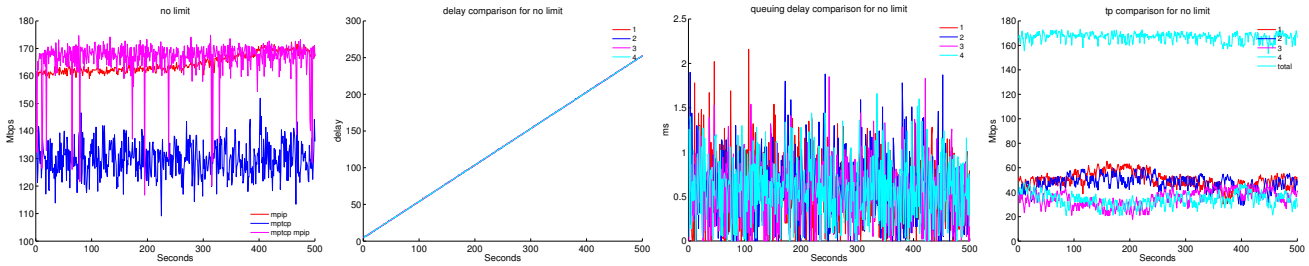
4.4 Smooth connection switch

In Figure 9(c), 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

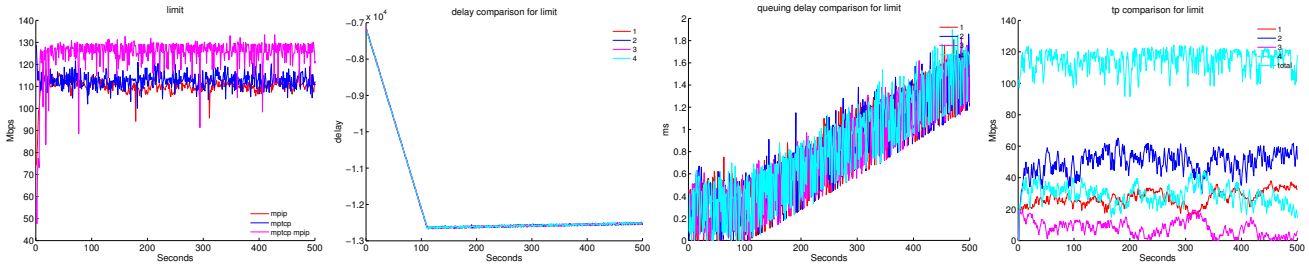
6. REFERENCES

[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.



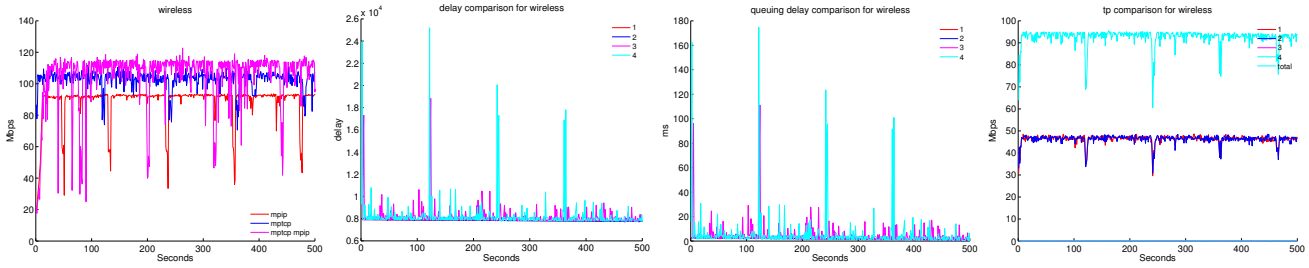
(a) Throughput among MPIP, MPTCP and Together for no limit (b) Delay among different paths for no limit (c) Queuing Delay among different paths for no limit (d) Throughput among different paths for no limit

Figure 6: Side-by-side comparison for no limit



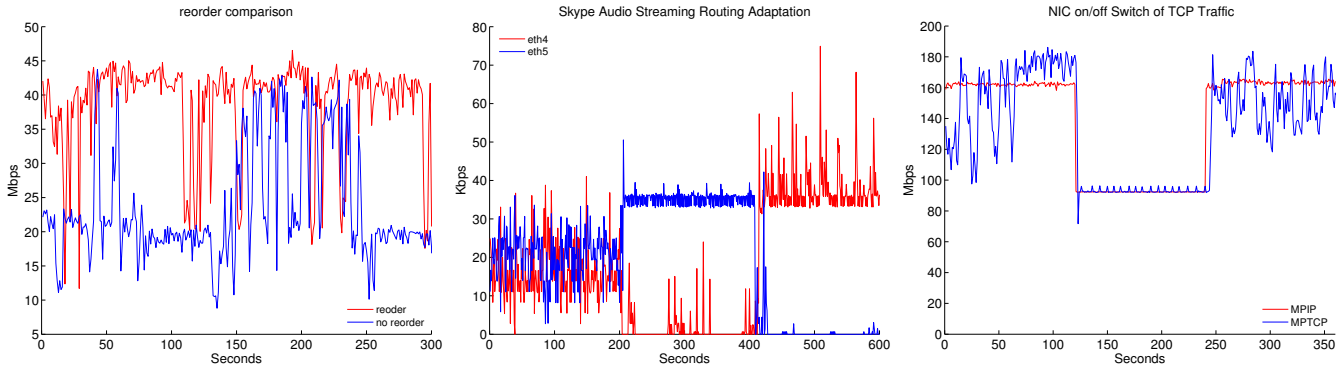
(a) Throughput among MPIP, MPTCP and Together for limit (b) Delay among different paths for limit (c) Queuing Delay among different paths for limit (d) Throughput among different paths for limit

Figure 7: Side-by-side comparison for limit



(a) Throughput among MPIP, MPTCP and Together for wireless (b) Delay among different paths for wireless (c) Queuing Delay among different paths for wireless (d) Throughput among different paths for wireless

Figure 8: Side-by-side comparison for wireless



(a) out-of-order process (b) Skype Audio Streaming Routing Adaptation (c) Connection Smooth Switch