Viscous: An End to End Protocol for Ubiquitous Communication over Internet of Everything

Abhijit Mondal, Sourav Bhattacharjee, Sandip Chakraborty
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur
Kharagpur, India 731303
Email: abhimondal@iitkgp.ac.in, souravb65@gmail.com, sandipc@cse.iitkgp.ernet.in

***Abstract*—The nature of Internet traffic has changed dramatically within the last few years, where a large volume of traffic is originated from mobile applications (known as apps), web based multimedia streaming, computation offloading like cloud computing and Internet of Things (IoT) etc. These types applications generate multiple parallel short lived end-to-end connections. However, the three major requirements of todays' end-to-end traffic over the Internet, such as (a) support for mobility of devices, (b) capacity improvement through multi-path end-to-end transmissions, and (c) support for short-lived parallel connections, are not substantiated through the widely-deployed transmission control protocol (TCP). Further, the recent developments of multi-path TCP (MPTCP) as well as User Datagram Protocol (UDP) based Google's Quick UDP Internet Connections (QUIC) also fail to support all the above three requirements. As a consequence, in this paper, we develop a new end-to-end transmission protocol, called Viscous, to support the above three requirements over the Internet. Viscous is developed as a wrapper between the application and the transport layer, that works on top of the UDP and supports end-to-end reliability as well as congestion control while transmitting short-lived flows over multiple end-to-end paths. We introduce a number of novel concepts at Viscous, such as parallel and sequential flow multiplexing, decoupling of flow and congestion control etc. to overcome the problems associated with the current transport protocols. Viscous has been implemented and tested over a variety of environments, and we observe that it can significantly boost up the performance of the end-to-end data transmission compared to TCP, MPTCP and QUIC.**

***Index Terms*—end-to-end transmission; TCP; MPTCP; QUIC; mobility; short flows**

I. Introduction

The Internet follows the conceptual design from a set of protocol suites where the major components are network layer Internet Protocol (IP) and the connection based reliable transport protocol TCP, which popularly form the TCP/IP protocol suite. However, during the last decade, it was well felt that the basic Internet architecture based on TCP/IP is not suitable for growing demands of extending the Internet over a large number of smart devices at the edge, that spans from the smart handheld devices to the network of low power sensors and actuators, such as Internet of Things (IoT). The increased number and variety of edge devices have reduced the scalability and performance of the present Internet architecture, fundamentally because the nature of data traffic in todays' Internet is different from the one that was envisioned during the development of the TCP/IP protocol suites [17].

Limitations of current transport protocols: While TCP provides reliable end to end connectivity, it has three fundamental problems. (a) *Mobility*: Supporting seamless communication during mobility is one of the major requirements for today's Internet. However TCP is a connection oriented protocol; a connection in TCP is identified by the tuple (source IP, destination IP, source port, destination port). If one of the source or destination IP changes, the ongoing TCP connection needs to be dropped, and a new connection needs to be established [18]. (b) *Multihoming*: If a device is connected to the Internet via multiple network interfaces, it is called a multihomed device. Multihomed devices have the power to transmit or receive data via multiple paths through multiple physical interfaces. However, normal TCP cannot utilize multiple interfaces available to a multihomed device [5]. (c) *Short-lived Flows*: TCP employs a slow start phase to handle network congestion. However, a short-lived flow may fail to come out of the TCP slow start phase, resulting in underutilization of available network resources [6]. Interestingly, many of the today's application over the Internet, like web browsing, IoT communication, many smartphone apps, generate parallel as well as sequential short lived flows.

Related works and their limitations: A number of recent research works, such as [1], [5], [6], [10], [12], [13], [18] and the references therein, have revisited the TCP design fundamentals considering the needs for optimization at the transport layer, so that the available network capacity can be fully utilized for both the event driven short-lived traffic as well as the real time multimedia streaming traffic. Consequently, the network community has explored end-to-end protocols to support the above mentioned features at the transport layer. *Multi-path TCP* [16] has been developed for this purpose, where the connection between a sender and a receiver is established via multiple paths through multiple interfaces. A large number of recent works, such as [2], [15] and the references therein, have explored various aspects of MPTCP and measured its performance over dynamic Internet traffic scenarios. However, as explored in [11], MPTCP does not perform well for short flows. To address the issue of short-lived flows, Dukkipati *et. al.* [7] have suggested to use an initial congestion window size of at least 10, so that the flows can come out of the slow start phase. Later Google

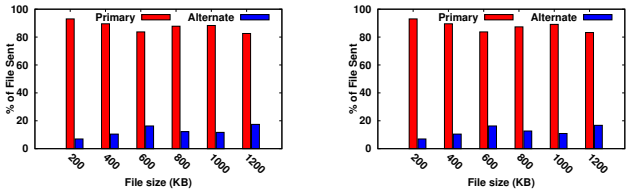
has developed an application layer protocol called SPDY [8], that can multiplex multiple web requests over a single TCP connection. However, it suffers from the *Head of Line* (HOL) blocking issue; where if one or more packets get lost during transmission, all the flows need to wait until TCP recovers the lost packets. To address the HOL blocking issue, Google has further developed a UDP based experimental protocol called QUIC [3], [4]. QUIC is similar to SPDY, but it uses UDP as the transport layer protocol instead of TCP. QUIC can handle reliability, congestion control and flows control over the Internet as well as supports mobility. However, it does not have any control over the path it selects; and the path selection mechanism is completely dependent on the underlying routing algorithm. Therefore, QUIC is not a truly multipath protocol and does not support multihoming.

Contributions of this Paper: As a consequence, in this paper, we develop a multipath protocol, called *Viscous*, that can address the three fundamental shortcomings of the existing end-to-end protocols – mobility, multi-homing and short-lived event driven flows. We develop Viscous as a user level flow management protocol that runs on top of the UDP protocol, similar to Google's QUIC, however is bundled with a number of new features. Viscous can multiplex flows from multiple applications and decouples flow control from congestion control, so that an application flow does not suffer from network bandwidth underutilization problem. Further, Viscous is free from HOL blocking, and supports reliability as well as congestion control over unreliable UDP based end-to-end data transmission. We implement Viscous as a Linux application library, and test its performance over an emulated platform. We observe that Viscous can significantly improve the transport layer protocol performance compared to standard TCP, MPTCP and QUIC, when large number of short lived flows are generated from the edge devices.

II. Motivation Behind Viscous: Why MPTCP is Not Good for Short Lived Flows?

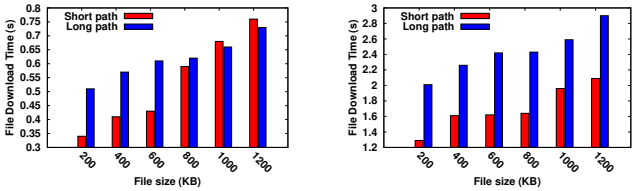
MPTCP is the most widely explored alternative for TCP, which supports multiple paths through multiple interfaces, while providing TCP like congestion control and reliability features for end-to-end connection. Here, we first explore whether we can develop a MPTCP like protocol or augment MPTCP, so that we can support better network utilization for short flows. For this, we conduct an experiment with the help of `Mininet` environment[1], where we setup a network with two multi-homed hosts with two distinct paths between them. We vary the round trip time (RTT) for the two paths, and then transfer data between the two hosts. For our experiment, we have configured the Linux kernel of the hosts with MPTCP version 0.91 [2]. We set the path bandwidth as 100 Mbps. We keep a file at the server, and download that file from the client through MPTCP based connection. To observe the MPTCP behavior for various connection types, we vary the size of the file, and measure the impacts.

[1]http://mininet.org/ (last accessed: 24 April 2017)
[2]http://multipath-tcp.org/ (last accessed: 24 April 2017)



(a) RTT = 80ms (b) RTT = 320ms

Fig. 1. Percentage of data share between the primary and the alternate paths



(a) RTT: short path = 80ms, long path = 120ms (b) RTT: short path = 320ms, long path = 480ms

Fig. 2. File download time when short path is the primary path vs when long path is the primary path

A. Utilization of Network Bandwidth at Alternate Paths for Short Flows

MPTCP first initiates the connection through one of the available paths, called the *primary path*, and then explores the *alternate paths* to initiate alternate connections through them. In the first experiment, we try to look into the amount of data shares between the primary path and the alternate path. For this, we keep the bandwidth and latency for both the paths same, and vary the flow duration by increasing the size of the file to be downloaded at the client from the server. The results are plotted in Fig. 1. We can observe from that figure that the primary path forwards significantly more data compared to the alternate path. By exploring the connection logs, we find that by the time MPTCP sets up a connection to the alternate path, most of the data for a short flow has been transferred over the primary path. Further, the congestion controls for the primary and the alternate paths are handled separately, and therefore the alternate path also needs to go through the slow start phase.

B. Impact of Path Selection for Connection Initiation

We then perform another experiment on path selection, where the two paths have different RTT. Here, we explore the effect of primary path selection. Consequently, we vary the RTT of the two paths, and the results are plotted in Fig. 2. The two different bars in the graphs show the average file download time for the two cases – (i) the low RTT path is used as the primary path, and (ii) the high RTT path is used as the primary path. We observe that for small size file download, the performance varies a lot based on the selection of the primary path.

C. Take Aways

In summary, the lessons learned from the above experiments are as follows. (a) **Connection establishment time is an**

overhead for short flows. Further flow based congestion control algorithms result in the underutilization of network bandwidth, because every transport layer flow independently increases its transmission rate from the scratch (by increasing the congestion window value for TCP or MPTCP), and the flow may end before its transmission rate reaches to the network capacity. (b) **Connection establishment and path selection cannot be independent for a multi-path protocol**, because the performance depends on the path selection mechanism. We need to develop a mechanism where the data transmission can be initiated simultaneously at multiple paths.

III. Viscous: Protocol Descriptions

To overcome the limitations of existing transport layer protocols as discussed in the previous section, we propose a new end-to-end protocol called Viscous. It is a connection oriented multipath multi-flow protocol which is not coupled with the network stack, and work as a wrapper or a middleware in between the users' application and the transport layer of the network protocol stack. The basic design philosophies for Viscous are as follows.

1) To mitigate the signaling overhead associated with connection establishment, Viscous multiplexes multiple flows over a single Viscous connection. This reduces the connection setup time for short-lived flows.
2) Viscous does not maintain a separate and independent congestion control for every application flows. Rather, it maintains a single congestion control mechanism for a Viscous connection which is a multiplex of multiple flows. Further, the congestion control is path specific, that is, congestion is monitored at every path from a source to a destination, where a Viscous sub-flow is initiated.
3) To handle HOL blocking problem, Viscous decouples congestion control from the flow control. Viscous flow control manages the data generation rate from the applications, whereas the congestion control maintains the rate of traffic ingestion into the paths based on congestion feedback. However, to avoid buffer overflow, a feedback if forwarded from the congestion control to the flow control module, whenever necessary.
4) Viscous follows a modular architecture for ease development of applications. Any Viscous module can be tuned independently to make it suitable for a specific requirement. This way, application layer quality of service (QoS) can be provided with the help of Viscous.
5) Viscous works on top of UDP, similar to Goggle's QUIC; therefore it mitigates the transport layer protocol overhead which is associated with TCP.
6) Viscous supports different types of mobility without breaking an existing connection. With the help of a unique client and server specific identifier which is shared during the initial connection establishment, Viscous can continue with the existing connection, even if the server or the client changes its IP address.

Fig. 3. Flow-Channel architecture in the Viscous protocol. Application sends and receives data through the flows. Internally, Viscous sends and receives data using multiple channels over multiple paths.

A. Key Concepts Behind the Design of Viscous

Viscous follows a layered architecture with two different layers, as shown in Fig. 3. There are two key concepts behind the design, development, and implementation of Viscous to support ubiquitous transport of data over any Internet devices – *flow* and *channel*.

1) Channel: Channels are individual connections between two devices over multiple paths. Paths are defined by source and destination IP pair. There is one single channel for each path. To avoid connection time path imbalance like MPTCP, as we observe in Fig. 2, we initiate all the channels simultaneously after connection establishment and before any flows can start. Connections are established specific to a destination host whenever an application from a source host requests for a connection. This connection is shared with other applications which intend to send data to the same destination host. This way, flows are multiplexed specific to a destination host.

Here connection means Viscous connection. Channels are similar to individual TCP connections or sub-flows in the MPTCP. Each channel has dedicated buffer to provide reliable packet transmission and have congestion control algorithm not to overflow the network. Channels are identified by four tuple of source IP, source port, destination IP and destination port. All channels are treated as regular, and every channel can participate in transmitting packets from any flows. However, based on other properties like RTT and goodput, one channel can carry more packets than others.

In Viscous, individual application flows do not require separate connection establishment as the connection has already been established between the source and the destination during Viscous flow initiation. Viscous handles congestion control for every individual paths between a source and a destination. Therefore, flow specific congestion control is not required for Viscous, and Viscous maintains path specific congestion control. This way Viscous avoids slow start for every individual flows that share the complete path from the source to the destination. Consequently, channels are persistent in Viscous, and they do not die out after the completion of a flow. As mentioned, a channel can carries data from multiple flows simultaneously by multiplexing them. If one or both the devices are multi-homed, multiple channels can be established that share the traffic load and therefore, achieve better performance.

2) Flow: Flows in Viscous are responsible for data communication while maintaining the flow-control between a source and a destination. An application can transmit data over multiple flows simultaneously. The user application sends

data as byte-streams to the flow, and it also receives data in the form of byte-streams from a flow. Flow is responsible for packetizing the byte-stream and sending the packets to the lower layer of Viscous. Viscous puts packets from flows to channels, and the channels take care of the congestion control. This way, we decouple congestion control from flow control in Viscous design.

This channel-flow decoupled architecture gives Viscous the power of utilizing multiple paths even for short-lived flows. Flows do not have to suffer from slow start or connection establishment overhead. The connection establishment is a separate event from channel management, and all the channels between a source and a destination start simultaneously. This removes the problem of path selection as we observe in MPTCP. Further, channel gives an option to support mobility. If one or both the devices change its network address, the associated channel cannot communicate anymore. Viscous can discard the affected channels and initiate new channels using the new network address without any interruption in Viscous connection (application flows).

3) Channel Scheduler: Viscous channel scheduler schedules packets from application flows to one of the channels. A straightforward way to schedule the packets is to allocate channels to the packets in a round robin way. However, round robin channel scheduling is not optimal because different channels can have different network characteristics in terms of bandwidth, delay or packet loss; as they are associated with different end-to-end paths in the network. Therefore we use an acknowledgement (ACK) driven scheduling for Viscous, that works as the base scheduler. Here the scheduler schedules a packet to a channel when it receives the ACKs corresponding to the already transmitted packets in the channel. Whenever the scheduler receives an ACK over a channel, it schedules the next packet to that channel. This provides a **self-clocking behaviour** to the channel scheduler. This way, Viscous resolves the issue of balancing packets among various channels.

B. Connection Establishment: Channels and Flows Creation

Viscous is a connection oriented protocol. So it uses the client-server architecture for creating and maintaining the connections. A Viscous server waits for a new connection from a Viscous client. For each client connection, the server maintains separate sets of channels and other associated modules. In Viscous, communication is possible only after association of the flows to the connections.

Fig. 4 shows the connection establishment and data transmission events for Viscous. Connection establishment at Viscous follows a three-way handshake procedure similar to the TCP connection establishment. However, as mentioned earlier, Viscous connection establishment is one time and does not depend on the number of flows between the same server, client pair. To establish a connection, a Viscous client sends a synchronization (`SYN`) packet with a temporary unique identifier or nonce. The nonce or temporary identifier required to identify a client if the `SYN` packet needs to retransmit.

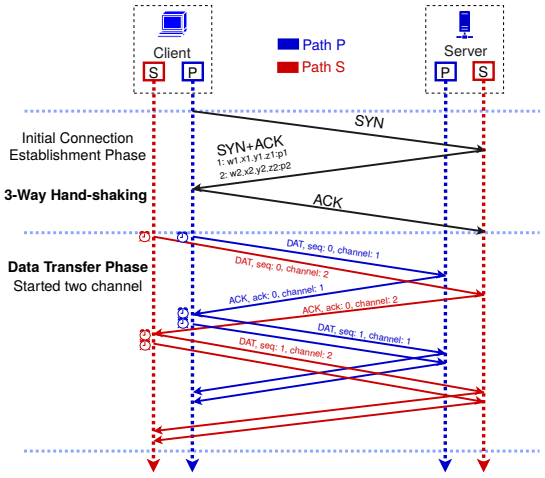


Fig. 4. Three-way handshaking between client and server and data transfer between them. Handshaking can go between any two pairs of interfaces. Here P is the primary path and S is one of the secondary or alternative paths.

We can use the MAC address of the client as this identifier. On receiving the `SYN` packet, the Viscous server generates a fingerprint for the client and sends a `SYN+ACK` packet containing the generated fingerprint. *This fingerprint is used as the connection identifier for the server, client pair*, and every packet between the server and the client includes this fingerprint. In our implementation, we use an SHA256 hash function to generate the fingerprint from the client MAC, server MAC and the current time-stamp used as a unique nonce. It can be noted that different fingerprint is used for different connections, and therefore a fingerprint can uniquely identify a path between a server, client pair. The connection establishment procedure ends with the Viscous client sending an `ACK` packet.

During the connection establishment, the Viscous server informs all its network addresses to the Viscous client, if the server has multiple interfaces. So, immediately after the connection establishment at one channel, the Viscous client initiates all possible channels to the Viscous server. There is no requirement to send an extra control packet to complete the channel establishment. After connection establishment, Viscous clients become ready to add flows to the channels, and initiates data transmissions as the applications send data over the flows.

C. Mobility Support in Viscous

Viscous can support different types of mobility events, as follows.

1) Connect-time Mobility: A connection can fail whenever the server or the client changes its address in between the connection establishment time. With the help of a global name server, this type of mobility can be supported in Viscous. A name server is required to get the new address of the server, when the server changes its IP address. There are two cases that needs to be handled.

1) *Client changes its address just after sending a* `SYN`

packet, Server changes its address just after receiving a `SYN` packet: This type of failure is automatically recovered by subsequent retries from the client. During the retry, the Viscous client is identified via temporary unique identifier which is used to generate the fingerprint.

2) *Client or server changes its IP address after receiving the `SYN+ACK` packet from the server*: As `SYN+ACK` packet contains the fingerprint, the client can send the `ACK` packet to complete the three-way handshake by sending the `ACK` packet to the new server address. The new server address is received with the help from the global name server. However, the success depends on how fast the global name server can update the server IP address against its domain name. If this process fails after three retries, the client re-initiates the connection.

2) Individual Mobility: Individual mobility event occurs when one of the interfaces from both the devices change its network address. This type of mobility can affect a Viscous connection, only if the affected channel already has some packets under transmission. The subsequent packets will carry the new address to the remote device. This does not create a problem, because the connection is identified by the unique fingerprint between the server and the client. Further, the new IP address can be forwarded via other unaffected channels. If all the channel fails, the the affected application can take help from the name-server to find out the new address, and the new channels are creates based on that.

3) Simultaneous Mobility: Simultaneous mobility is rather complex, which occurs when all the interfaces of the server or the client change their network addresses simultaneously. In this scenario, the remote application is not reachable at all. In this event, all existing channels are affected. Therefore, the application needs to get the remote addresses from the name server. Once it get the new addresses, Viscous can continue with the existing connections as the fingerprint remains unchanged.

D. Decoupling of Flow and Congestion Control

As mentioned earlier, Viscous decouples congestion and flow control, where congestion control is associated with the channel layer, and flow control is associated with the flow layer. The flow control algorithm uses the receiver advertised window size ($rwnd$) to control the rate of traffic flows from the application, so that the traffic generation rate from the source application does not overshoot the traffic reception rate at the target application. The interesting design choice for Viscous is that if there is congestion in one of the paths among all the available ones, the traffic generation rate from the application may not need to be dropped down when other paths (or channels) have sufficient bandwidth. However, when congestion is severe (there is congestion in more than one paths), that feedback needs to be passed to the flow control module so that the application traffic generation rate can be shaped accordingly to avoid data overflow from the Viscous packet buffer. Therefore, we design the following approach for decoupled flow and congestion controls in Viscous.

In TCP, the effective sender's window size (wnd) is computed as follows.

$$wnd = min(cwnd, rwnd) \tag{1}$$

where $cwnd$ is congestion window size and $rwnd$ is receiver's advertised window size. In the case of Viscous, we separate congestion control and flow control in two different layers. Here, every flow maintains its flow window and every channel maintains its congestion window. However, to satisfy Equation (1) for providing feedback from the congestion control to the flow control, we need to maintain effective sender's window size as follows.

$$\sum_{c \in \mathcal{C}} wnd_c = min\left(\sum_{f \in \mathcal{F}} rwnd_f, \sum_{c \in \mathcal{C}} cwnd_c\right) \tag{2}$$

where $\mathcal{C}$ is set of active channels and $\mathcal{F}$ is the set of active flows. wnd_c and $cwnd_c$ denote the effective window size and the congestion windows size of the channel c respectively. $rwnd_f$ denotes the receiver window size for the flow f. As channels are unaware of flow window size, channel scheduler needs to communicate this information from flows to channels. Channel scheduler has to set $rwnd_c, c \in \mathcal{C}$ in such a way so that it satisfy the following equation.

$$\sum_{c \in \mathcal{C}} rwnd_c = \sum_{f \in \mathcal{F}} rwnd_f \tag{3}$$

Therefore, the calculation of $rwnd_c$ can be done as follows.

$$rwnd_c = \alpha_c \times \sum_{f \in \mathcal{F}} rwnd_f; \quad \text{where} \quad \sum_{c \in \mathcal{C}} \alpha_c = 1 \tag{4}$$

We can apply tweaks in the calculation of α_c. With simple fair queuing mechanism, we can use α_c as follows.

$$\alpha_c = \frac{cwnd_c}{\sum_{c \in \mathcal{C}} cwnd_c} \tag{5}$$

Equation (4) allows each channel to use the Equation (1) directly that satisfies Equation (2) for providing congestion feedback to the flow control module.

IV. Viscous API Implementation

We have implemented and tested Viscous protocol using C++ language in Linux kernel environment with `pthread`. We have made the Viscous implementation open-source, which is available at https://github.com/abhimp/Viscous. Further, due to space constraints, we are not able to provide the complete API details in this paper, and additional implementation details are available at https://abhimp.github.io/Viscous/.

A. Viscous Packet Structure

In Fig. 5, we have depicted the packet structure used in our implementation. Each packet is divided into three parts. They are a) mandatory header, b) variable length optional headers for additional information, and c) data region. The 28 bits mandatory header has all the common fields required for the

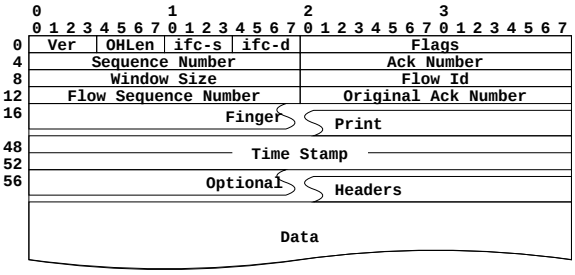


Fig. 5. Viscous packet header structure

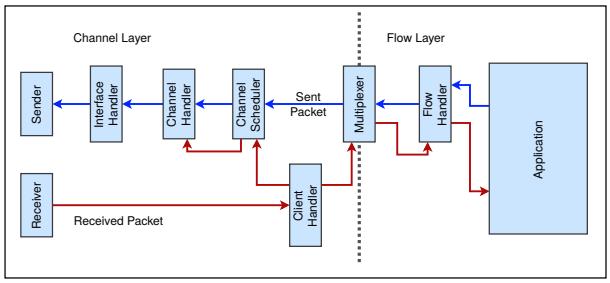


Fig. 6. Internal packet and data flow diagram

communication. The details of the mandatory fields are as follows.

Ver: 4 bits protocol version number.

Optional Header Length(OHLen): We have optional headers of different sizes. This field helps decoder to understand how many optional headers need to be read before data section can be reached.

ifc_s: 4 bit application defined source interface ID.

ifc_d: 4 bit application defined destination interface ID.

Flags: We have used a set of boolean flags. It is similar to TCP. However Viscous need more flags as it is significantly different from TCP.

Sequence Number: Unlike TCP, the sequence number in Viscous is used to identify a packet rather than the byte stream. We use packet based sequence number because of two reasons – a) packets are not created by channel handler; and b) as we multiplex multiple flows, it is easier to track a packet from a flow than a byte stream from a flow. Sequence numbers are used by channel handler to provide reliable communication between two applications. It can be noted that two channels can have same sequence number.

ACK Number: It is cumulative acknowledgement number like TCP acknowledgement number. It denotes that the receiver has received contiguous packet up to this sequence number and it did not receive next packet until the time it was sent from the receiver.

Fingerprint: It is Viscous client's unique ID generated by the server. Every packet includes this field except the initial synchronization packet for connection establishment. In Viscous, packets are discarded if this field is zero or if there is no connection matching this fingerprint (i.e. invalid fingerprint).

Flow ID: Flow ID is an important field in a Viscous packet, which is used by the multiplexer to identify appropriate flow and to forward received packet accordingly.

Flow Sequence Number: Each flow has its flow sequence number independent of the sequence number used by the channel. It requires at the flow layer to reorder the packets at the receiver side. We use packet based flow sequence number in Viscous, similar to the sequence number field.

Original ACK Number: Viscous uses selective acknowledgement mechanism. When the receiver receives an out of order packet, it is supposed to send duplicate acknowledgement packet acknowledging the last conscious packet received. The receiver puts the original sequence number of the packet, which triggers the duplicate acknowledgement. This field gives sender an indication about packets received at the receiver side. So, the sender does not resend them again. It helps Viscous in reducing overall retransmission.

Sent time-stamp: When a sender sends a data packet, it includes the current Unix timestamp in microseconds (μs). When a receiver receives a data packets, it includes this timestamp with the ACK packet. It helps the sender in measuring the RTT more accurately.

B. Modules and Layers in Viscous

Viscous follows a modular architecture as shown in Fig. 6. The various modules in Viscous are as follows.

1) Application: An application is the users' application that uses Viscous library.

2) Flow Handler: In Viscous, an application directly sends data to this module and receives from it. Flow handler packetizes the raw data from the application and sends packets to the lower layer for further processing. It does not need to store any outgoing packets, as the channel layer ensures the reliability. It only keeps track of the packets that it sends, to control the flow rate. Viscous uses a sliding window based flow control mechanism based on the receiver advertised window size. The flow handler also reorders the out of order packets that it receives from the lower layer. There is a receive buffer that stores all the out of order packets. This buffer is an array of packets. The first index of this packet array point to the next expected packet sequence. Whenever the flow handler receives one or more contiguous packets from the expected sequence, it delivers the data from those flows to the application. In Viscous, there is one independent flow handler instance for each of the flows.

3) Multiplexer: The multiplexer is responsible for multiplexing the outgoing packets from multiple flows and forwarding them to the packet scheduler. It is also responsible for demultiplexing incoming packets and forwarding them to appropriate flow handler.

4) Client Handler: Client handler is the manager of Viscous protocol API. For incoming packets, it checks the packet validity using the fingerprint generated during the connection establishment. After validation, it forwards each incoming packet to the *Channel Handler* via *Channel*

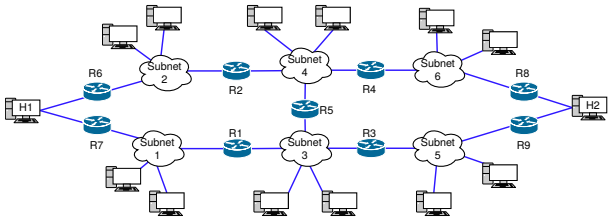


Fig. 7. Topology for Viscous protocol performance evaluation

Scheduler for further handling and processing for congestion control algorithm.

5) Channel Scheduler: It schedules the outgoing packets to one of the channels. As mentioned earlier, we use `ACK` driven channel scheduling. Whenever a channel is ready to send packets, it asks the *Channel Scheduler* for new packets to be sent. A smart scheduler can decide which packet to be sent through a channel based on its algorithm to achieve high throughput with lower latency.

6) Channel Handler: Channel handler is responsible for reliable communication and congestion control in the network. In the Viscous implementation, we use the TCP New Reno congestion control [9] algorithm with the following modifications. In Viscous, the channel handler handles packets, not TCP like byte streams. So, we use packet based sequence number because it is easier to track a packet from a flow than a byte stream from a flow when we multiplex multiple flows. Further, in the Viscous congestion control, each `ACK` contains the sequence number of the packet for which this acknowledgement is triggered. This gives a similar effect as TCP selective acknowledgement (`SACK`) [14] mechanism. Further, as flows are multiplexed, we have modified the fast recovery phase describe in RFC2581 [9] with `SACK` modifications. After receiving the first partial new `ACK`, the channel handler sends all the unacknowledged (via `SACK` or original `ACK` in the packet header) packets up to the received acknowledgement number for each duplicate `ACK`. This modification reduces retransmissions due to timeout events when a series of packets get lost in the network.

V. Experimental Evaluation

To evaluate the performance of Viscous, we have performed several experiments over `Mininet` network emulation platform. We have used the network topology as shown in Fig. 7. In the topology, $H1$ and $H2$ act as the Viscous server and the Viscous client, respectively, or vice-versa. Rest of the hosts generate background traffic to increase the traffic load in the network. We compare the performance of Viscous with following protocols – (a) TCP Cubic with the optimization in [7], MPTCP (version 0.91), and QUIC (open source implementation of `proto-quic`[3]).

A. Experimental Setup

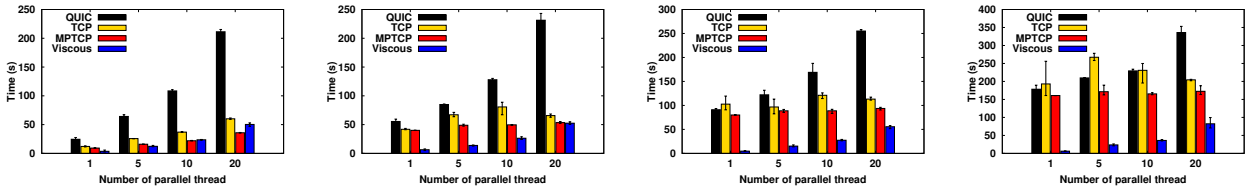
The various hardware and operating system level parameters used in our experimental setup are as follows. We have used Oracle VitualBox to create virtual machines which have been configured with `Mininet` kernel to support network interface virtualization. The guest machine configuration is as follows – single core Intel i7-5500U 64bit CPU with clock speed of 2.40 GHz, RAM: 8GB. The host machine configuration is as follows – Intel Genuine i7-5500U 64bit CPU with clock speed of 2.40GHz, RAM: 8GB with 250GB Solid State Drive. We have used version 2.2.2 of `Mininet` kernel have been used to develop our test setup. In our virtual machine configurations for test setup, the guest operating system is Ubuntu 14.04.4 LTS, whereas the host operating system is Ubuntu 16.04.1 LTS. The link bandwidth for every paths of the topology has been kept as 50 Mbps, and we vary the RTT of the paths from 16ms to 320ms. The background traffic is generated using `iperf` TCP sessions from every host shown in the topology of Fig. 7, attached with the various subnets. Next we present the results and observations from various test scenarios as performed over the two topologies.

B. Performance for Short-lived Flows

In this experiment, we have explored the performance of QUIC, TCP, MPTCP and Viscous for a large numbers of short-lived flows, which is the major drawback of the existing end-to-end protocols. For every setup, we have sent flows using multiple parallel threads to generate multiple simultaneous flows. For each thread, we have generated 100 back-to-back flows; and have varied the flow duration (application traffic generation time) based on an exponential distribution with mean flow duration of 25 seconds. The number of parallel threads, those generate the simultaneous flows, have been varied from 1 to 20. First, we observe the average flow completion time for all the flows, as shown in Fig. 8 and Fig. 9 for without background flows and with background flows,respectively. From the figure, we can see that Viscous performs much better when multiple short-lived flows are transferred over the network. QUIC in this scenario does not perform good, because it fails to utilize multiple interfaces available in the network. Further it multiplexes the sequential flows from the same thread, but treats parallel flows from different threads as separate flows and maintains separate slow start for them. We have observed that when the channel condition is very good (RTT is low), the performance of QUIC is even worse than TCP. By analysing the trace, we found that many of the QUIC connections get stalled due to coupled congestion control and flow control over multiplexed flows, and the average flow completion time gets severely affected. Here Viscous takes the advantage by decoupling flow control from congestion control, and improves the end-to-end performance.

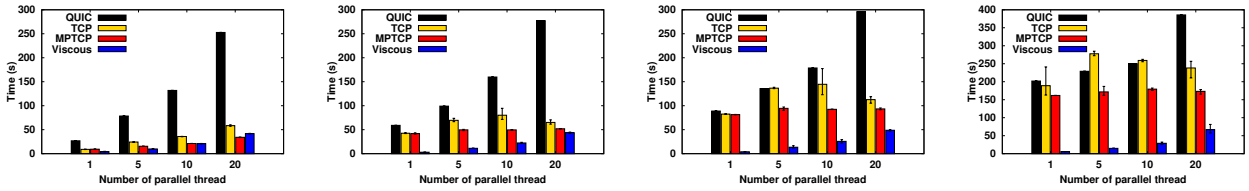
MPTCP suffers from the path imbalance problem as discussed earlier, and TCP Cubic can not utilize multi-homing capacity for the flows. To analyze the effect of short-lived flows, we compare the congestion window evolution for TCP and Viscous for a specific case, as given in Fig. 10, when RTT is 16ms with 5 parallel threads. We observe that the average congestion window for TCP is significantly less.

[3]https://github.com/google/proto-quic (last accessed: April 25, 2017)



(a) RTT=16ms (b) RTT=80ms (c) RTT=160ms (d) RTT=320ms

Fig. 8. Average flow completion time for short flows without background traffic



(a) RTT=16ms (b) RTT=80ms (c) RTT=160ms (d) RTT=320ms

Fig. 9. Average flow completion time for short flows with background traffic

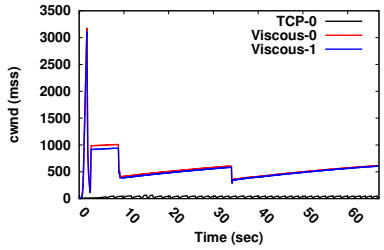


Fig. 10. Congestion window evolution

The congestion window evolutions for Viscous along the two paths (R6-R2-R4-R8, termed as Viscous-0, and R7-R1-R3-R9, termed as Viscous-1) have been shown, and in both the paths, Viscous can achieve a significant higher values of the congestion window. Consequently, Viscous sub-flows can utilize the available bandwidth at the paths in a significantly better way for multiplexed short-lived flows.

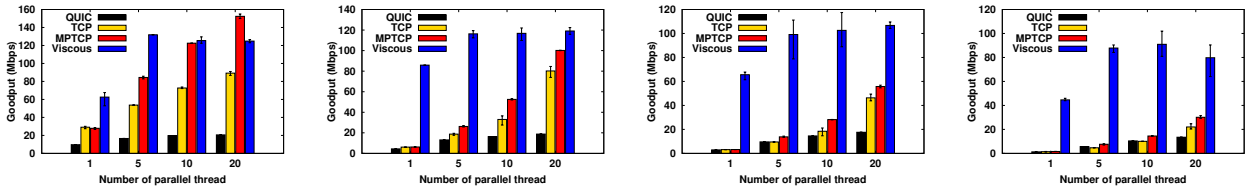
Next, we observe the average goodput of the flows for the four different protocols under the same scenario, as shown in Fig. 11 and Fig. 12, for without background flows and with background flows, respectively. Similar to the flow completion time, we observe that the goodput of Viscous is higher than all other protocols. Viscous can utilize the capacity of multiple paths through flow multiplexing and maintaining connection specific congestion window evolution over the time; and therefore it attains higher goodput compared to other competing end-to-end protocols. Further, we observe that the goodput for Viscous is significantly boosted up compared to other protocols, when RTT of the paths is high. This indicates that Viscous overshoots the performance compared to other protocols when network condition is poor. QUIC, TCP and MPTCP can not cope up with the poor network condition, whereas Viscous can effectively utilize the available capacity under all the scenarios. The reason behind this better performance is the Viscous channel scheduler, that uses a self-clocking mechanism to schedule and trigger the transmissions over multiple channels based on the ACK packets. This self-clocking helps in proper utilization of the channels even under poor RTT conditions.

C. Performance for Long Flows over a Single Path – Worst Case Performance of Viscous

Till now we have tested the performance of Viscous for short-lived flows. To compare its performance for long flows, we have performed this experiment, where we send the various size of files from the host $H1$ to the host $H2$ over Topology-1. We plot the time required to transmit a complete file over the network using TCP and Viscous when both use only one path in the network. The results are plotted in Fig. 13. From this figure, we observe that the performance of Viscous is almost equal for long flows over a single path, which we consider as the worst case performance of this protocol. This experiment shows that in the worst case, Viscous performs as good as TCP Cubic, however, under the scenarios with multiple short-lived events triggered or user generated flows, Viscous can significantly boost up the end-to-end performance.

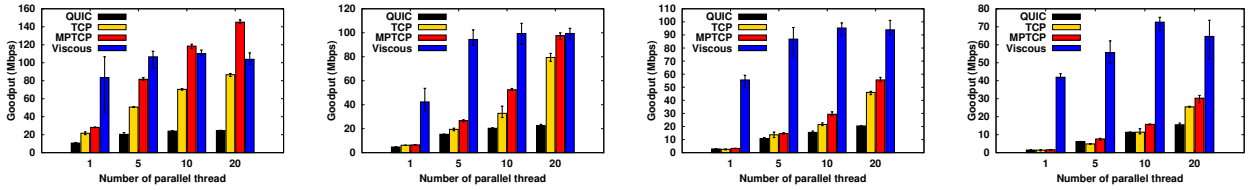
VI. Conclusion

Considering the limitations of current transmission protocols for todays diverse device and traffic characteristics, in this paper, we have developed Viscous that provides a middleware between the user application and the transport layer to handle end-to-end network characteristics. Viscous is completely compatible with the current network protocol stack, while it provides significant performance boost in the application throughput. It is developed on top of the UDP along with a set of novel features to support mobility,



(a) RTT=16ms (b) RTT=80ms (c) RTT=160ms (d) RTT=320ms

Fig. 11. Average goodput for short-lived flows without background traffic



(a) RTT=16ms (b) RTT=80ms (c) RTT=160ms (d) RTT=320ms

Fig. 12. Average goodput for short-lived flows with background traffic

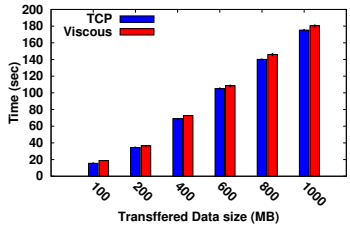


Fig. 13. Performance of Viscous and TCP for long Flows over a single path

multi-homing and short-lived parallel and sequential flows. From the prototype implementation and performance analysis over an emulated environment, we show that the worst case performance of Viscous is similar to TCP, which indicates its applicability for a wide range of applications. In future, our target is to enhance Viscous with application layer QoS support to make it an alternate for both non-real time and real time traffic transport.

REFERENCES

[1] A. Abdrabou and M. Prakash, "Experimental performance study of multipath TCP over heterogeneous wireless networks," in *Proceedings of the 41st IEEE LCN*. IEEE, 2016, pp. 172–175.

[2] R. Barik, M. Welzl, S. Ferlin, and O. Alay, "LISA: A linked slow-start algorithm for MPTCP," in *Proceedings of the 2016 IEEE ICC*. IEEE, 2016, pp. 1–7.

[3] G. Carlucci, L. De Cicco, and S. Mascolo, "HTTP over UDP: an experimental investigation of QUIC," in *Proceedings of the 30th Annual ACM SAC*. ACM, 2015, pp. 609–614.

[4] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, "Innovating transport with QUIC: Design approaches and research challenges," *IEEE Internet Computing*, vol. 21, no. 2, pp. 72–76, 2017.

[5] Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure, "Observing real smartphone applications over multipath TCP," *IEEE Communications Magazine*, vol. 54, no. 3, pp. 88–93, 2016.

[6] G. De Silva, M. C. Chan, and W. T. Ooi, "Throughput estimation for short lived TCP cubic flows," in *Proceedings of the 13th ACM MobiQuitous*. ACM, 2016, pp. 227–236.

[7] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing TCP's initial congestion window," *ACM SIGCOMM Computer Communications Review*, vol. 40, pp. 27–33, 2010.

[8] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan, "Towards a SPDYier mobile web?" *IEEE/ACM Transactions on Networking*, vol. 23, no. 6, pp. 2010–2023, 2015.

[9] S. Floyd and T. Henderson, "The New Reno modification to TCP's fast recovery algorithm," Internet Requests for Comments, RFC Editor, RFC 2582, April 1999.

[10] S. Islam and M. Welzl, "Start Me Up: Determining and sharing TCPs initial congestion window," in *Proceedings of the 2016 ACM ANRW*, 2016, pp. 52–54.

[11] M. Kheirkhah, I. Wakeman, and G. Parisis, "MMPTCP: A multipath transport protocol for data centers," in *Proceedings of the 35th Annual IEEE INFOCOM*. IEEE, 2016, pp. 1–9.

[12] K. Liu and J. Y. Lee, "On improving TCP performance over mobile data networks," *IEEE Transactions on Mobile Computing*, vol. 15, no. 10, pp. 2522–2536, 2016.

[13] M. Maity, B. Raman, and M. Vutukuru, "TCP download performance in dense WiFi scenarios: Analysis and solution," *IEEE Transactions on Mobile Computing*, vol. 16, no. 1, pp. 213–227, 2017.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options," Internet Requests for Comments, RFC Editor, RFC 2018, October 1996.

[15] B.-H. Oh and J. Lee, "Feedback-based path failure detection and buffer blocking protection for MPTCP," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3450–3461, 2016.

[16] C. Paasch and O. Bonaventure, "Multipath TCP," *Communications of the ACM*, vol. 57, no. 4, pp. 51–57, 2014.

[17] J. Rexford and C. Dovrolis, "Future internet architecture: clean-slate versus evolutionary research," *Communications of the ACM*, vol. 53, no. 9, pp. 36–40, 2010.

[18] A. Yadav and A. Venkataramani, "msocket: System support for mobile, multipath, and middlebox-agnostic applications," in *Proceedings of the 24th IEEE ICNP*. IEEE, 2016, pp. 1–10.