OpenTCP: Combining Congestion Controls of Parallel TCP Connections

Safiqul Islam, Michael Welzl, Stein Gjessing
Networks and Distributed Systems Group, Department of Informatics
University of Oslo, Norway
Email: {safiquli, michawe, steing}@ifi.uio.no

Jianjie You
Huawei
Email: youjianjie@huawei.com

Abstract—Most Internet communication uses the Transmission Control Protocol (TCP) at the transport layer. TCP carries out many tasks, including reliability, flow control and congestion control. Congestion control is a mechanism that is concerned with the path between two hosts, but TCP instances operate for separate communicating processes (as identified by port numbers). This means that multiple TCP connections between the same pair of hosts compete on the network. We propose OpenTCP – a method to combine the congestion controls of multiple TCP connections. We report experiments which show that OpenTCP improves the performance of concurrent TCP connections in terms of packet loss and queuing delay. OpenTCP also allows to divide the available bandwidth between the flows according to the needs of the applications.

Index Terms—OpenTCP, coupled congestion control

I. INTRODUCTION

Often, multiple TCP connections are initiated between the same two Internet hosts. For example, when browsing the web, using separate connections for images and other files is quite common (this has only recently been addressed by multi-streaming in HTTP/2, the successor of SPDY [9], but older websites will still require opening several TCP connections for a long time). Other examples include servers that provide multiple services – e.g. the Google Content Distribution Network (CDN) offers YouTube, Google search and many other Google services, and it is then possible that a Google search and a Youtube stream require TCP connections between the same pair of IP addresses.

TCP has traditionally been a “closed” protocol. Its behavior is very strictly based on a large set of standards ([7]), which include the specification of congestion control between two endpoints. Here, an endpoint is attached to a port, i.e. multiple parallel connections between the same IP addresses use multiple endpoints. TCP congestion control begins with a phase called “Slow Start”, where the sending rate (as determined by a variable called the “Congestion Window” (cwnd)) is doubled every Round-Trip Time (RTT) – the time it takes for a packet from the sender to reach the receiver and for the response to return. This phase terminates when, upon exceeding the capacity of the bottleneck, one or more packets are dropped (or congestion-marked [15]). Then, after repairing the loss, “Congestion Avoidance” begins, making the TCP sender increase cwnd by 1 packet per RTT and halve it when incoming acknowledgments (ACKs) inform the sender about packet loss (or marks).

This process repeatedly produces queuing delay whenever the sending rate begins to exceed the bottleneck capacity, and – in the absence of a marking mechanism – results in repeated packet loss as well. Since TCP retransmits lost packets, this also produces even more delay. It is obvious that these problems are magnified when multiple TCP connections individually operate across the same bottleneck. Figure 1 shows the cwnd values of 4 TCP flows that compete across the same bottleneck, from a Simulation using the ns-2 network simulator with TCP-Linux, a module that allows to run the actual Linux TCP code in simulations (Linux kernel 3.17.4 in our case). The 4 flows competed across a 10 Mbit/s bottleneck in a dumbbell topology with an RTT of 100 ms. The bottleneck queue was a normal FIFO (“DropTail”) queue, with a length of one bandwidth×delay-product (BDP), which was 83 packets (the packet size was 1500 bytes). Background traffic was generated using TMIX traffic [21] from a 60-minute trace of campus traffic at Univ. North Carolina (available from the TCP evaluation suite [2]. RTTs of background flows varied between 80 and 100 ms. The average link utilization, loss ratio and queue length in this simulation were 68%, 0.78% and 58 packets, respectively.

This behavior results in a form of fairness that has been called “TCP-friendliness”, which has recently been much criticized [5]. In fact, however, when flows originate from the same
The idea of combining the congestion controls of multiple flows when they traverse the same network bottleneck is not new. To the best of our knowledge, the oldest – and perhaps still best known – proposal along these lines is the Congestion Manager (CM) [3], [4]: here, there is a single congestion control instance in the host that is external to protocols such as TCP, and the flows essentially request their next cwnd or rate value from the CM. This is a good method, but has turned out to be exceedingly hard to implement; moreover, it is even harder to allow the code to make coupling optional (switch it on or off).

The two mechanisms “Ensemble-TCP” (E-TCP) [8] and “Ensemble Flow Congestion Management” (EFCM) [18] share TCP variables such as the cwnd across connections. This approach is close to what we propose for OpenTCP, but it is too simplistic: cwnd and the other variables shared by E-TCP and EFCM are variables of a stateful algorithm, and not correctly sharing the state can produce errors. Here are two examples:

1) TCP reacts once to any number of packet losses within one “loss event” (typically one RTT). For example, losing one or two packets within the same RTT does not normally change the sender behavior: cwnd is halved once. Simply sharing cwnd across multiple flows produces wrong behavior that, depending on how sharing is implemented, can either become too aggressive or too conservative. This is because the duration of a loss event must be shared too, not just cwnd.

2) TCP should never enter Slow Start when ACKs arrive. If one flow stays in Congestion Avoidance and another flow experiences a timeout, this second flow will enter Slow Start, but it really should not. Again, simply sharing cwnd does not solve this problem: the states of the TCP flows should also be shared.

These issues are addressed by our preliminary sharing algorithm in [23], but this algorithm still has some missing features, as we will discuss in Section III. TCP Control Block (TCB) Sharing [20] resembles E-TCP and EFCM in that it simply shares variables, but it is more limited in that this sharing only affects the beginning of new flows. Table I provides an overview of all related work that has been mentioned so far.

In addition to the individual problems mentioned with the previously discussed mechanisms, there is a major common issue of all the related works that are summarized in Table I, with the exception of our own earlier preliminary work [23]: while routers should in theory only be concerned with IP addresses, it is the operational reality of the Internet that two packets destined for the same IP address may take a different path if they have different destination TCP port numbers [12]. In this case, coupling their congestion controllers can go very wrong, as two TCP connections between the same hosts may in fact not traverse the same bottleneck. This problem is an obstacle to deployment of mechanisms to combine congestion controls. TiU [23] addresses this problem by encapsulating multiple TCP connections using a single UDP port number pair. The traffic then looks like a single UDP flow, which has some disadvantages (also explained in [23]), including the need for the receiver to understand the encapsulation and undo it.

Multi-Path TCP (MPTCP) also has a form of coupled congestion control [13], which is similar to OpenTCP in that it tries to make sure that multiple flows (in MPTCP’s case, “subflows”) act like a single TCP connection when they traverse the same network bottleneck. There are, however, important differences: MPTCP’s coupling assumes that flows

The idea of combining the congestion controls of multiple flows should be a system policy – it seems pointless to have to accept a form of fairness that evolves from the competition in the network. We have run a preliminary simulation using the algorithm described in [23] to combine the congestion controls of these TCP flows; the resulting diagram, shown in Figure 2, shows only one line for all four flows because they were able to use precisely the same cwnd values. With this algorithm, priorities can be used to divide the total cwnd among the flows exactly as desired (not necessarily equal as in the diagram).

The average link utilization, loss ratio and queue length in this simulation were 66%, 0.13% and 37 packets, respectively. This is a pronounced reduction in packet loss and queuing delay, at the cost of slightly reduced utilization: multiple separate TCP congestion controllers achieve better utilization because together, they more aggressively probe for available capacity than a single or combined congestion controller. However, this is only a side-effect of their behavior and not necessarily ideal: when desired, a single TCP connection can be made to be much more aggressive, and this is in fact done by the experimental CUBIC congestion control mechanism [11] (default in Linux) and many others.

In the next section, we will describe related work and explain why it has not become commonly used, leaving the problem unsolved. Then, in Section III, we will describe the requirements and our derived design of OpenTCP. Section IV concludes.

II. RELATED WORK

The average link utilization, loss ratio and queue length in this simulation were 66%, 0.13% and 37 packets, respectively. This is a pronounced reduction in packet loss and queuing delay, at the cost of slightly reduced utilization: multiple separate TCP congestion controllers achieve better utilization because together, they more aggressively probe for available capacity than a single or combined congestion controller. However, this is only a side-effect of their behavior and not necessarily ideal: when desired, a single TCP connection can be made to be much more aggressive, and this is in fact done by the experimental CUBIC congestion control mechanism [11] (default in Linux) and many others.

In the next section, we will describe related work and explain why it has not become commonly used, leaving the problem unsolved. Then, in Section III, we will describe the requirements and our derived design of OpenTCP. Section IV concludes.

II. RELATED WORK

The two mechanisms “Ensemble-TCP” (E-TCP) [8] and “Ensemble Flow Congestion Management” (EFCM) [18] share TCP variables such as the cwnd across connections. This approach is close to what we propose for OpenTCP, but it is too simplistic: cwnd and the other variables shared by E-TCP and EFCM are variables of a stateful algorithm, and not correctly sharing the state can produce errors. Here are two examples:

1) TCP reacts once to any number of packet losses within one “loss event” (typically one RTT). For example, losing one or two packets within the same RTT does not normally change the sender behavior: cwnd is halved once. Simply sharing cwnd across multiple flows produces wrong behavior that, depending on how sharing is implemented, can either become too aggressive or too conservative. This is because the duration of a loss event must be shared too, not just cwnd.

2) TCP should never enter Slow Start when ACKs arrive. If one flow stays in Congestion Avoidance and another flow experiences a timeout, this second flow will enter Slow Start, but it really should not. Again, simply sharing cwnd does not solve this problem: the states of the TCP flows should also be shared.

These issues are addressed by our preliminary sharing algorithm in [23], but this algorithm still has some missing features, as we will discuss in Section III. TCP Control Block (TCB) Sharing [20] resembles E-TCP and EFCM in that it simply shares variables, but it is more limited in that this sharing only affects the beginning of new flows. Table I provides an overview of all related work that has been mentioned so far.

In addition to the individual problems mentioned with the previously discussed mechanisms, there is a major common issue of all the related works that are summarized in Table I, with the exception of our own earlier preliminary work [23]): while routers should in theory only be concerned with IP addresses, it is the operational reality of the Internet that two packets destined for the same IP address may take a different path if they have different destination TCP port numbers [12]. In this case, coupling their congestion controllers can go very wrong, as two TCP connections between the same hosts may in fact not traverse the same bottleneck. This problem is an obstacle to deployment of mechanisms to combine congestion controls. TiU [23] addresses this problem by encapsulating multiple TCP connections using a single UDP port number pair. The traffic then looks like a single UDP flow, which has some disadvantages (also explained in [23]), including the need for the receiver to understand the encapsulation and undo it.

Multi-Path TCP (MPTCP) also has a form of coupled congestion control [13], which is similar to OpenTCP in that it tries to make sure that multiple flows (in MPTCP’s case, “subflows”) act like a single TCP connection when they traverse the same network bottleneck. There are, however, important differences: MPTCP’s coupling assumes that flows
III. OpenTCP

A. Requirements

Our analysis of the problem as well as the state of the art lets us derive the following requirements for OpenTCP:

1) Simple to implement: for example, the CM [3] has never become widely used, and it is exceedingly hard to implement. We therefore assume that OpenTCP must be much easier to implement.

2) Correctly share TCP states: While being simpler than the CM, problems with E-TCP and EFCM of sharing too little state have been identified. OpenTCP must avoid these problems.

3) Ensure that packets traverse the same bottleneck: this has only been preliminarily addressed by our own earlier work [23], and it is a problem with all the other related work.

B. OpenTCP design

Addressing item 1, our previous work has shown that it can be easy to combine congestion controllers of real-time media applications [17], [22]. Because of item 2, sharing the state of the TCP congestion controller is a little more sophisticated. This is shown by our preliminary algorithm in [23]; however, it is clear that gradual extensions of this algorithm will still be much simpler and much easier to implement than the CM.

Item 3 is so far only addressed by [23], which encapsulates multiple TCP connections in UDP, using the same UDP port number pair for all of them. This, however, requires new code on the receiver side and may have other problems (e.g., sometimes UDP traffic is rate-limited in the network). For OpenTCP, we can consider other possibilities:

- The IPv6 flow label [1] identifies packets belonging to the same flow, and allows for easier classification in routers based on a 3-tuple of IP addresses and flow label rather than the common 5-tuple (IP addresses, protocol number, port numbers). If a sender sets the same flow label for the multiple combined connections of OpenTCP, it becomes easy for routers to forward them along the same path and for receiver-side code is needed.

- Generic UDP Encapsulation (GUE) gives us another possibility to multiplex several TCP connections over the same UDP port number pair, but using a method that is already deployed in Linux, meaning that we do not need to change receiver code.

- VPNs can use various methods to tunnel all traffic over what looks like a single connection to the network, and hence OpenTCP will work seamlessly for VPNs.

When a new flow joins a long flow that has been operating over a high-capacity network, it may be able to immediately obtain a large cwnd value to start with. If it then immediately makes use of this value, this can produce a burst of hack-to-back packets on the wire, which has a high chance of creating transient congestion and loss. E-TCP and EFCM suggest pacing, which is also not without problems (issues related to timer granularity etc); rather, cwnd could be gradually handed over to the new flow, appropriately increasing it whenever an ACK arrives. This method is planned for OpenTCP, and not yet included in the preliminary algorithm in [23]; it maintains TCP’s ACK-clocking without requiring extra timers.

Another issue that has not yet been addressed by any prior work on TCP congestion control coupling is the sharing of cwnd when applications are not fully using their credit. When TCP connections become application-limited, they cannot know about the current congestion state of the network, and this leads to a more conservative recommended behavior in the TCP standard [10]. However, at the same time, a different

TABLE I: Overview of work related to sharing congestion information, using two flows as an example
connection might be fully using cwnd and probing for network capacity – OpenTCP then allows the limited connection to benefit from the other one.

C. Next steps

The OpenTCP algorithm can be applied to very different congestion control mechanisms and even allows combining a heterogeneous set of congestion controllers with minimal changes. To illustrate this, we first implemented a simple mechanism to combine parallel LEDBAT [19], [16] connections in ns-2 and simulated their behavior using a dumbbell network topology (bottleneck capacity 10 Mbit/s, RTT 100 ms, packet size 1500 bytes). All tests reported in Fig. 3 and Fig. 4 were carried out 10 times with different randomly picked start times over the first second. It can be seen that OpenTCP reduces the average queuing delay without harming utilization. Our tests also showed that fairness becomes perfect and controllable, e.g. by priorities.

Having established that jointly controlling LEDBAT flows works, we now turn to an evaluation of a combination of TCP and LEDBAT using OpenTCP. LEDBAT is an interesting mechanism for this evaluation because it tries to “step out of the way” of other congestion control mechanisms. Figure 5 shows this behavior: TCP attains an average cwnd of around 30 packets, and LEDBAT is pushed aside, getting a cwnd close to 0. While LEDBAT is meant for low-priority background traffic, this behavior is quite extreme and it might be interesting to give it a share of the bandwidth based on a priority instead.

For the test depicted in Fig. 6, we decided that LEDBAT traffic is just as important as TCP traffic, giving both connections the same priorities. As we can see, the cwnd becomes almost exactly equal, at around 10 packets per flow. Note that the total of 20 packets is smaller than the previously mentioned TCP average of around 30 in Fig. 5. This is because TCP increases its rate until the queue overflows before reacting. LEDBAT, however, notices increasing delay as soon as the queue grows and stops increasing its cwnd. Our algorithm notices this reaction and, when used for both TCP and LEDBAT, ensures that none of the connections make the queue grow. In this way, OpenTCP lets the TCP connection benefit from the good delay reduction behavior of LEDBAT. Indeed, our tests showed a pronounced delay reduction when OpenTCP was used while maintaining approximately equal bottleneck utilization in both cases.

IV. Conclusion

In this paper we have proposed OpenTCP as a method to coordinate the behaviour of multiple TCP connections between the same endpoints. The behavior of multiple “closed” standard TCP connections is very complex and may not behave according to the needs of the application. OpenTCP puts the application programmer back in control, and makes it easy to implement open and transparent packet scheduling algorithms according to the needs of the users.
The experiments reported in this paper show that, with OpenTCP, it is easy to implement mechanisms that improve the performance of concurrent TCP connections. OpenTCP reduces packet loss and queuing delay. We have also explained that it is possible to divide the available bandwidth between the flows according to the needs of the applications. Hence, in OpenTCP, the application is in full control of the communication resources.

In future work we will show more examples of the use of OpenTCP, both based on different underlying TCP-algorithms and on different needs from the applications. We will conduct real-life experiments to prove that OpenTCP can efficiently control several concurrent end-to-end flows.

ACKNOWLEDGMENT
This work was funded by a collaborative project between the University of Oslo and Huawei Technologies Co., Ltd.

REFERENCES

Fig. 6: OpenTCP: TCP and LEDBAT can be fair if needed