Master Thesis Project Essay

TCP-in-UDP: Feasible TCP Congestion Control Coupling

Kristian A. Hiorth
kristahi@ifi.uio.no

May 26, 2015

Abstract

In this essay I outline the desirability of TCP Congestion Control Coupling for parallel TCP connections. Current limitations that oppose the use of such a mechanism are examined, and I propose solutions for working around these. Additionally, I propose a software design for implementing both Congestion Control Coupling and the necessary workarounds in the FreeBSD operating system kernel.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
  - 1.1.1 TCP Congestion Control . . . 1
  - 1.1.2 Equal-Cost Multi-Path . . . 2
  - 1.1.3 Network Address Translation and other Middleboxes . . . 2
- 1.2 Research questions . . . 3

2 Discussion **3**
- 2.1 Congestion Control Coupling . . . 3
  - 2.1.1 Simulation results . . . 4
  - 2.1.2 Analytical proof of fairness . . . 4
- 2.2 ECMP path splitting avoidance . . . 4
  - 2.2.1 TCP-in-TCP multiplexing . . . 4
  - 2.2.2 SCTP . . . 4
  - 2.2.3 TCP-in-UDP . . . 4
- 2.3 Tunneling protocol design . . . 5
  - 2.3.1 Header format . . . 5
  - 2.3.2 Capability probing . . . 5
- 2.4 FreeBSD implementation . . . 6
  - 2.4.1 Application layer transparency . . . 6
  - 2.4.2 Connection setup . . . 6
  - 2.4.3 Encapsulation . . . 7
  - 2.4.4 Congestion Control . . . 7
  - 2.4.5 Control interface . . . 7

3 Conclusion **7**

1 Introduction

The Transmission Control Protocol (TCP) provides a Congestion Control (CC) mechanism. This mechanism crucially enables TCP to dynamically adjust its sending rate to actual network conditions, the aim being to send as fast as possible without causing disruptions (i.e. congestion).

However, CC is applied independently to each and every connection, even if they are between the same host and destination pair. Intuitively, this seems sub-optimal, as such parallel connections logically ought to share the same network path, and thus be subject to identical network conditions.

The authors of [3] and [RFC2140] make a good case for coupling the CC for such connections, as well as lay down some ideas about how one might go about doing it.

In practice, complications arise. Due to network mechanisms like Equal Cost Multi-Path (ECMP) routing, one cannot be sure that such parallel connections actually do use the same paths through the network.

The aim of my master thesis project is to develop and test techniques which can help work around these difficulties, in order to allow CC to be coupled for parallel connections, despite running over such uncooperative networks.

In the remainder of this section, I will present some background information on the most relevant concepts, before finally formulating a set of research questions to guide my work. Then, in section 2, I will discuss in more detail ways in which these goals can be achieved, both from a higher-level design point of view, as well as with respect to an actual implementation. Section 3 will sum up the discussion.

1.1 Background

1.1.1 TCP Congestion Control

TCP Congestion Control was developed as a consequence of severe congestion events suffered by the growing Internet of the mid-to-late 1980's [5].

It initially consisted of the Slow-Start and Congestion Avoidance algorithms. Slow-Start extends TCP flow control with a Congestion Window (`cwnd`), which is used to pace the rate at which packets are sent when starting a new connection or recovering from packet loss.

After this initial phase, the system enters Congestion Avoidance, governed by Additive Increase, Multiplicative Decrease (AIMD) of the window size. This makes senders highly sensitive to congestion signals, while being more conservative in their probing of bandwidth (through sending packets faster and faster), allowing for the timely reception of congestion signals and stabilization of network usage.

Beyond these algorithms, the use of exponential back-off in retransmit timers is recommended to avoid introducing timing problems related to retransmitted packets.

TCP Congestion Control has seen a number of revisions since then, both with respect to the tuning of parameters and different ways to recognize congestion signals, for example delay-measurement based CC.

An Explicit Congestion Notification mechanism [RFC3168] has also been introduced, which enables routers to explicitly announce that they are expe-

riencing unsustainable load, rather than rely on indirect observations such as packet loss and delay measurements.

To date, all actively deployed standard TCP congestion control is entirely connection-oriented, and does not explicitly take into account the possibility of several parallel connections between the same two hosts.

1.1.2 Equal-Cost Multi-Path

ECMP [RFC2992] is a routing technique which allows packets to be forwarded along multiple paths as long as they have the same cost – allowing to share traffic load across "tied" routes, so to speak. It is gaining momentum since it allows for load balancing of networks and better utilization of link resources.

Unfortunately, it also makes it very difficult to infer path-linked attributes by observing single connections and extrapolating it to other connections that would logically be routed along the same path. While ECMP implementations usually are devised so as to be TCP-friendly, in that they avoid splitting individual TCP streams, thus avoiding segment reordering, they nullify many of the assumptions made in [RFC2140] about the applicability of per-connection measurements to other connections.

I believe it is a relatively safe assumption to make that, in order to achieve the above mentioned TCP-friendliness, ECMP will avoid splitting a flow as characterized by the flow-id five-tuple consisting of:

$$(\langle\text{destination L3 address}\rangle, \langle\text{source L3 address}\rangle, \langle\text{dest. L4 port}\rangle, \langle\text{src. L4 port}\rangle, \langle\text{L3 protocol type}\rangle)$$

1.1.3 Network Address Translation and other Middleboxes

Network Address Translation (NAT) [RFC2663] is a traffic rewriting and multiplexing technique which is predominantly used to share a single or a few globally routable IP addresses between a larger set of hosts. Since appearing in the late 1990's, it has become almost ubiquitous in access networks that connect end-users (both consumers and enterprises) to the Internet due to the increasingly precarious shortage of IPv4 address space.

NAT functions by intercepting packets at a gateway between the local domain and the rest of the network, rewriting addresses and ports while maintaining a mapping of current connections. Thus, local hosts may be assigned IP addresses in private address space, only the NAT gateway's externally facing network interface needs a globally routable address.

However, this approach poses a problem when establishing new connections. If the translator has not seen outbound packets in a flow, it does not have any mapping of which local host traffic is destined for. This affects both passive ends of TCP connections (listening servers) and UDP endpoints. To overcome this, one can configure static translation rules at the gateway, or attempt the connection in reverse.

In addition to NAT, there exist a great many other sorts of mechanisms in the network that intercept and modify traffic. They are often referred to collectively as Middleboxes. Examples of such contraptions are firewalls, traffic shapers, load balancers and so on.

1.2 Research questions

Overall: *Can parallel TCP connections between the same two end hosts be coaxed into reliably employing congestion control coupling across the Internet?*

ECMP Can ECMP be worked around by embedding TCP segments of different connections (different port numbers) between a same pair of hosts into a single UDP stream (always the same port numbers) between this same pair of hosts?

TCP-in-UDP
How can TCP segments efficiently be embedded in UDP datagrams for this purpose?

Congestion Control Coupling
How can Congestion Control Coupling (CCC) be implemented in an actual production network stack?

Interface
How can CCC+TCP-in-UDP be activated on demand in the most non-disruptive way possible, without hurting latency and stability?

Performance
Does CCC+TCP-in-UDP deliver on the expected performance boosts predicted by simulation data? Can it deliver other positive side effects such as smoother passage through meddlesome middleboxes?

2 Discussion

2.1 Congestion Control Coupling

The notion that it might be wise to perform Congestion Control Coupling is by no means new. [RFC2140] discusses parameters stored in TCP Control Blocks (TCBs) that are inherently common to a specific host pair (association), rather than being characteristic only of an application pair (connection). Herein it is assumed that an association's traffic always takes the same network path.

These parameters are further classified into two groups. Those that are essentially characteristics of the association, such as Round Trip Time (RTT). Others characterize an association in their aggregate, like congestion control window sizes (`cwnd`), i.e. they should not be shared by simply copying the values around.

Parameters can be shared temporarily, by caching them for reuse when opening new connections, overcoming slow-start, although this could lead to problems if network behavior has changed in the mean time. They can also be shared among ongoing concurrent connections, so-called ensemble sharing. This is more challenging to accomplish, but could lead to fairer bandwidth sharing and less interference.

Parts of RFC2140 are currently implemented by mainstream operating systems such as FreeBSD (the *TCP host cache* mechanism) and Linux. As far as I could tell, this is only used to initialize additional connections with cached values, there is no on-going sharing implemented as of now.

2.1.1 Simulation results

The potential speedup from TCP CC coupling, as embodied by the Ensemble-TCP (E-TCP) scheme, is demonstrated in simulations by Eggert et al. in [3].

The use cases in which they showed the largest benefit, namely parallel HTTP requests during a Web page load, will likely be served by the new connection multiplexing features in the recently ratified HTTP/2 standard [RFC7540]. However, those solutions will be prone to Head of Line blocking (HOL) since they still use a single TCP connection for multiplexing.[1]

Similarly positive results are provided by Savorić et al. in [7]. Their simulation also showed an even greater improvement when the last hop is unreliable, as is the case in the ever more common wireless scenario.

2.1.2 Analytical proof of fairness

In [7], Savorić et al. prove that the EFCM CCC scheme (loosely based on E-TCP) upholds the so called TCP fairness property. That is, it does not aggressively swallow all network resources, which would hurt non-TCP background traffic.

2.2 ECMP path splitting avoidance

In order to make CCC useful, it is imperative to somehow ensure that all the parallel connections use the same network path. In my opinion, the easiest and most feasible way of doing this is to bundle together the connections so that they all look like a single flow to an external observer along the route. In other words, I want to ensure they all have the same flow-id five tuple as described in section 1.1.2.

I will now discuss some possible solutions in the following sections.

2.2.1 TCP-in-TCP multiplexing

Multiplex several logical TCP connections over a single actual connection. Bad idea, especially because of HOL.

2.2.2 SCTP

The Stream Control Transmission Protocol (SCTP) [RFC4960] supports this kind of scenario right out of the box. It is a good solution to the problem; in fact, a more future-oriented approach would be to move away from TCP and use SCTP instead. However, this requires radical application level changes and SCTP traffic may not actually pass through all middleboxes because it is a relatively new, at least in Internet terms.

2.2.3 TCP-in-UDP

This is the solution I propose; TCP segments are encapsulated as UDP datagrams while rewriting the TCP header slightly for efficiency reasons. All segments will appear as one UDP flow, thus achieving path unification.

[1] In fact, Google, which proposed SPDY, the precursor to HTTP/2, has also proposed and implemented an alternative transport protocol running over UDP, QUIC – Quick UDP Internet Connections – which avoids the HOL problem.

However, this approach does requires support at both ends of the connection. Overzealous packet scrubbers in security appliances might also hamper the setup negotiation, since it will use non-standard TCP options. On the other hand, it avoids HOL since each segment is sent and delivered as separate and independent datagrams. It can be made quite efficient by omitting fields that are redundantly present in the UDP header.

This approach can be made transparent to the application layer, all trickery happens at the transport layer so it looks like a regular TCP connection to the layers above. If one opts to removes fields in order to reduce the size of the TCP header, it would potentially be visible to applications, since they can rightfully expect the features supported by those fields to work.

2.3 Tunneling protocol design

2.3.1 Header format

To achieve the lowest possible overhead when encapsulating TCP segments in UDP datagrams, entire segments will not be placed as payloads in the datagrams as is. Instead, the TCP header (figure 1) will be rewritten, removing redundant and, perhaps, infrequently used fields. Other fields may also be encoded more compactly to save header space.

The checksum field is entirely redundant, since UDP provides an equivalent facility. Furthermore, it is tempting to elide the infrequently used Urgent Pointer field, along with the associated urgent flag, although this will break compatibility with the, admittedly few, applications that make use of this feature. As of 2011, the Internet Engineering Task Force (IETF) has reaffirmed that TCP implementations must support the urgent pointer feature, although strongly discouraging its use [RFC6093].

Earlier efforts to design such a header format, as described in Internet Drafts by R. Denis-Courmont [2] and S. Cheshire et al. [1], also remove the source and destination port pairs and rely only on the port fields in the UDP header, achieving header length parity with regular TCP by sacrificing the urgent pointer. Unfortunately, I can not replicate this since, unlike the aforementioned authors, my primary goal is to be able to multiplex several TCP connections over the same outside-observable transport flow, as defined by the (source address, destination address, source port, destination port, IP protocol number) five-tuple. However, if desirable, it would be possible to compress the full source and destination port fields into some narrower flow identification field at the cost of additional complexity (introduction of further setup negotiation as well as another demultiplexing step on reception) and fewer possible parallel streams.

An interesting possibility is allowing the transmission of Session Traversal Utilities for NAT (STUN) [RFC5389] payloads on the same UDP socket, allowing for brokered NAT traversal. This can be achieved by reordering the TCP header and using so called magic numbers to distinguish STUN from tunnel traffic, see the above mentioned Internet Drafts.

2.3.2 Capability probing

Since I mainly do not intend applications to deal with controlling the scheme, it will need to rely on some sort of automatic intelligence for activation.

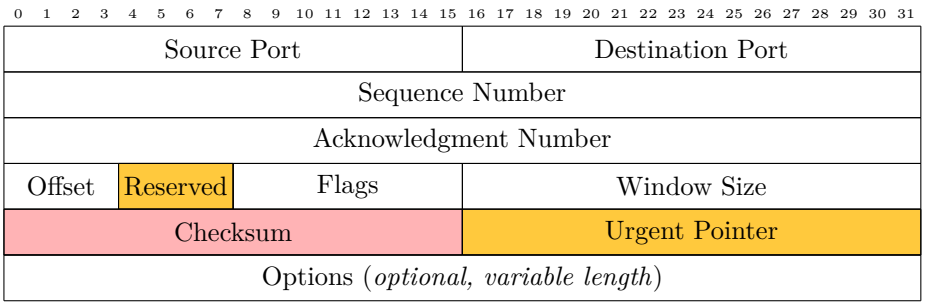

Figure 1: Standard TCP header with redundant fields on red background, compatibility-compromising candidates for removal on orange background

For detecting tunneling capability, which must be present at both ends of the connection to function, I intend to use TCP options during the three-way-handshake to signal this.

Tunneling capability as well as connectivity (no interference from middleboxes etc.) will be prerequisite to enabling the CC coupling. I will seek inspiration from the Happy Eyeballs scheme [RFC6555] for automatic IPv4/IPv6 selection in the design of this activation logic.

2.4 FreeBSD implementation

I will implement the TCP-in-UDP mechanism with TCP CC coupling in the FreeBSD[2] operating system's network stack. My choice fell on FreeBSD because, aside from mere personal preference, it is a high quality Free Software operating system, with a well documented (see e.g. [8, 6]), standards-compliant and well-performing IP network stack.

2.4.1 Application layer transparency

A major goal of the implementation of both the TCP-in-UDP and CCC schemes will be to make them as transparent as possible to the application layer. This way, legacy applications can benefit from them without any adaption, much like they would from other TCP congestion control improvements.

2.4.2 Connection setup

To facilitate auto-detection of TCP-in-UDP tunneling capability, the TCP connection establishment routines must be extended to support the new TCP option flag as described in section 2.3.2.

When TCP-in-UDP is enabled, this option will be appended to the header of the initial `SYN` packet of the TCP three-way-handshake used for opening a new connection. In listening mode, TCP will reciprocate when replying. This option will contain the parameters necessary to connect the tunnel endpoints, namely which UDP port the tunnel is bound to at either host.

[2] `http://freebsd.org`

2.4.3 Encapsulation

In the encapsulation step, the TCP segment header is rewritten as described in section 2.3.1. Then the segment is passed as outgoing payload data to UDP.

Proper TCP segments will be reconstructed before being passed to normal TCP processing upon reception at the destination. This involves splicing out the UDP header and rewriting both the IP and TCP headers – the FreeBSD transport layer expects to be presented with a full IP datagram.

The encapsulation and decapsulation steps will introduce some minor delay into packet processing, notably due to having to carry out some additional copy operations on the message buffers in order to rewrite them.

It is possible to alleviate this by embedding the implementation deeper within the existing TCP and UDP implementation, however I would like to avoid this to reduce the complexity of my code.

2.4.4 Congestion Control

The CCC implementation is logically separate from the TCP-in-UDP tunneling. In theory it should work fine without the tunneling, but as discussed earlier it is generally not practically applicable without.

Similarly to the implementation of the TCP-in-UDP tunneling mechanism, I will seek to implement CCC as non-intrusively as possible. To that effect, it may be possible to leverage the Modular Congestion Control framework [4] introduced in FreeBSD 9. At the time of writing, I have not fully evaluated whether this interface is flexible enough to permit implementing CCC, but I am hopeful that it is sufficient to implement the approach described in [7], which does not rely on added complex functionality such as an output scheduler.

2.4.5 Control interface

Initially, the scheme will be controllable using the `sysctl` configuration system.

Socket options may be added at a later stage, to allow individual TCP-in-UDP aware applications to configure the feature on a per-connection basis.

My implementation will take advantage of the VIMAGE network virtualisation feature of FreeBSD, allowing it to be selectively enabled per virtual network stack. This should hopefully prove advantageous during testing and measuring.

3 Conclusion

In summing up, I believe the TCP-in-UDP encapsulation proposed herein should overcome the ECMP path uncertainty problem, allowing EFCM-like TCB sharing for accomplishing TCP Congestion Control Coupling that actually functions over the Internet. The main drawbacks are that this tunneling requires support on both ends of a connection, unlike other TCP CC adaptations, and that it introduces some minor overhead.

Auto-detection of parallel connections together with probing of UDP encapsulation capability should come together in a Happy Eyeballs like manner to deliver applications the advantages of CCC, when available, without any need for manual intervention.

Combined, I believe these mechanisms could deliver feasible TCP Congestion Control Coupling over the Internet of today, without the need for further cooperation or standards adoption from network equipment operators in the "middle" of the Internet.

References

[RFC7540] Mike Belshe, Roberto Peon, and Martin Thomson (editor). *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7540.txt.

[1] Stuart Cheshire, Josh Graessley, and Rory McGuire. *Encapsulation of TCP and other Transport Protocols over UDP*. Internet-Draft draft-cheshire-tcp-over-udp-00.txt. IETF Secretariat, July 1, 2014.

[2] Rémi Denis-Courmont. *UDP-Encapsulated Transport Protocols*. Internet-Draft draft-denis-udp-transport-00.txt. IETF Secretariat, July 4, 2008.

[RFC4960] Randall R. Stewart (editor). *Stream Control Transmission Protocol*. RFC 4960. RFC Editor, Sept. 2007. URL: http://www.rfc-editor.org/rfc/rfc4960.txt.

[3] Lars Eggert, John Heidemann, and Joe Touch. "Effects of Ensemble-TCP". In: *ACM SIGCOMM Computer Communication Review* 30.1 (2000), pp. 15–29.

[RFC6093] Fernando Gont and Andrew Yourtchenko. *On the Implementation of the TCP Urgent Mechanism*. RFC 6093. RFC Editor, Jan. 2011. URL: http://www.rfc-editor.org/rfc/rfc6093.txt.

[4] David Hayes, Lawrence Stewart, and Grenville Armitage. *Evaluating the FreeBSD 9. x Modular Congestion Control Framework's Performance Impact*. Tech. rep. 110228A. Centre for Advanced Internet Architectures, Swinburne University of Technology, Feb. 28, 2011. URL: http://caia.swin.edu.au/reports/110228A/CAIA-TR-110228A.pdf.

[RFC2992] Christian E. Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992. RFC Editor, Nov. 2000. URL: http://www.rfc-editor.org/rfc/rfc2992.txt.

[5] Van Jacobson. "Congestion Avoidance and Control". In: *SIGCOMM Computer Communication Review* 18.4 (Aug. 1988), pp. 314–329. ISSN: 0146-4833. DOI: 10.1145/52325.52356. URL: http://doi.acm.org/10.1145/52325.52356.

[6] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. 2nd ed. Pearson Education, 2014. ISBN: 9780133761832.

[RFC3168] K. K. Ramakrishnan, Sally Floyd, and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. RFC Editor, Sept. 2001. URL: http://www.rfc-editor.org/rfc/rfc3168.txt.

[RFC5389] Jonathan Rosenberg et al. *Session Traversal Utilities for NAT (STUN)*. RFC 5389. RFC Editor, Oct. 2008. URL: http://www.rfc-editor.org/rfc/rfc5389.txt.

[7] Michael Savorić et al. "Analysis and performance evaluation of the EFCM common congestion controller for TCP connections". In: *Computer Networks* 49.2 (2005), pp. 269–294. ISSN: 1389-1286.

[RFC2663] Pyda Srisuresh and Matt Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 6093. RFC Editor, Aug. 1999. URL: http://www.rfc-editor.org/rfc/rfc2663.txt.

[RFC2140] Joe Touch. *TCP Control Block Interdependence*. RFC 2140. RFC Editor, Apr. 1997. URL: http://www.rfc-editor.org/rfc/rfc2140.txt.

[RFC6555] Dan Wing and Andrew Yourtchenko. *Happy Eyeballs: Success with Dual-Stack Hosts*. RFC 6555. RFC Editor, Apr. 2012. URL: http://www.rfc-editor.org/rfc/rfc6555.txt.

[8] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated. The Implementation*. Vol. 2. Addison-Wesley Professional Computing Series. Pearson Education, 1995. ISBN: 9780321617644.