Internet Engineering Task Force
Internet-Draft
Intended status: Experimental
Expires: October 13, 2011

C. Raiciu
M. Handley
D. Wischik
University College London

April 11, 2011

Coupled Congestion Control for Multipath Transport Protocols
draft-ietf-mptcp-congestion-03

Abstract

Often endpoints are connected by multiple paths, but communications are usually restricted to a single path per connection. Resource usage within the network would be more efficient were it possible for these multiple paths to be used concurrently. Multipath TCP is a proposal to achieve multipath transport in TCP.

New congestion control algorithms are needed for multipath transport protocols such as Multipath TCP, as single path algorithms have a series of issues in the multipath context. One of the prominent problems is that running existing algorithms such as TCP New Reno independently on each path would give the multipath flow more than its fair share at a bottleneck link traversed by more than one of its subflows. Further, it is desirable that a source with multiple paths available will transfer more traffic using the least congested of the paths, hence achieving resource pooling. This would increase the overall efficiency of the network and also its robustness to failure.

This document presents a congestion control algorithm which couples the congestion control algorithms running on different subflows by linking their increase functions, and dynamically controls the overall aggresiveness of the multipath flow. The result is a practical algorithm that is fair to TCP at bottlenecks while moving traffic away from congested links.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on October 13, 2011.

Copyright Notice

Table of Contents

1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Introduction

Multipath TCP (MPTCP, [I-D.ford-mptcp-multiaddressed]) is a set of extensions to regular TCP [RFC0793] that allows one TCP connection to be spread across multiple paths. MPTCP distributes load through the creation of separate "subflows" across potentially disjoint paths.

How should congestion control be performed for multipath TCP? First, each subflow must have its own congestion control state (i.e. cwnd) so that capacity on that path is matched by offered load. The simplest way to achieve this goal is to simply run TCP New Reno congestion control [RFC5681] on each subflow. However this solution is unsatisfactory as it gives the multipath flow an unfair share when the paths taken by its different subflows share a common bottleneck.

Bottleneck fairness is just one requirement multipath congestion control should meet. The following three goals capture the desirable properties of a practical multipath congestion control algorithm:

- Goal 1 (Improve Throughput) A multipath flow should perform at least as well as a single path flow would on the best of the paths available to it.

- Goal 2 (Do no harm) A multipath flow should not take up more capacity from any of the resources shared by its different paths, than if it was a single flow using only one of these paths. This guarantees it will not unduly harm other flows.

- Goal 3 (Balance congestion) A multipath flow should move as much traffic as possible off its most congested paths, subject to meeting the first two goals.

Goals 1 and 2 together ensure fairness at the bottleneck. Goal 3 captures the concept of resource pooling [WISCHIK]: if each multipath flow sends more data through its least congested path, the traffic in the network will move away from congested areas. This improves robustness and overall throughput, among other things. The way to achieve resource pooling is to effectively "couple" the congestion control loops for the different subflows.

We propose an algorithm that couples only the additive increase

function of the subflows, and uses unmodified TCP New Reno behavior in case of a drop. The algorithm relies on the traditional TCP mechanisms to detect drops, to retransmit data, etc.

Detecting shared bottlenecks reliably is quite difficult, but is just one part of a bigger question. This bigger question is how much bandwidth a multipath user should use in total, even if there is no shared bottleneck.

The congestion controller aims to set the multipath flow's aggregate bandwidth to be the same as a regular TCP flow would get on the best path available to the multipath flow. To estimate the bandwidth of a regular TCP flow, the multipath flow estimates loss rates and round trip times and computes the target rate. Then it adjusts the overall aggresiveness (parameter alpha) to achieve the desired rate.

While the mechanism above applies always, its effect depends on whether the multipath TCP flow influences or does not influence the link loss rates (high vs. low statistical multiplexing). If MPTCP does not influence link loss rates, MPTCP will get the same throughput as TCP on the best path. In cases with low statistical multiplexing, where the multipath flow influences the loss rates on the path, the multipath throughput will be strictly higher than a single TCP would get on any of the paths. In particular, if using two idle paths, multipath throughput will be sum of the two paths' throughput.

This algorithm ensures bottleneck fairness and fairness in the broader, network sense. We acknowledge that current TCP fairness criteria are far from ideal, but a multipath TCP needs to be deployable in the current Internet. If needed, new fairness criteria can be implemented by the same algorithm we propose by appropriately scaling the overall aggressiveness.

It is intended that the algorithm presented here can be applied to other multipath transport protocols, such as alternative multipath extensions to TCP, or indeed any other congestion-aware transport protocols. However, for the purposes of example this document will, where appropriate, refer to the MPTCP protocol.

The design decisions and evaluation of the congestion control algorithm are published in [NSDI].

The algorithm presented here only extends TCP New Reno congestion control for multipath operation. It is foreseeable that other congestion controllers will be implemented for multipath transport to achieve the bandwidth-scaling properties of the newer congestion control algorithms for regular TCP (such as Compound TCP and Cubic).

3. Coupled Congestion Control Algorithm

The algorithm we present only applies to the increase phase of the congestion avoidance state specifying how the window inflates upon receiving an ack. The slow start, fast retransmit, and fast recovery algorithms, as well as the multiplicative decrease of the congestion avoidance state are the same as in TCP [RFC5681].

Let cwnd_i be the congestion window on the subflow i. Let tot_cwnd be the sum of the congestion windows of all subflows in the connection. Let p_i, rtt_i and mss_i be the loss rate, round trip time (i.e. smoothed round trip time estimate) and maximum segment size on subflow i.

We assume throughout this document that the congestion window is maintained in bytes, unless otherwise specified. We briefly describe the algorithm for packet-based implementations of cwnd in section Section 4.1.

Our proposed "Linked Increases" algorithm MUST:

o For each ack received on subflow i, increase cwnd_i by min (alpha*bytes_acked*mss_i/tot_cwnd , bytes_acked*mss_i/cwnd_i)

The increase formula takes the minimum between the computed increase for the multipath subflow (first argument to min), and the increase TCP would get in the same scenario (the second argument). In this way, we ensure that any multipath subflow cannot be more aggressive than a TCP flow in the same circumstances, hence achieving goal 2 (do no harm).

"alpha" is a parameter of the algorithm that describes the aggresiveness of the multipath flow. To meet Goal 1 (improve throughput), the value of alpha is chosen such that the aggregate throughput of the multipath flow is equal to the throughput a TCP flow would get if it ran on the best path.

To get an intuition of what the algorithm is trying to do, let's take the case where all the subflows have the same round trip time and MSS. In this case the algorithm will grow the total window by approximately alpha*MSS per RTT. This increase is distributed to the individual flows according to their instantaneous window size. Subflow i will increase by alpha*cwnd_i/tot_cwnd segments per RTT.

Note that, as in standard TCP, when tot_cwnd is large the increase may be 0. In this case the increase MUST be set to 1. We discuss how to implement this formula in practice in the next section.

We assume appropriate byte counting (ABC, [RFC3465]) is used, hence the bytes_acked variable records the number of bytes newly acknowledged. If ABC is not used, bytes_acked SHOULD be set to mss_i.

To compute tot_cwnd, it is an easy mistake to sum up cwnd_i across all subflows: when a flow is in fast retransmit, its cwnd is typically inflated and no longer represents the real congestion window. The correct behavior is to use the ssthresh value for flows in fast retransmit when computing tot_cwnd. To cater for connections that are app limited, the computation should consider the minimum between flight_size_i and cwnd_i, and flight_size_i and ssthresh_i where appropriate.

The total throughput of a multipath flow depends on the value of alpha and the loss rates, maximum segment sizes and round trip times of its paths. Since we require that the total throughput is no worse than the throughput a single TCP would get on the best path, it is impossible to choose a-priori a single value of alpha that achieves the desired throughput in every ocasion. Hence, alpha must be computed based on the observed properties of the paths.

The formula to compute alpha is:

$$\text{alpha} = \text{tot\_cwnd} * \frac{\max_i \frac{\text{cwnd}_i}{\text{rtt}_i^2}}{\left(\sum_i \frac{\text{cwnd}_i}{\text{rtt}_i}\right)^2}$$

The formula is derived by equalizing the rate of the multipath flow with the rate of a TCP running on the best path, and solving for alpha.

4. Implementation Considerations

The formula for alpha above implies that alpha is a floating point value. This would require performing costly floating point operations whenever an ACK is received, Further, in many kernels floating point operations are disabled. There is an easy way to approximate the above calculations using integer arithmetic.

Let alpha_scale be an integer. When computing alpha, use alpha_scale

* tot_cwnd instead of tot_cwnd, and do all the operations in integer arithmetic.

Then, scale down the increase per ack by alpha_scale. The algorithm is:

o For each ack received on subflow i, increase cwnd_i by min ((alpha*bytes_acked*mss_i/tot_cwnd)/alpha_scale , bytes_acked*mss_i/cwnd_i)

Alpha scale denotes the precision we want for computing alpha. Observe that the errors in computing the numerator or the denominator in the formula for alpha are quite small, as the cwnd in bytes is typically much larger than the RTT (measured in ms).

With these changes, all the operations can be done using integer arithmetic. We propose alpha_scale be a small power of two, to allow using faster shift operations instead of multiplication and division. Our experiments show that using alpha_scale=512 works well in a wide range of scenarios. Increasing alpha_scale increases precision, but also increases the risk of overflow when computing alpha. Using 64bit operations would solve this issue. Another option is to dynamically adjust alpha_scale when computing alpha; in this way we avoid overflow and obtain maximum precision.

It is possible to implement the algorithm by calculating tot_cwnd on each ack, however this would be costly especially when the number of subflows is large. To avoid this overhead the implementation MAY choose to maintain a new per connection state variable called tot_cwnd. If it does so, the implementation will update tot_cwnd value whenever the individual subflows' windows are updated. Updating only requires one more addition or subtraction operation compared to the regular, per subflow congestion control code, so its performance impact should be minimal.

Computing alpha per ack is also costly. We propose alpha be a per connection variable, computed whenever there is a drop and once per RTT otherwise. More specifically, let cwnd_new be the new value of the congestion window after it is inflated or after a drop. Update alpha only if cwnd_i/mss_i != cwnd_new_i/mss_i.

In certain cases with small RTTs, computing alpha can still be expensive. We observe that if RTTs were constant, it is sufficient to compute alpha once per drop, as alpha does not change between drops (the insight here is that cwnd_i/cwnd_j = constant as long as both windows increase). Experimental results show that even if round trip times are not constant, using average round trip time instead of instantaneous round trip time gives good precision for computing

alpha. Hence, it is possible to compute alpha only once per drop according to the formula above, by replacing rtt_i with rtt_avg_i.

If using average round trip time, rtt_avg_i will be computed by sampling the rtt_i whenever the window can accomodate one more packet, i.e. when cwnd / mss < (cwnd+increase)/mss. The samples are averaged once per sawtooth into rtt_avg_i. This sampling ensures that there is no sampling bias for larger windows.

Given tot_cwnd and alpha, the congestion control algorithm is run for each subflow independently, with similar complexity to the standard TCP increase code [RFC5681].

4.1. Implementation Considerations when CWND is Expressed in Packets

When the congestion control algorithm maintains cwnd in packets rather than bytes, the algorithms above must change to take into account path mss.

To compute the increase when an ack is received, the implementation for multipath congestion control is a simple extension of the TCP New Reno code. In TCP New Reno cwnd_cnt is an additional state variable that tracks the number of segments acked since the last cwnd increment; cwnd is incremented only when cwnd_cnt > cwnd; then cwnd_cnt is set to 0.

In the multipath case, cwnd_cnt_i is maintained for each subflow as above, and cwnd_i is increased by 1 when cwnd_cnt_i > max(alpha_scale * tot_cwnd / alpha, cwnd_i).

When computing alpha for packet-based stacks, the errors in computing the terms in the denominator are larger (this is because cwnd is much smaller and rtt may be comparatively large). Let max be the index of the subflow used in the numerator. To reduce errors, it is easiest to move rtt_max (once calculated) from the numerator to the denominator, obtaining the equivalent formula below.

$$alpha = alpha\_scale * tot\_cwnd * \frac{cwnd\_max}{\left(\sum_i \frac{rtt\_max * cwnd\_i}{rtt\_i} \right)^2}$$

Note that the formula for computing alpha does not take into account path mss, and is the same for stacks that keep cwnd in bytes or packets. With this formula, the algorithm for computing alpha will match the rate of TCP on the best path in B/s for byte-oriented

stacks, and in packets/s in packet-based stacks. In practice, mss rarely changes between paths so this shouldn't be a problem.

However, it is simple to derive formulae allowing packet-based stacks to achieve byte rate fairness (and viceversa) if needed. In particular, for packet-based stacks wanting byte-rate fairness, the formula above changes as follows: cwnd_max is replaced by cwnd_max * mss_max * mss_max, while cwnd_i is replaced with cwnd_i * mss_i.

5. Discussion

To achieve perfect resource pooling, one must couple both increase and decrease of congestion windows across subflows, as in [KELLY]. Yet this tends to exhibit "flappiness": when the paths have similar levels of congestion, the congestion controller will tend to allocate all the window to one random subflow, and allocate zero window to the other subflows. The controller will perform random flips between these stable points. This doesn't seem desirable in general, and is particularly bad when the achieved rates depend on the RTT (as in the current Internet): in such a case, the resulting rate with fluctuate unpredictably depending on which state the controller is in, hence violating Goal 1.

By only coupling increases our proposal removes flappiness but also reduces the extent of resource pooling the protocol achieves. The algorithm will allocate window to the subflows such that p_i * cwnd_i = constant, for all i. Thus, when the loss rates of the subflows are equal, each subflow will get an equal window, removing flappiness. When the loss rates differ, progressively more window will be allocated to the flow with the lower loss rate. In contrast, perfect resource pooling requires that all the window should be allocated on the path with the lowest loss rate.

6. Security Considerations

None.

Detailed security analysis for the Multipath TCP protocol itself is included in [I-D.ford-mptcp-multiaddressed] and [REF]

7. Acknowledgements

We thank Christoph Paasch for his suggestions for computing alpha in packet-based stacks. The authors are supported by Trilogy (http://www.trilogy-project.org), a research project (ICT-216372)

partially funded by the European Community under its Seventh Framework Program. The views expressed here are those of the author(s) only. The European Commission is not liable for any use that may be made of the information in this document.

8. IANA Considerations

None.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

9.2. Informative References

[I-D.ford-mptcp-multiaddressed]
Ford, A., Raiciu, C., Handley, M., and S. Barre, "TCP Extensions for Multipath Operation with Multiple Addresses", draft-ford-mptcp-multiaddressed-01 (work in progress), July 2009.

[KELLY] Kelly, F. and T. Voice, "Stability of end-to-end algorithms for joint routing and rate control", ACM SIGCOMM CCR vol. 35 num. 2, pp. 5-12, 2005, <http://portal.acm.org/citation.cfm?id=1064415>.

[NSDI] Wischik, D., Raiciu, C., Greenhalgh, A., and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP", Usenix NSDI , March 2011, <http://www.cs.ucl.ac.uk/staff/c.raiciu/files/mptcp-nsdi.pdf>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC3465] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, February 2003.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

[WISCHIK] Wischik, D., Handley, M., and M. Bagnulo Braun, "The Resource Pooling Principle", ACM SIGCOMM CCR vol. 38 num. 5, pp. 47-52, October 2008,

<http://ccr.sigcomm.org/online/files/p47-handleyA4.pdf>.

Authors' Addresses

Costin Raiciu
University College London
Gower Street
London WC1E 6BT
UK

Email: c.raiciu@cs.ucl.ac.uk

Mark Handley
University College London
Gower Street
London WC1E 6BT
UK

Email: m.handley@cs.ucl.ac.uk

Damon Wischik
University College London
Gower Street
London WC1E 6BT
UK

Email: d.wischik@cs.ucl.ac.uk