Internet Engineering Task Force (IETF)
Request for Comments: 6356
Category: Experimental
ISSN: 2070-1721

C. Raiciu
Univ. Politehnica of Bucharest
M. Handly
D. Wischik
Univ. College London

October 2011

Coupled Congestion Control for Multipath Transport Protocols

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 standard TCP 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, achieving a property called "resource pooling" where a bundle of links effectively behaves like one shared link with bigger capacity. This would increase the overall efficiency of the network and also its robustness to failure.

This document presents a congestion control algorithm that couples the congestion control algorithms running on different subflows by linking their increase functions, and dynamically controls the overall aggressiveness 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

Copyright Notice

Table of Contents

1. Introduction

Multipath TCP (MPTCP, [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 standard TCP congestion control 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 were 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 the additive increase function of the subflows, and uses unmodified TCP 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 that of 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 (RTTs) and computes the target rate. Then, it adjusts the overall aggressiveness (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 (low versus high 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 that 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.

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

The algorithm presented here only extends standard TCP 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).

2. 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] .

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 standard TCP [RFC5681].

Let cwnd_i be the congestion window on the subflow i. Let cwnd_total 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 used by TCP), 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 4.2.

Our proposed "Linked Increases" algorithm MUST:

o For each ACK received on subflow i, increase cwnd_i by

$$\min\left(\frac{alpha * bytes\_acked * MSS\_i}{cwnd\_total}, \frac{bytes\_acked * MSS\_i}{cwnd\_i}\right) \quad (1)$$

The increase formula (1) 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 aggressiveness 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 idea of what the algorithm is trying to do, let's take the case where all the subflows have the same round-trip time and Maximum Segment Size (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/cwnd_total segments per RTT.

Note that, as in standard TCP, when cwnd_total 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 implementations use an approach similar to appropriate byte counting (ABC, [RFC3465]), where the bytes_acked variable records the number of bytes newly acknowledged. If this is not the case, bytes_acked SHOULD be set to MSS_i.

To compute cwnd_total, 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 (slow start threshold) value for flows in fast retransmit when computing cwnd_total. To cater to 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 occasion. Hence, alpha must be computed based on the observed properties of the paths.

The formula to compute alpha is:

$$\text{alpha} = \text{cwnd\_total} * \frac{\text{MAX}\ (\text{cwnd\_i}/\text{rtt\_i}^2)}{(\text{SUM}\ (\text{cwnd\_i}/\text{rtt\_i}))^2} \qquad (2)$$

Note:

MAX (x_i) means the maximum value for any possible value of i.

SUM (x_i) means the summation for all possible values of i.

The formula (2) 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

Equation (2) 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.

4.1. Computing "alpha" in Practice

Let alpha_scale be an integer. When computing alpha, use alpha_scale * cwnd_total instead of cwnd_total and do all the operations in integer arithmetic.

Then, scale down the increase per ACK by alpha_scale. The resulting algorithm is a simple change from Equation (1):

o For each ACK received on subflow i, increase cwnd_i by:

$$\min\left(\frac{alpha * bytes\_acked * MSS\_i}{alpha\_scale * cwnd\_total}, \frac{bytes\_acked * MSS\_i}{cwnd\_i}\right) \quad (3)$$

The alpha_scale parameter 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 64-bit 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 cwnd_total 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 "cwnd_total". If it does so, the implementation will update the cwnd_total value whenever the individual subflow’s 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 the quotient of cwnd_i/MSS_i differs from the quotient of 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 per sawtooth instead of instantaneous round-trip time (i.e., TCP's smoothed RTT estimator) gives good precision for computing alpha. Hence, it is possible to compute alpha only once per drop using a modified version of equation (2) where rtt_i is replaced 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 accommodate 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 cwnd_total and alpha, the congestion control algorithm is run for each subflow independently, with similar complexity to the standard TCP increase code [RFC5681].

4.2. 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 standard TCP code. In standard, TCP 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 * cwnd_total / 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, changing equation (2) to obtain the equivalent formula below.

(4)

$$\text{alpha} = \text{alpha\_scale} * \text{cwnd\_total} * \frac{\text{cwnd\_max}}{(\text{SUM}((\text{rtt\_max} * \text{cwnd\_i}) / \text{rtt\_i}))^2}$$

Note that the calculation of 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 vice versa) if needed. In particular, for packet-based stacks wanting byte-rate fairness, equation (4) 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

The algorithm we've presented fully achieves Goals 1 and 2, but does not achieve full resource pooling (Goal 3). Resource pooling requires that no traffic should be transferred on links with higher loss rates. To achieve perfect resource pooling, one must couple both increase and decrease of congestion windows across subflows, as in [KELLY].

There are a few problems with such a fully coupled controller. First, it will insufficiently probe paths with high loss rates and will fail to detect free capacity when it becomes available. Second, such controllers tend 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 probes high loss paths, detecting free capacity quicker. Our proposal does not suffer from flappiness but also achieves less resource pooling. 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 windows 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. Further details can be found in [NSDI].

6. Security Considerations

One security concern relates to what we call the traffic-shifting attack: on-path attackers can drop packets belonging to a multipath subflow, which, in turn, makes the path seem congested and will force the sender's congestion controller to avoid that path and push more data over alternate subflows.

The attacker's goal is to create congestion on the corresponding alternative paths. This behavior is entirely feasible but will only have minor effects: by design, the coupled congestion controller is less (or similarly) aggressive on any of its paths than a single TCP flow. Thus, the biggest effect this attack can have is to make a multipath subflow be as aggressive as a single TCP flow.

Another effect of the traffic-shifting attack is that the new path can monitor all the traffic, whereas before it could only see a subset of traffic. We believe that if privacy is needed, splitting traffic across multiple paths with MPTCP is not the right solution in the first place; end-to-end encryption should be used instead.

Besides the traffic-shifting attack mentioned above, the coupled congestion control algorithm defined in this document adds no other security considerations to those found in [MPTCP-MULTIADDRESSED] and [RFC6181]. Detailed security analysis for the Multipath TCP protocol itself is included in [MPTCP-MULTIADDRESSED] and [RFC6181].

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. References

8.1. Normative References

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

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

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

8.2. Informative References

[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>.

[MPTCP-MULTIADDRESSED] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", Work in Progress, July 2011.

[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>.

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

[RFC6181] Bagnulo, M., "Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6181, March 2011.

[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 Politehnica of Bucharest
Splaiul Independentei 313
Bucharest
Romania

EMail: costin.raiciu@cs.pub.ro

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