 Coupled Multipath-Aware Congestion Control
draft-raiciu-mptcp-congestion-00

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 April 22, 2010.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents in effect on the date of publication of this document (http://trustee.ietf.org/license-info).
Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Abstract

Often endpoints are connected by multiple paths, but communications are usually restricted to a single path per socket. Resource usage
within the network would be more efficient were it possible for these multiple paths to be used concurrently.

The use of multiple paths simultaneously, specifically within a Multipath TCP protocol, necessitates the development of new congestion control algorithms. If existing algorithms such as TCP New Reno were run independently on each path, the multipath flow would take more than its fair share if there was a common bottleneck. 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 utilization 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 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.
Table of Contents

1. Requirements Language ...................................... 4
2. Introduction .................................................. 4
3. Coupled Congestion Control Algorithm ....................... 5
4. Implementation Optimizations ............................... 7
   4.1. Implementation Considerations when CWND is Expressed in Packets ....................... 8
5. Discussion ..................................................... 8
6. Security Considerations ..................................... 9
7. Acknowledgements ............................................. 9
8. IANA Considerations ......................................... 9
9. References .................................................... 9
   9.1. Normative References .................................. 9
   9.2. Informative References ................................. 9
Authors’ Addresses .............................................. 10
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 allow 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 on any one of its paths than if it was a single path flow using only that route. 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, so our proposal always assumes there is a shared bottleneck and throttles the aggressiveness of the multipath flow such that its total throughput is no more than that of a regular TCP running on the best path available.

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.

It is foreseeable that different congestion controllers will be implemented for Multipath transport, each aiming to achieve different properties in the resource pooling/fairness/stability design space. In particular, solutions that give better resource pooling may be proposed. This algorithm is conservative from this point of view, sacrificing resource pooling for stability.

3. Coupled Congestion Control Algorithm

The algorithm we present only applies to the congestion avoidance state. The slow start, fast retransmit, and fast recovery algorithms are the same as [RFC5681].

Let cwnd_i be the congestion window on the subflow i, and assume there is always data to send. 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 drop probability, round trip time 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 is:

- For each ack received on subflow i, increase cwnd_i by min (ceil(alfa*bytes_acked*mss_i/tot_cwnd) , bytes_acked*mss_i/cwnd_i)

- For each drop event on subflow i, decrease set cwnd_i to max(cwnd_i/2,2). A drop event is one or more packet drops
experienced by a subflows in the same round trip time.

The decrease function is the same as in TCP New Reno, so we will not discuss it further in the remainder of this document.

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 a multipath subflow is NEVER more aggressive than a TCP flow, hence achieving goal 2 (do no harm).

ceil returns the smallest integer greater than or equal to its real-valued input. As long as bytes_acked is non-zero, the increase is non-zero. The increase rounds up the computed value; and it ensures that multipath does not suffer more from rounding errors than TCP (which it might, as tot_cwnd is always greater than individual cwnds). 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.

"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 rate a TCP flow would get if it ran on the best path.

The total throughput of a multipath flow depends on the value of alpha and the drop probabilities, 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 fastest 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 for each multipath flow, based on the observed properties of the paths.

The formula to compute alpha is:
\[
\alpha = \frac{\text{tot}\_\text{cwnd}}{\max_{i} \frac{\text{rtt}_i}{\text{cwnd}_i \times \text{mss}_i}} \times \frac{\sum_{i} \frac{\text{rtt}_i}{\text{cwnd}_i \times \text{mss}_i}}{2}
\]

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

4. Implementation Optimizations

It is possible to implement our algorithm by calculating \(\text{tot}\_\text{cwnd}\) on each ack, however this would be costly especially when the number of subflows is large. To avoid this overhead the implementation SHOULD maintain \(\text{tot}\_\text{cwnd}\) per connection, and MUST update its value when 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. Simply maintaining \(\alpha\) per connection does not help, as its value would still need to be updated per ack. If we assume RTTs are constant, it is sufficient to compute a once per drop, as a does not change between drops (the insight here is that \(\text{cwnd}_i/\text{cwnd}_j = \text{constant}\) 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, we recommend that \(\alpha\) SHOULD be computed once per drop according to the formula above, by replacing \(\text{rtt}_i\) with \(\text{rtt}\_\text{avg}_i\).

\(\text{rtt}\_\text{avg}_i\) is computed by sampling the \(\text{sr}\_\text{rtt}_i\) whenever the window can accommodate one more packet, i.e. when \(\text{cwnd} / \text{mss} < (\text{cwnd} + \text{increase}) / \text{mss}\). The samples are averaged once per sawtooth into \(\text{rtt}\_\text{avg}_i\).

This sampling ensures that there is no sampling bias for larger windows.

Given \(\text{tot}\_\text{cwnd}\) and \(\alpha\), the congestion control algorithm is run for each subflow independently. The window increase per ack is \(\alpha \times \text{mss} \times \text{bytes}\_\text{acked} / \text{tot}\_\text{cwnd}\). Compared to traditional increase code,
this would require floating point operations to be performed on each ack.

To avoid such costly operations, implementors SHOULD add a state variable to each subflow $\text{incr}_i = \text{mss}_i \times \text{mss}_i \times \text{alfa} / \text{tot_cwnd}$. When alfa or tot_cwnd changes, incr_i MUST be updated. Since the change is rare (once per sawtooth) the performance impact should be minimal. With incr_i properly set, the increase per ack becomes $\text{bytes_acked} \times \text{incr}_i / \text{mss}_i$. As this requires only integer multiplication, the overhead is comparable to existing implementations of TCP.

### 4.1. Implementation Considerations when CWND is Expressed in Packets

When the congestion control algorithm maintains cwnd in packets rates than bytes, the code to compute tot_cwnd remains unchanged.

To compute the rtt_avg_i, srtt will be sampled when cwnd_i is incremented.

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 records the number of bytes acked since the last cwnd increment. cwnd is incremented again only when cwnd_cnt > cwnd.

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 > tot_cwnd / alfa . To avoid costly floating point operations, the right hand side of the inequality can be stored as a per connection state variable that is updated only when tot_cwnd or alfa change.

### 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. points. This seems not 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 \( p_i \times cwnd_i = \text{constant}, \) for all \( i \). Thus, when the drop probabilities of the subflows are equals, each subflow will get an equal window, removing flappiness. When they are different, progressively more window will be allocated to the flow with the lower drop probability. In contrast, perfect resource pooling requires that all the window should be allocated on the path with the lowest drop probability.

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

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


9.2. Informative References

[I-D.ford-mptcp-multiaddressed]


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