Implementation and Evaluation of Coupled Congestion Control for Multipath TCP

Régel González Usach and Mirja Kühlewind

Institute of Communication Networks and Computer Engineering (IKR),
University of Stuttgart, Germany
regusach@ikr.uni-stuttgart.de,
mirja.kuehlewind@ikr.uni-stuttgart.de

Abstract. Multipath TCP (MPTCP) is an experimental protocol currently under standardization in the IETF. MPTCP allows to use multiple TCP connections for one data transmission if at least one of the endpoints is multi-homed. For example, this can be a mobile device with a Wifi and a 3G interface. It is assumed that the paths are disjoint or partly disjoint. As such these paths have different transmission characteristics, like speed or delay. With MPTCP the congestion control of each single TCP transmission is coupled in such a way that the transmission data is distributed over all subpaths depending on the load situation on each path. In this paper, we present our implementation of the MPTCP congestion control algorithm in the Linux kernel. We evaluated, based on simulations that use the real Linux kernel implementation, if the intended goals on resource pooling and sharing could be reached.

1 Introduction

Multipath TCP (MPTCP) is an experimental protocol currently under standardization in the IETF. [1] specifies the operation modes and protocol extensions needed for MPTCP. MPTCP extends the transport layer with additional functionality on top of TCP. At the lower layer one MPTCP connection looks like one or multiple TCP connections, while the upper layer maintains only one connection. We call all single TCP flows belonging to one MPTCP connection subflows of this MPTCP connection. MPTCP extends TCP by some optional TCP header fields. During the TCP handshake of the first initial TCP connection (or MPTCP subflow), two hosts can negotiate the use of MPTCP. Later, additional subflows can be added.

Today, many hosts are multi-homed if e.g. connected over multiple wireless interfaces to different networks like Wifi or 3G. Thus each interface gets a different IP address. Within the first initial subflow, these IP addresses can be announced using an MPTCP option and then be used later during this MPTCP connection to setup additional subflows. If at least one of the hosts is multi-homed, another subflow can be setup. The transmission data will be split over all subflows.

R. Szabó and A. Vidács (Eds.): EUNICE 2012, LNCS 7479, pp. 173–182, 2012.
© IFIP International Federation for Information Processing 2012

Usually, if the used paths are disjoint or partly disjoint, all subflows have different path characteristics. The question arises how to split the data over all available paths without getting restricted by the slowest one. In consequence, if more capacity is available on one of the paths, more data should to be sent on this path. This control is done by a coupling of the TCP congestion control of each subflow. The idea to use multiple paths for routing and also to realize the load balancing based on congestion control has been proposed earlier [2][3][4]. The MPTCP congestion control approach is based on this preliminary work.

Moreover, MPTCP aims to achieve equal resource pooling and an equal share over all available resources. Assume a scenario with two links that have the same capacity. Both links are used by one MPTCP connection and one other (single) TCP flow is present on each link. MPTCP aims to share all available resources equally between all (three) parties. Each connection (a single TCP flow or all subflows of the MPTCP connection together) should get $1/3$ of the sum capacity of all links. In this case, each single TCP flow should get $2/3$ of the capacity of one link and the MPTCP congestion will get $1/3$ on each of the two links.
This leads to three design goals for the MPTCP congestion control:

1. **Improve Throughput.** The throughput of all subflows of one multi-path connection should perform at least as well as a single TCP flow would on the best of the paths.
2. **Do No Harm.** All MPTCP subflows on one link should not take more capacity than a single TCP would get on this link.
3. **Balance Congestion.** A MPTCP connection should utilize each subflow dependent on the congestion on the path.

In this paper we implemented the proposed and in the IETF standardized MPTCP congestion control approach [5] in the Linux kernel. We investigated if the stated goals could be achieved in simulations using the kernel code. Up to now there only is an evaluation of the proposed MPTCP congestion control by the original authors available [6] [7] and a comparison with other multipath approaches in [8]. This paper provides an independent implementation and evaluation.

This paper is structured as follow: The next section gives an overview of the proposed Reno-based congestion control for MPTCP. Implementation challenges are described in Section 3. Afterwards, in Section 4, we evaluate our implementation based on simulations and show that the aspired capacity sharing goals in a multi-path scenario can mostly be achieved. Section 5 summarizes our results and gives an outlook on possible different further approaches on congestion control for MPTCP.

2 Semi-coupled Congestion Control for MPTCP

RFC 6356 [5] describes an approach for MPTCP congestion control based on the design principle of TCP Reno [9]. The approach leads to a smaller share of the link capacity than a TCP Reno flow would get for each subflow but, as explained,

aims for an equal resource sharing if the throughput of all subflows is summed up. To achieve the right resource sharing on each link, the increase in rate of each subflow is coupled such that it is slower (or equal) than the increase of a single TCP. In [6] the authors show that it is valuable to only couple the increase rate but keep the decrease events separate on each subflow. This does not allow a perfect resource pooling but avoids flappiness. With perfect resource pooling, usually all the traffic would be shifted to the best path. This has two problems. First, if there are two equal paths, the traffic would permanently *flap* from one path to the other. And second, if all traffic is shifted to one path, changes in the available bandwidth on the other slower path(s) are not recognized anymore. We present an investigation of these effects in Section 4.

In TCP congestion control, the congestion window (*cwnd*) gives the number of packets that are allowed to send within one Round-Trip Time (RTT). The congestion window can be increased whenever a new TCP acknowledgment (ACK) is received as the capacity appears to be sufficient. The window is decreased if the link is overloaded and loss is observed. As explained above, the proposed coupled congestion control for MPTCP is based on TCP Reno. TCP Reno can be described by the following operations:

Increase $cwnd$ by $1/cwnd$ for each received ACK
Decrease $cwnd$ to $max(cwnd - \frac{cwnd}{2}, 1)$ for each loss event

In fact, TCP Reno increases the window by one packet per RTT and halves the window if a loss occurs. A loss event is one or more losses within one RTT.

With the semi-coupled congestion control for MPTCP, the congestion window $cwnd_i$ of each subflow i is coupled by a factor α, which determines the aggressiveness and depends on the sum of the congestion windows of all subflows $cwnd_{total}$. This gives the following algorithm for the linked increase proposal:

Increase $cwnd_i$ by $\min(\frac{\alpha}{cwnd_{total}}, \frac{1}{cwnd_i})$ for each rec. ACK on subflow i
Decrease $cwnd_i$ to $max(cwnd_i - \frac{cwnd_i}{2}, 1)$ for each loss event on subflow i

This realizes the coupling of the increase while the decrease only depends on the congestion window of each subflow. The increase is limited by the maximum increase a TCP Reno flow would have. α is derived in [6] to be

$$\alpha = cwnd_{total} * \frac{\max_i(\frac{cwnd_i}{rtt_i^2})}{(\sum_i \frac{cwnd_i}{rtt_i})^2} \quad (1)$$

3 Implementation

In our Linux kernel implementation we did not implement the MPTCP operations nor the TCP protocol extension but only the coupled congestion control. For simplification in our simulations, we assumed that all TCP flows, which have been started by the same host, belong to one MPTCP connection without having additional protocol information exchange or negociation between the endhosts.

This leads to the same congestion control behavior than MPTCP but neglects the additional signaling overhead which has no influence when greedy sources are used. Of course, this approach cannot be used in a real operating system. But for our simulation, this approach was sufficient to evaluate the proposed congestion control approach.

Linux provides plugable kernel modules to load additional functionality at run time and also a specific interface for congestion control modules [10]. We realized the MPTCP congestion control as an own Linux kernel module. Additionally, we introduced a small number of state variables in the TCP stack which can be accessed by the congestion control procedure of each subflow. To include our implementation into a full implementation of the MPTCP protocol, additional functionality would be needed to maintain and initialize the MPTCP subflows.

In addition to the Linux congestion control module, we introduced two state variables which are needed by all subflows to calculate α and will be updated for each ACK that arrives at any subflow. All subflows are coupled by using the same α for the increase. The two state variables are called *mptcp_rate_sum* and *mptcp_crate_max* (compare equation 1).

We caculate the sum of the throughputs of all subflows by storing the last value within each subflow congestion control procedure separately and then on each ACK add the new value and subtract the old one from *mptcp_rate_sum*.

The for the maintenance of the maximum window divided by the squared RTT, we introduced a simple algorithm that approximates the maximum. This does simplify the implementation as only one value need to be maintained outside of each subflow congestion control procedure. Otherwise a list of variable length would need to be maintained depending on the number of subflows.

```
crate_curr = tp->snd_cwnd/srtt/srtt
if (crate_curr > tp->mptcp_crate_max)
   tp->mptcp_crate_max = crate_curr
else if (crate_old == tp->mptcp_crate_max)
   tp->mptcp_crate_max = crate_curr
crate_old = crate_curr
```

We update the maximum if the current value is larger than the *mptcp_crate_max*. If the value is smaller, we also update if the current *mptcp_crate_max* is equal to the previous value (*crate_old*) of this subflow. In this case, we assume that the subflow, which is currently updating the state, is the subflow with the largest throughput. And thus we also have to update the maximum if the throughput has decreased on this subflow. If another subflow is now the one with largest throughput, this will be updated with the next ACK that arrives on that subflow.

For the simulation we use the IKR SimLib, a event-driven Java-based simulation library [11]. We include the real network stack of the kernel code into the IKR SimLib by leveraging the Network Simulation Cradle framework [12] which also to convert the code such that it can be used in user space and thus in a simulation environment.

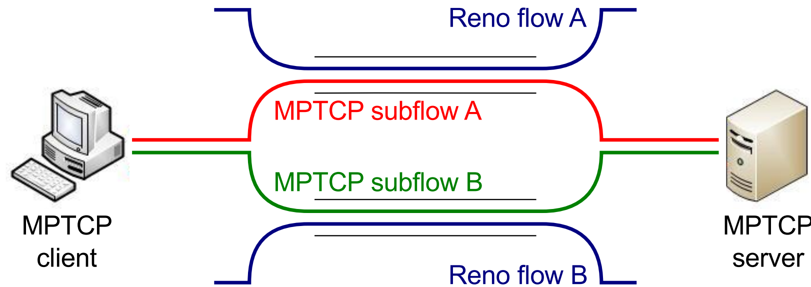

Fig. 1. Simulation scenario with two MPTCP subflows and cross traffic

4 Evaluation Results

4.1 Simulation Setup

For the evaluation we use a scenario with two links as shown in Figure 1. The MPTCP connection always has two subflows, one on each link. Each link can have one single TCP flow as cross traffic using Reno congestion control. In the first scenario, we assume each link sized with a capacity of 5 Mbps and the same base one-way delay (OWD) for all (sub)flows of 50 ms (thus a base RTT of 100ms). The buffers are always sized by the Bandwidth-Delay-Product (BDP). In the second scenario, we evaluate two links with different capacity but still the same base OWD; path A with 10 Mbps and path B with 5 Mbps. The third scenario, the links have an equal capacity of 5 Mbps but the Reno flow and the MPTCP subflow on path A have a base OWD of 50 ms while all flows using path B have a base OWD of 100 ms. We measure the *cwnd* and throughput at sender-side. As the bottleneck is later on the link, the instantaneous throughput at the sender can be slightly larger than the bottleneck capacity.

4.2 Understanding Coupled Congestion Control

First, we look at the scenarios without cross traffic to better understand the general behavior of the proposed semi-coupled Reno-based congestion control.

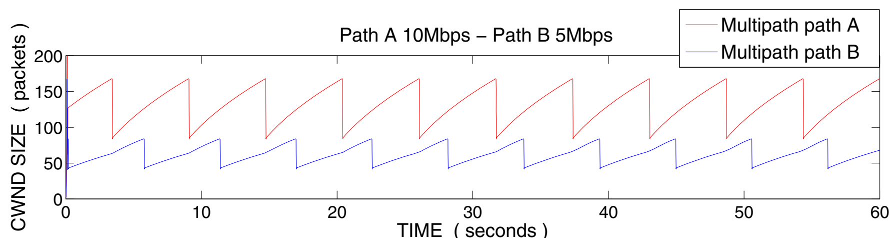

Fig. 2. Scenario 2: Different capacity on each link (no cross traffic)

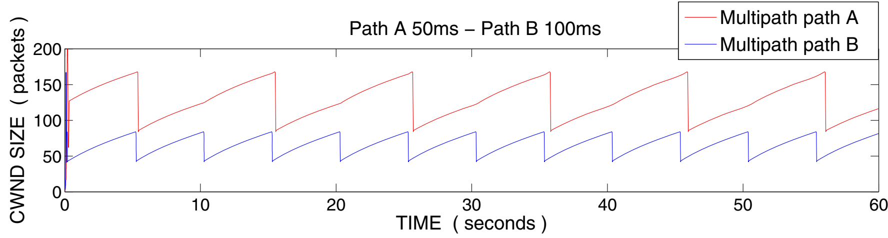

Fig. 3. Scenario 3: different base delay on each the path (no cross traffic)

Figure 2 shows the *cwnd* of each subflow for scenario 2 with different capacities but the same base delay. It can be seen that the slope of the increase is different. While TCP Reno would always increase by 1 packet per RTT, this is different for MPTCP linked increase. Here, we increase by $Inc = {}^{\alpha}/_{cwnd_{total}}$ per ACK. α, itself, contains the factor $cwnd_{total}$. Thus we have an increase of $Inc = {}^{\max_i(\frac{cwnd_i}{rtt_i^2})}/_{(\sum_i \frac{cwnd_i}{rtt_i})^2}$ per ACK. This is the same increase on each subflow per ACK (even independent of $cwnd_{total}$). This leads to an increase of $Inc{*}cwnd_i$ per RTT per subflow. Thus with an equal base RTT but different capacity, we

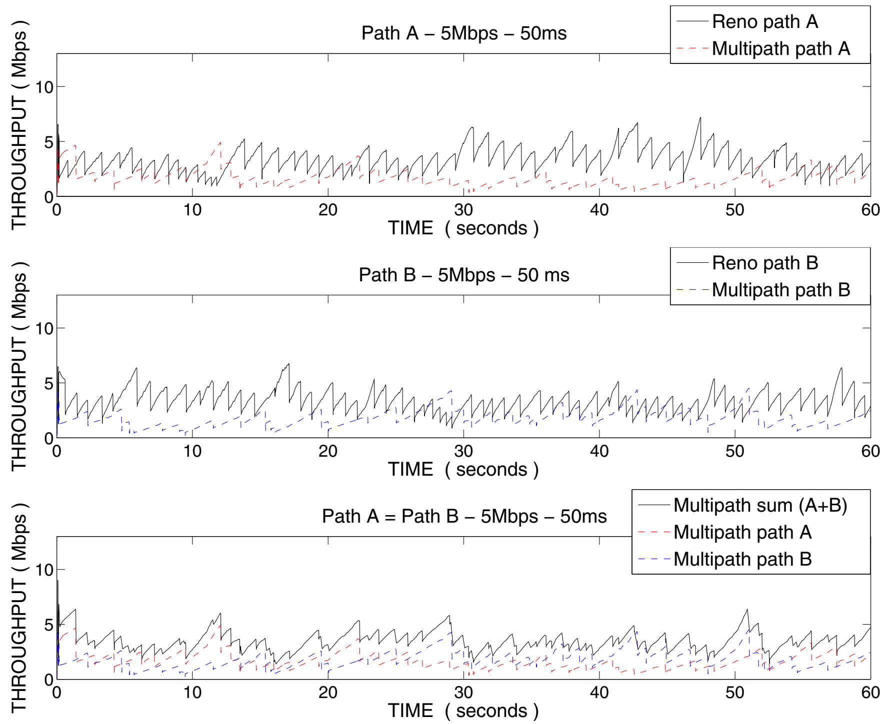

Fig. 4. Scenario 1: Links with same capacity and same base delay for all flows

have a different number of packets in the same time interval. That means we perform a different number of increases (excatly $cwnd_i$ ones) each of the same size of Inc. This leads to a different slope on each subflow for the increase of the congestion window over time.

In contrast, in Figure 3 we have the same capacity on both links but different base delays. We see the same slope for the *cwnd* increase on both paths as the proposed congestion control for MPTCP is design to be RTT-fair. Basically, the algorithm is designed as a rate control algorithm and is only mapped to an ACK-clocked mechanism by making α depended on $cwnd_{total}$. Moreover, it can be seen that, every time one of the subflows is decreasing, there is a bend in the curve of the other flow. This is because the sum rate has been decreased drastically at once and thus α refects that change as well.

4.3 Evaluation of MPTCP Design Goals

In Figure 4 we investigated scenario 1 with the same capacity on both links and same base delay for all (sub)flows but now with cross traffic on each link. MPTCP aims to share the sum capacity of both links equally. Thus we have to compare the throughput of each of the two (single) TCP flows on path A

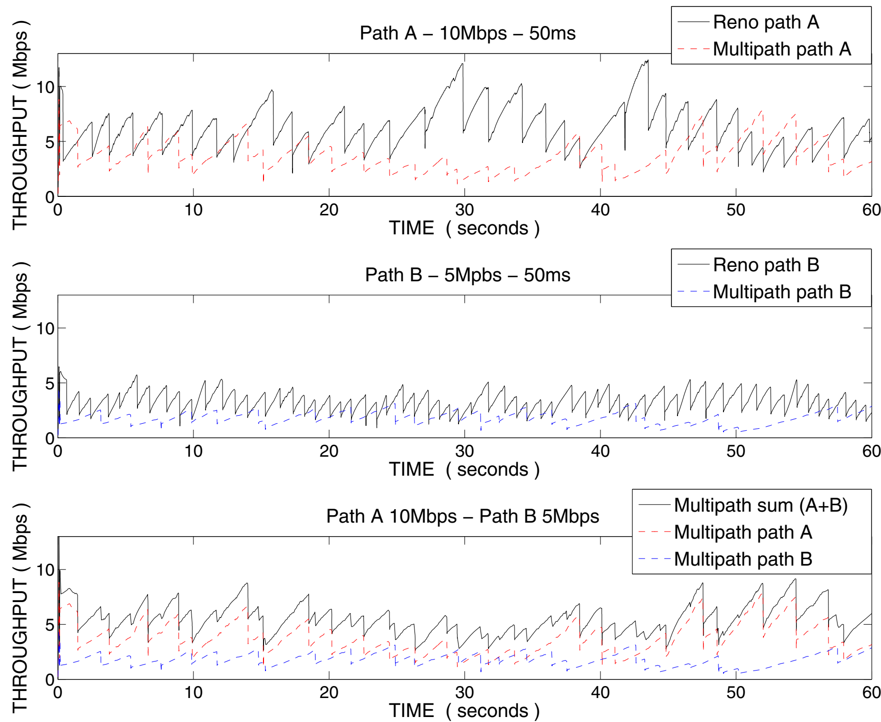

Fig. 5. Scenario 2: Different capacity on each link but same base delay for all flows

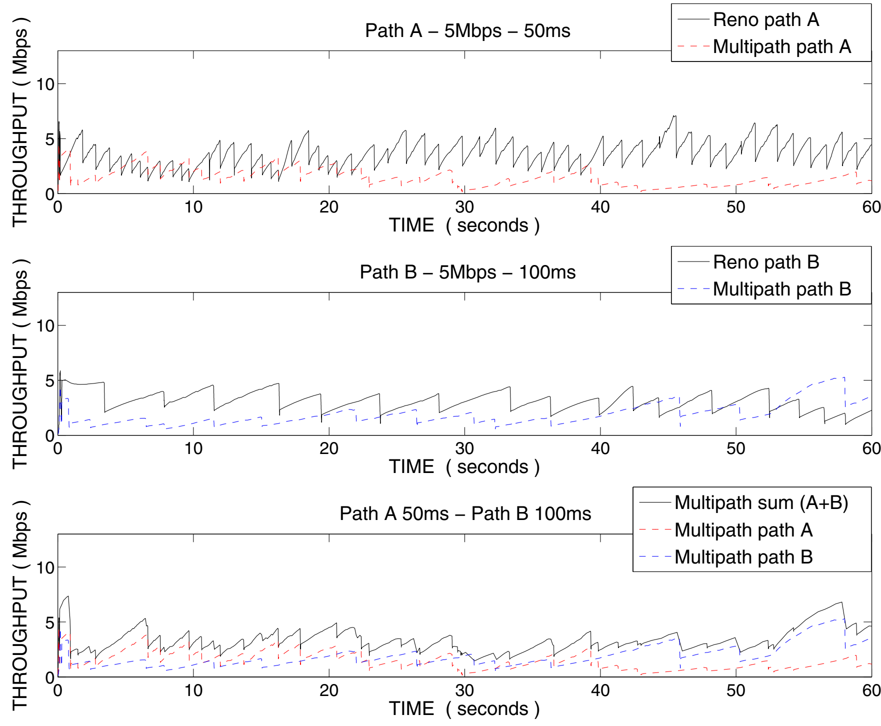

Fig. 6. Scenario 3: Links with same capacity but different base delay on each the path

and B with the sum throughput of both MPTCP subflows. These are the blacks lines in the diagrams. With a simulation time of 85 seconds all connections got similar average throughput but slightly higher for the MPTCP connection. This is 3.54 Mbps for the MPTCP connection, 3.13 Mbps for the single TCP connection on path A, and 3.24 Mbps for the single TCP on path B. By chance the MPTCP gets a higher rate than the single TCP. But at the same time the rate of the subflow on the other links decreases a little more. Thus there is still some flappiness between both subflows. But we can conclude that the long-time development of the rate is stable. In fact the semi-coupled MPTCP approach is not explicitly designed to achieve a perfect resources pooling. The observed behavior is driven by the goal to get at least ths same throughput as a single TPC would get on the best path.

Table 1. Means throuput in Mbps with single TCP cross traffic

	MPTCP sum	TCP path A	TCP path B
scenario 1	3.54	3.13	3.24
scenario 2	5.73	6.0	3.19
scenario 3	3.34	3.56	2.87

Scenario 2 with cross traffic, but different capacity on each link and same base delays for all flows, is displayed in Figure 5. The average throughput for the single TCP connection on path A is 6.0 Mbps while on path B the single TCP achieves an average throughput of 3.19 Mbps. The sum average throughput of MPTCP is 5.73 Mbps. In this scenario all MPTCP traffic would needed to be shifted to path A to achieve an equal share. This is not the case but it is also not intended as otherwise changes on path B could not be recognized anymore. Thus the equal sharing could not fully be achieved. But the goal to have a throughput that is not worse than it would be for a single TCP on the best link is fulfilled.

Next, we look at a scenario with again equal capacity but different base delay. The results are shown in Figure 6. Here, MPTCP has a sum average throughput of 3.34 Mbps which is about a third of the total capacity. The average throughput of the single TCP connection on path A is slightly higher with 3.56 Mbps. And thus the single TCP throughput on path B is slightly lower with 2.87 Mbps. As MPTCP compensates for RTT differences, it grabs slightly more capacity away from the link with higher base delay. In the diagrams above it can also be seen, that in all scenarios the throughput of a MPTCP subflow on any link is not larger that the throughput of the single TCP on that link. Table 1 shows an overview of the mean throughputs in each scenario (simulation time of 85 sec).

5 Conclusion and Outlook

In this paper, we presented an implementation and evaluation of the congestion control algorithm proposed for the use with MPTCP. The congestion control of all subflows is coupled in such a way that resource pooling of all used paths and an equally resource sharing between all connections using these paths should be achieved. To avoid flappiness the proposal couples only the rate increase of all subflows while each subflow will halve its congestion window independent of the others on a loss event. This does not allow perfect resource pooling anymore. MPTCP congestion control states three goals: To have at least an as good throughput as the best subflow would have, to not harm other (single) TCP flows, and to reach this by congestion balancing. We evaluated if these goals could be reached.

In all scenarios, MTCP achieves at least throughput as good as a single TCP would get on the best path and each subflow does not get more than a single TCP would get on a path. Thus MPTCP does not harm but always provides at least the best throughput one single connection could get. This is import to provide an incentive for deploying MPTCP. Moreover, we have shown that resource pooling has been implemented but perfect capacity sharing could not always be achieved. Thus MPTCP can help to distribute the traffic more equally in the Internet and to move the traffic away from the most congested links. In contrast using multiple TCP connection in parallel would still cause to overload the network, while the receiver would need to wait for the slowest transmission anyway. Moreover using multiple TCP connections is consider as unfair while MPTCP tries to maintain TCP-friendliness.

For the implementation of the MPTCP congestion control, we introduced a Linux kernel congestion control module and a small number of state variable in the TCP stack to realize the coupling. We demonstrated in easy-to-understand simulation scenarios the effects of designing MPTCP as a rate control algorithm which is mapped to an ACK-clocked mechanism for implementation purposes.

The proposed algorithm is a first approach to couple the congestion control of different MPTCP subflows and thus provide resource pooling. This approach is based on the design principles of TCP Reno. In future work, we aim to develop further approaches for the coupling which can be based on other congestion control algorithms than TCP Reno. Many more advanced mechanisms have been proposed for congestion control, like TCP Cubic, which is the default mechanism used in Linux, or maybe delay-based variants, like TCP Vegas. Currently, we were working on a coupled, delay-based congestion control for MPTCP. To evaluate our own algorithm, we will extend the simulation scenarios with e.g. more links and other rate differences and more dynamic cross traffic as well as cross traffic using different kind of congestion control algorithms.

References

[1] Ford, A., Raiciu, C., Handley, M., Bonaventure, O.: Tcp extensions for multipath operation with multiple addresses: draft-ietf-mptcp-multiaddressed-07. Internet-draft, IETF (March 2012)

[2] Kelly, F., Voice, T.: Stability of end-to-end algorithms for joint routing and rate control. Computer Communication Review 35, 5–12 (2005)

[3] Key, P., Massoulié, L., Towsley, D.: Combining multipath routing and congestion control for robustness. In: 40th IEEE Conference on Information Sciences and Systems, CISS (March 2006)

[4] Han, H., Shakkottai, S., Hollot, C.V., Srikant, R., Towsley, D.: Multi-path tcp: a joint congestion control and routing scheme to exploit path diversity in the internet. IEEE/ACM Trans. Netw. 14(6), 1260–1271 (2006)

[5] Raiciu, C., Handly, M., Wischik, D.: Coupled Congestion Control for Multipath Transport Protocols. RFC 6356, IETF (October 2011)

[6] Raiciu, C., Wischik, D., Handley, M.: Practical Congestion Control for Multipath Transport Protocols. Technical report, UCL (2010)

[7] Wischik, D., Raiciu, C., Greenhalgh, A., Handley, M.: Design, implementation and evaluation of congestion control for multipath TCP. In: Proceedings of USENIX Conference on Networked Systems Design and Implementation, NSDI (2011)

[8] Becke, M., Dreibholz, T., Adhari, H., Rathgeb, E.P.: On the Fairness of Transport Protocols in a Multi-Path Environment. In: Proceedings of IEEE International Conference on Communications, ICC (June 2012)

[9] Allman, M., Paxson, V., Blanton., E.: TCP Congestion Control. RFC 5681, IETF (September 2009)

[10] Sarolahti, P., Kuznetsov, A.: Congestion Control in Linux TCP. In: Proceedings of the FREENIX Track: USENIX Annual Technical Conference, pp. 49–62 (2002)

[11] IKR, University of Stuttgart: IKR Simulation and Emulation Library (2012), `http://www.ikr.uni-stuttgart.de/Content/IKRSimLib/`

[12] WAND Network Research Group: Network Simulation Cradle (2012), `http://research.wand.net.nz/software/nsc.php`