MulTFRC: Providing Weighted Fairness for Multimedia Applications (and others too!)

Dragana Damjanovic
Institute of Computer Science
University of Innsbruck, Austria
dragana.damjanovic@uibk.ac.at

Michael Welzl
Department of Informatics
University of Oslo
michawe@ifi.uio.no

ABSTRACT

When data transfers to or from a host happen in parallel, users do not always consider them to have the same importance. Ideally, a transport protocol should therefore allow its users to manipulate the fairness among flows in an almost arbitrary fashion. Since data transfers can also include real-time media streams which need to keep delay — and hence buffers — small, the protocol should also have a smooth sending rate. In an effort to satisfy the above requirements, we present MulTFRC, a congestion control mechanism which is based on the TCP-friendly Rate Control (TFRC) protocol. It emulates the behavior of a number of TFRC flows while maintaining a smooth sending rate. Our simulations and a real-life test demonstrate that MulTFRC performs significantly better than its competitors, potentially making it applicable in a broader range of settings than what TFRC is normally associated with.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Design, Algorithms, Experimentation

Keywords

transport protocols, TFRC, TCP-friendliness

1. INTRODUCTION

The TCP protocol is the most common reliable transport protocol in the Internet. After the inclusion of congestion control in the 80's, it has only undergone minor changes; congestion control in TCP made the Internet stable and has managed to keep it stable for all these years. On this basis, the notion of TCP-friendliness, which requires flows to not exceed the bandwidth of a conforming TCP running under comparable conditions, has been introduced. Applying TCP-friendliness to all data transfers on the Internet would ensure that all flows fairly share their bottleneck links and the network's stability is preserved.

There is, however, a flaw in this logic: data transfers are not always of the same importance, i.e. the benefit from a user's point of view is not necessary equal for each of them. Example scenarios include parallel file downloads, where a user is more eager to obtain file A than file B, and situations where a P2P application could disturb a streaming video. In a common home network setup, where the narrow-band access link (e.g., dial-up, DSL, etc.) is usually the bottleneck, the user's own flows are actually only competing with each other and not with "foreign" traffic. In such a situation, sharing the available bandwidth according to the user's needs is the only fairness notion that truly counts. While TCP-friendliness could theoretically be neglected in such a scenario, e.g. by controlling the sending rate with a simple queuing scheme at the access router, doing so would however be harmful when the bottleneck shifts, i.e. when congestion would happen in other parts of the network. In such a case, differentiated fairness on the bottleneck is about users competing with each other, where some users might be willing to pay more for getting a better quality of service.

A solution is to prioritize flows according to the user's preferences by making the sender's congestion control more or less aggressive. Since an aggregate of multiple TCP's is more aggressive than a single TCP, simply using several flows in parallel for important data transfers has become common. Users can manually initiate multiple transfers or use a download tool or application level protocol which provides such a capability, e.g. GridFTP [2]. This method has several disadvantages: data transfers must be multiplexed onto individual TCP flows (e.g. by offering large files which are split into several parts for easier parallel downloading). Parallel flows increase overhead, as state information must be kept for all the flows. Moreover, the granularity of aggression is limited. If we introduce the notion of "n-TCP-friendliness", i.e. being as TCP-friendly as n TCPs, then parallel TCPs require n to be a positive integer.

What would be needed is a n-TCP-friendly protocol which allows for a wide range of choices for n and exhibits a smooth sending rate, such that it can be used for a multitude of applications. We address this need by introducing MulTFRC, a protocol which, based on the arguably most common TCP-friendly real-time protocol TFRC [6], appears to realize n-TCP-friendliness with much greater flexibility and precision than any other protocol in the literature.

Like TFRC, MulTFRC is a rate based protocol. TFRC uses the steady-state throughput equation from [12] to derive the throughput of a TCP flow from measured network conditions. For MulTFRC, we developed a similar equation which yields the throughput of n parallel TCP flows. Since it does not really matter for (Mul)TFRC whether this equation is calculated on the sender or the receiver side, it is easy to use this protocol in situations where the prioritization is carried out by the receiver (e.g. for the file A vs. file B example discussed earlier). In the next section, we briefly

introduce the equation and show that it works accurately. Then we present the MulTFRC protocol in section 3, followed by an evaluation of its precision. Then we discuss and compare our protocol with related work and conclude.

2. THE EQUATION

The equation in [12] provides the sending rate of a TCP flow as a function of the round-trip time (RTT) and "loss events" that a flow experiences, where a loss event is defined as one or more packet loss occurrences during an RTT. For extending this equation to multiple TCPs, it could be a first thought to simply multiply the equation from [12] with n — but this does not work because loss event probability measurements of all n single flows would be needed. Since MulTFRC is in fact only one flow that just behaves like n flows, and since n should be a positive real number, it is not possible to distinguish which of these n "virtual" flows a packet belongs to, and it is therefore not possible to measure the loss event probability of these (non-existent) flows.

We developed a new equation by following the derivation in [12], but considering the throughput of n TCP flows depending on the loss event probability of the cumulative flow (denoted by p_e). In a cumulative flow's loss event, more than one of the n flows can experience a loss event. We assume that each lost packet belongs to a different flow, and therefore the number of flows affected by a loss event of the cumulative flow, denoted by j, is equal to the number of packets lost in the loss event. It can be calculated as $j = p_r/p_e$, where p_r is the packet loss probability. Due to lack of space we refer to [5] for a detailed derivation of the equation and only present the final result here, in the form of an algorithm for calculating the throughput of n flows. We also enclose some key results regarding the validation of its accuracy using simulations and real-life measurements.

In the algorithm, b is the number of packets acknowledged by an ACK, RTT is the round-trip time and T is the initial period of time (in a time-out phase) after which the sender retransmits unacknowledged packets. The rest of the variables are defined in the text above.

As in [12] we model TCP, first, assuming just triple-duplicate ACKs loss indications, and then we add the possibility that some flows are in time-out. The TCP congestion avoidance phase is observed in terms of rounds, i.e. in a round all flows send their current window size before the next round starts for all flows. Depending on the loss probability we calculate the number of rounds between two loss events (denoted by x in the algorithm) and the number of packets sent by all flows in that period ($1/pe$), which leads to tp, the rate of all flows without timeouts. To include timeouts, depending on the packet loss probability, we calculate the average number of flows that experience a timeout and are in the slow start phase (denoted by q in the algorithm), the average duration of a timeout period (z) and the average number of packets sent by a flow that is in the timeout phase (r). In the end the output of the algorithm is the throughput of n flows including timeouts: $((n-q)/n) * tp + q * (r/z)$.

ALGORITHM 1. *The throughput of n parallel flows [pkt/s]*
if $(j > n)$ { $j = n$ }
$x = jp_eb(2j-n) + \sqrt{(p_ebj(24n^2 + p_ebj(n-j)^2))/(6n^2p_e)}$
$tp = 1/(p_exRTT)$
$w = nx/(2b)(1+3n/j)$
$z = T(1+32p_e^2)/(1-p_e)$
$r = 1/(1-p_e)$
$q = (p_r/p_e)n/w$
if $(q > n)$ { $q = n$ }
if $(qz/(xRTT) \geq n)$ { $q = n$ }
else { $q = qz/(xRTT)$ }
return $(1-q/n)tp + q(r/z)$

With ns-2 simulations we showed that the equation works well in a broad range of conditions. Real background traffic can produce different distributions of loss events. These events include isolated packet losses and burst losses with variations of the length of the burst. All these loss distributions influence the throughput of an aggregate of TCP flows in a different way. We therefore validated the equation with uniformly distributed random loss as well as more bursty loss, which we produced by varying the capacity of a bottleneck link with a DropTail queue. For both sets of simulations, we used the common "dumbbell" topology and varied the loss percentage and, in the DropTail queue case, the bottleneck capacity. A number of parallel TCP flows were run, the throughput and the loss experienced by these flows was measured. We used the measured loss as an input parameter to calculate the throughput with our equation, and we compared the result with the measured throughput.

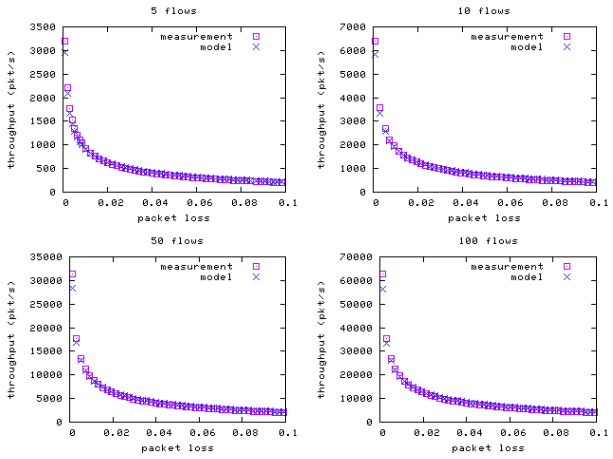


Figure 1: Equation vs. ns-2, RED + uniform loss

In the first scenarios, the access links had a bandwidth of 10 Gbit/s and a delay of 30 ms, whereas the bandwidth and delay of the bottleneck link were 1 Gbit/s and 1 ms, respectively. We added uniform random loss to the bottleneck link, where we additionally avoided phase effects by using a RED queue. According to the recommendation in [1], we allowed ns-2 to automatically configure the RED parameters. The amount of loss that was generated by the loss model on the bottleneck link is the parameter that we varied. Figure 1 shows that our equation yields a good estimate of the throughput of parallel TCP flows.

In the second set of simulations, a DropTail queue was used on the bottleneck link, where the queue length was set to the bandwidth × delay product. All links had a delay of 10 ms. The capacity of the access links was 1 Gbit/s. The bottleneck capacity was changed to cause a varying amount of loss. It had the values 1, 2, 4, 8, 16, 32, 64, 128 and 256 Mbit/s. Notably, using different bottleneck capacities influences the RTT (by 26.7% in the 1 Mbit/s case and 0.1% in the 256 Mbit/s case); this makes our result, shown in figure 2, somewhat similar to a real-life test where packet loss is a measured parameter.

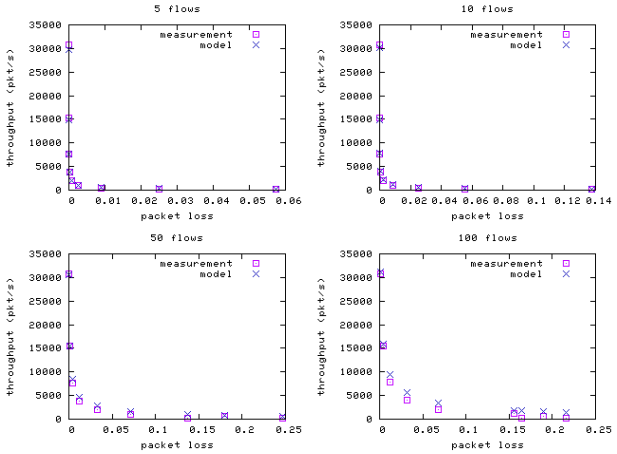
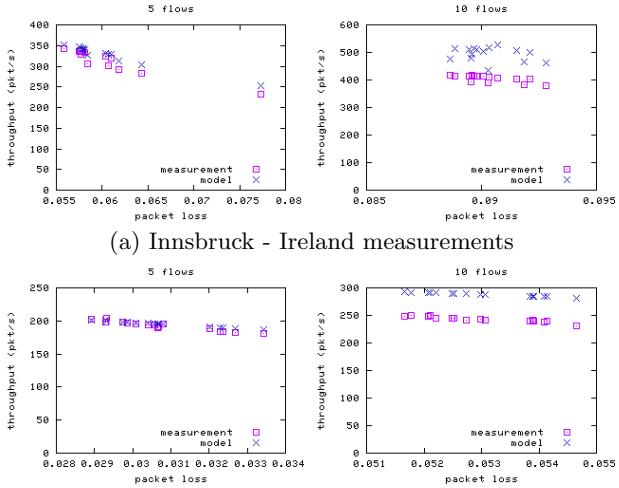


Figure 2: Equation vs. ns-2, DropTail queue



(a) Innsbruck - Ireland measurements

(b) Innsbruck - Texas measurements

Figure 3: Real-life measurements

We also validated our equation with real-life measurements. We measured the throughput of $n = 1..10$ connections between two hosts; all connections started at the same time. We sent data from our site in Austria to Ireland and Texas and used web100 (http://www.web100.org), on our site, for monitoring. Our host (Athlon64 3200+, 512 MB RAM, 100 Mbit network card) had a Linux kernel version 2.6.17.1 and web100 version 2.5.11. Because of almost no loss in the network we would have needed to transfer files of 1-2 GB to get sustained steady state TCP behavior. Therefore we set the network card to work with only 10 Mbit/s.

The host in Ireland (149.157.192.252) ran Linux and the following TCP parameters were set: window scaling was turned on, the advertised window scaling value was 12, and SACK was enabled. The measurements took from the 30th of May 2008 (16:17) till the 1st of June 2008 (17:28). We transmitted files of 70 MB using HTTP, opening n (1..10) uploads at the same time. For each number n, the measurement was run multiple times. Every transfer lasted at least 700 s (up to 1300 s).

The host in Texas (129.110.241.44) ran Linux and used the following TCP parameters: window scaling was turned on, the advertised window scaling value was 6, and SACK was enabled. We measured starting from the 9th of May 2008 (15:33) till the 13th of May 2008 (15:35). We performed the same set of HTTP file transfers as to Ireland. Each measurement took at least 800s (up to 2100s).

The loss event probability in our real-life tests was measured in the same way as in our simulations (not more than one loss event per RTT). Figure 3 shows that, with 5 flows, our equation yields a good estimate of the throughput. The deviation from the real-life measurements is larger with 10 flows, which we show here as a "worst case"; from our measurements it seems that this error does not generally grow with the number of flows and is bounded. We believe it to be due to the different loss patterns that occur as flows synchronize in the bottleneck queue, which may sometimes be a bad match for the assumptions that our equation is based upon. In general, our equation represents a trade-off between precision and ease of use, as there are several other, more precise yet significantly more complex models in the literature; this is elaborated upon in [5].

3. MULTFRC: DESIGN AND EVALUATION

Since the MulTFRC protocol is based on TFRC, many input parameters for the rate calculation are measured in the same way (RTT, T and b). The loss event probability is measured for the cumulative flow. It is obtained as in TFRC, counting just one loss event per RTT. To obtain a smooth rate, the TFRC protocol uses the weighted average of the last k loss intervals. In [6] and [7] k is set to 8; we use the same setting for MulTFRC.

In [5] the number of packets lost in a loss event is calculated using the approximation $j = p_r/p_e$ (the real loss probability divided by the loss event probability). Since it is possible to precisely count the number of lost packets in a loss event in the MulTFRC protocol, we use this as an input for j instead. The same method for sample discounting is used as for measuring the loss event probability.

Figure 4 illustrates that, because having a single n-TCP-friendly flow eliminates the potential of individual TFRC flows to harm each other, MulTFRC reacts faster to congestion than TFRC. The figure shows the throughput over time for a flow traversing a 15 Mbit/s, 20 ms RED bottleneck link with periodic loss from an ns-2 simulation. At the beginning, loss was 1%, at the 5th second it increased to 10% and at the 14th second it changed to 0.5%. To compare our protocol with TFRC, we ran five separate simulations: one TFRC flow, two TFRC flows at the same time, four TFRC flows at the same time, one MulTFRC flow with $n = 2$, and one MulTFRC flow with $n = 4$. As it can be seen the responsiveness of MulTFRC does not change as n increases.

Our equation assumes that each lost packet belongs to a different flow, which is rarely true in reality. As figure 5 shows, a MulTFRC flow with a large n is less affected by this assumption (additionally, MulTFRC gets slightly less throughput than TCP with $n > 100.0$ in this figure because of the exceedingly high loss rate in this extreme case (more than 15%)). To incorporate the possibility that more than one packet belongs to the same flow, we assume that a packet belongs to any of n flows with the same probability ($\frac{1}{n}$). The probability that a flow is not affected in a loss event

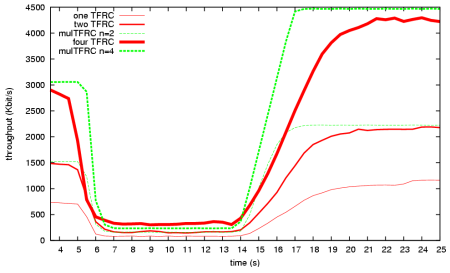


Figure 4: Dynamic behavior of MulTFRC

is equal to the probability that all j lost packets belong to the other $n-1$ flows: $((n-1)/n)^j$. The probability p that the flow is affected is $1-(1-1/n)^j$. Therefore the number of flows affected in a loss event with j lost packets is $n*p$. A comparison between the old and new j calculation is shown in figure 5, where MulTFRC and TCP flows shared a 32 Mbit/s, 20 ms RED bottleneck link.

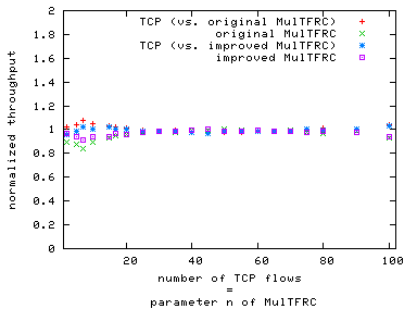
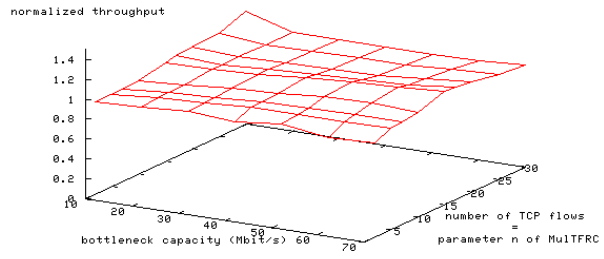


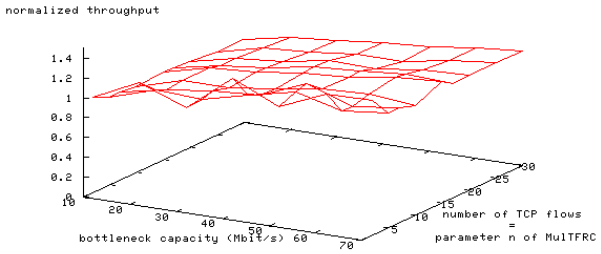
Figure 5: TCP-friendliness of MulTFRC with old and new calculation of j

We ran simulations with several network conditions including RED and DropTail queuing, based on the simulations used in [6], i.e. the same network parameters were used. We evaluated the MulTFRC protocol with a range of values for n, and in each simulation a MulTFRC flow was run together with several TCP flows in a dumbbell topology; the number of TCP flows was the same as MulTFRC's parameter n. Access links had a 100 Mbit/s capacity and 2 ms delay on the left side of the bottleneck link, and 1 ms on the right-hand side of the bottleneck link. The impact of MulTFRC on TCP with a RED queue is shown in figure 6 a). The slight increase at the top left corner of this figure (minimum bottleneck capacity, maximum number of flows) is due to the large loss rate of roughly 15%. We ran similar simulations with a DropTail queue on the bottleneck link. As figure 6 b) shows, MulTFRC does not significantly affect the throughput of TCP flows.

If $n \in \mathbb{N}$, the number of "virtual" flows affected in a loss event must be between 1 and n (in algorithm 1: $if(j > n)$ $then$ $j = n$). For $0 < n < 1$, deriving the correct number of these flows is not straightforward. We consider such a flow to be just like one flow that is less aggressive. With this assumption, the value for the number of flows affected in a loss event must be 1. In a general case where $n \in \mathbb{R}^+$ the idea is the same. For example, if $n = 1.4$, we can consider that we



a) RED queue at the bottleneck



b) DropTail queue at the bottleneck

Figure 6: TCP under the influence of MulTFRC

have two flows: one is a normal flow ($n = 1$), and the other one is just a less aggressive flow ($n = 0.4$). Therefore the upper limit for the number of affected flows can be obtained by rounding up ($\lceil n \rceil$) in the first line of algorithm 1.

To evaluate MulTFRC with $n \in \mathbb{R}^+$ we ran a MulTFRC flow with $0.1 < n < 2$ against a single TCP flow. Figure 7 shows results with a RED queue at the bottleneck link, which had a delay of 20 ms and a capacity of 4 Mbit/s. The behavior was similar in simulations with different bottleneck parameters.

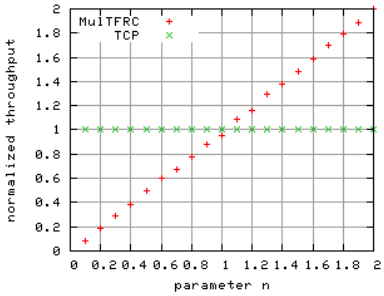


Figure 7: Changing n from 0.1 to 2

We also tested MulTFRC against standard TCP with a real-life implementation, extending the original code provided by the TFRC authors (http://www.icir.org/tfrc/code). The tests were run in a local testbed, all hosts run Linux kernel version 2.6.17.1. We used tc command to introduce a delay of 20 ms in both directions and to set the bandwidth to 32 Mbit/s. Figure 8 shows that the normalized throughput is also close to 1 in real-life tests.

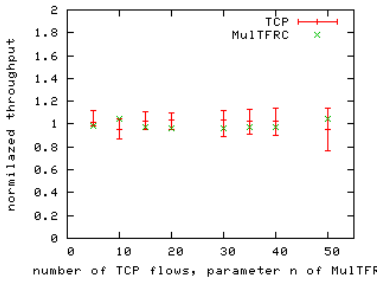


Figure 8: Real-life tests: MulTFRC vs. TCP

The Coordination Protocol (CP) is developed on the basis of TFRC, with the intention of behaving like n TFRC flows. The underlying idea is to multiply the equation from [12] with n in the protocol. As the authors of [11] have shown, and as we have argued in section 2, this is not enough because the loss event rate of a single flow share is not the same as the loss event rate of the cumulative flow. They therefore extended this concept by considering the n emulated flows as so-called "flow shares" and using a stochastic technique for filtering packets. The goal of this method is to determine which packets belong to a single flow share, and then use this flow share to calculate the loss event rate. For CP, it is explicitly assumed that n is always greater than 1; therefore this protocol would not work for $0 < n < 1$.

Because of using a stochastic technique, this protocol fluctuates more than ours. This is shown in figure 9, which was generated using the original code that was provided to us by the authors of CP. We used the same setup as the one that we used for testing smoothness in section 3; MulTFRC with $n = 4$ and CP with 4 flow shares traversed the same bottleneck link with periodic loss.

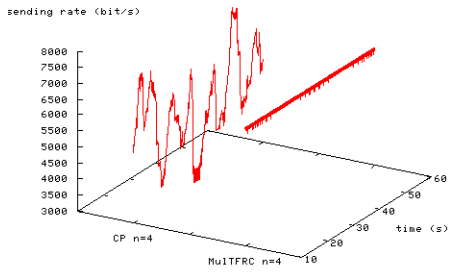


Figure 9: Smoothness of MulTFRC and CP

Since CP is based on simply multiplying the equation in TFRC with n, any error that the equation produces will be amplified as n grows. On the other hand, because of the nature of our equation from [5], our protocol works even better with an increasing number of flows. Figure 10 shows results of running CP against a number of TCPs and running MulTFRC against a number of TCPs with the simulation setup that we already used for figure 5. With CP, multiple runs of the same setup yield significantly different results.

We tested MulTCP [4] in the same simulation setup as CP, using an update that we made to the code that is available

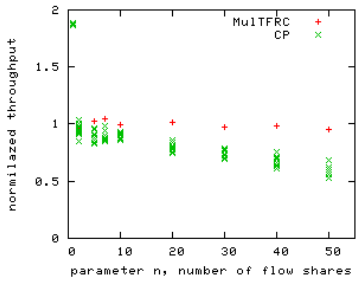


Figure 10: MulTFRC vs. TCP and CP vs. TCP with a larger number of flow shares

under "contributed code" via the main ns-2 webpage to make it work with the most recent version of ns-2. As stated in [4] and as our comparison shows, MulTCP performs reasonably well with values up to $n = 10$. This is shown in figure 11. This figure was generated using the same simulation setup as for figure 9, we are just zooming into the relevant range. The figure also shows Stochastic TCP [8], which is another protocol that we tested using the code provided by the authors. This protocol, which partitions a single congestion window into a set of "virtual streams" that are stochastically managed as individual TCP streams, was built for high-speed networks; in our simulations with a smaller bandwidth its performance was poor.

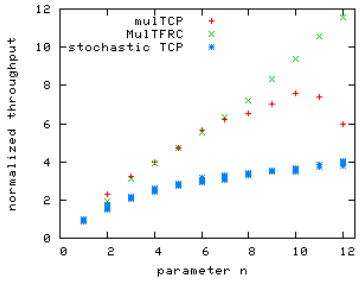


Figure 11: MulTFRC, MulTCP and Stochastic TCP

4. RELATED WORK

We already discussed a few protocols which are related to ours: MulTCP, Stochastic TCP and CP. Like CP, Probe-Aided MulTCP (PA-MulTCP) [9] was proposed as a means to control aggregates of end-to-end flows between two intermediate network nodes. As its name suggests, it is based on MulTCP, and improves upon it by using probes. The goal of these probes is to determine a loss rate which is closer to the loss rate experienced by a real TCP than the loss rate of the original MulTCP. The loss rate that is normally experienced by MulTCP is smaller than the loss rate of n standard TCP flows, making MulTCP too aggressive. The probe based loss rate measurement effectively adds another control loop to the protocol, which is used to impose an upper limit on the window size of the already existing (MulTCP-like) one. We did not run simulations with PA-MulTCP because the original code was not available and because its applicability is limited by the overhead of using an additional control loop. MulTFRC, on the other hand, was designed to be a general purpose protocol with a broad range of usage scenarios.

MPAT [14] is another mechanism that is concerned with controlling aggregates. Unlike PA-MulTCP and CP, however, MPAT maintains the control loops of individual TCP flows and lets them share their congestion state. In the example in [14], if two TCP flows would be allowed to send 5 packets each but the bandwidth should be apportioned according to a 4:1 ratio, MPAT could allow one flow to send 8 packets and let the other flow send 2. Such differentiation between flows is a way to provide Quality of Service via congestion control, and this is also done in OverQoS [15], albeit with MulTCP. While we believe that MulTFRC could also be used for this purpose, we consider a discussion of such usage to be beyond the scope of this paper.

On the extreme end of "friendliness" towards the network, several mechanisms for so-called "less-than best effort" behavior (which is the same as our $0 < n < 1$ case, but with a possibly unkown value of n under this constraint) have been proposed. TCP Nice [16], for example, is a variant of TCP Vegas [3] with a more conservative congestion control behavior. Just like these two algorithms, TCP Low Priority (TCP-LP) [10] relies on increasing delay as an indicator of imminent congestion, and defines a rate control method that lets the end system back off earlier than standard TCP.

Since increasing delay is an effect that can be measured earlier than packet loss or ECN marking (which are the normal congestion indicators of standard TCP), relying on it is a common theme in such work. This is not the case for MulTFRC with $0 < n < 1$. Therefore, our protocol is likely to be more aggressive than these alternatives if they share a bottleneck at the same time; on the positive side, this aggression is tunable in MulTFRC.

5. CONCLUSION

We have introduced MulTFRC, a protocol which realizes n-TCP-friendliness with $n \in \mathbb{R}^+$ while showing a smooth sending rate. Simulations and real-life experiments (which were not carried out with our "best competitor", CP) showed very encouraging results, indicating that MulTFRC could indeed be used as the general purpose protocol that we intended it to be. In order to support this claim, we plan to carry out Internet experiments with our real-life implementation, which we are currently refining to incorporate retransmissions of lost packets. This is necessary because, other than TFRC, our protocol should be applicable for non-real-time data transfers too.

Another item on our agenda for future work is related to the dynamic behavior of TFRC itself. The authors of [13] state that there can be an imbalance in the long-term throughput of TFRC and TCP. This is mainly attributed to the fact that the loss rate which is measured by TFRC flows can be different from the loss rate which is measured by TCP flows. Since this problem is very similar to the problem of MulTCP that is fixed in PA-MulTCP [9], it seems to be logical that the PA-MulTCP approach of adding an additional control loop could also be applied to TFRC and MulTFRC. While it was not our goal to fix an inherent problem of TFRC up to now, doing so is in our interest because MulTFRC naturally inherits such properties of TFRC. We therefore plan to investigate this possibility as a next step. The ns2 MulTFRC code is available at http://dps.uibk.ac.at/~dragana/mulTFRC.html.

Acknowledgments

This work was partially funded by the EU IST project EC-GIN under the contract STREP FP6-2006-IST- 045256. We would also like to thank David E. Ott and Thomas J. Hacker for providing us with their simulation code, Andrea Fumagalli and Doug Leith for letting us access their sites for measurements, and Grenville Armitage for useful feedback.

6. REFERENCES

[1] RED parameters: http://icir.org/floyd/red.html#parameters.

[2] W. Allcock. GridFTP: Protocol extensions to FTP for the grid. Technical report, Open Grid Forum, 2003.

[3] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *ACM SIGCOMM '94*, pages 24–35.

[4] J. Crowcroft and P. Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *SIGCOMM Comput. Commun. Rev.*, 28(3):53–69, 1998.

[5] D. Damjanovic, M. Welzl, M. Telek, and W. Heiss. Extending the TCP Steady-State Throughput Equation for Parallel TCP Flows. Technical Report 2, University of Innsbruck, Institute of Computer Science, DPS NSG Technical Report, August 2008. http://dps.uibk.ac.at/~dragana/mulEQ-TechRep.pdf

[6] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *SIGCOMM '00*, pages 43–56.

[7] S. Floyd, M. Handley, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348 (Proposed Standard), Sept. 2008.

[8] T. J. Hacker and P. Smith. Stochastic TCP: A Statistical Approach to Congestion Avoidance. In *PFLDnet 2008*, Manchester, GB.

[9] F.-C. Kuo and X. Fu. Probe-Aided MulTCP: an aggregate congestion control mechanism. *SIGCOMM Comput. Commun. Rev.*, 38(1):17–28, 2008.

[10] A. Kuzmanovic and E. W. Knightly. TCP-LP: low-priority service via end-point congestion control. *IEEE/ACM Trans. Netw.*, 14(4):739–752, 2006.

[11] D. E. Ott, T. Sparks, and K. Mayer-Patel. Aggregate Congestion Control for Distributed Multimedia Applications. In *IEEE Infocom 2004*, Hong Kong.

[12] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: a simple model and its empirical validation. In *SIGCOMM '98*, pages 303–314. ACM Press, 1998.

[13] I. Rhee and L. Xu. Limitations of equation-based congestion control. In *SIGCOMM '05*, pages 49–60, New York, NY, USA, 2005.

[14] M. Singh, P. Pradhan, and P. Francis. MPAT: aggregate TCP congestion management as a building block for Internet QoS. In *ICNP 2004*, pages 129–138, Berlin, Germany.

[15] L. Subramanian, I. Stoica, H. Balakrishnan, and R. H. Katz. OverQoS: an overlay based architecture for enhancing Internet QoS. In *NSDI'04*, Berkeley, USA.

[16] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: a mechanism for background transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, 2002.