Using DCCP: Issues and Improvements

Michael Schier  
University of Innsbruck, Austria  
michael.schier@uibk.ac.at

Michael Welzl  
University of Oslo, Norway  
michawe@ifi.uio.no

Abstract—The Datagram Congestion Control Protocol (DCCP) is no longer too young to be usable: the first RFCs were published in 2006, and a stable and quite complete Linux implementation exists. DCCP over UDP has also recently been specified to address network traversal problems. But how good is the service provided to applications by this protocol? This paper identifies some deficiencies of the current implementation—the lack of transparency in the API with regard to packet loss, the coarse granularity of the lookup table used to calculate the TFRC equation, and the lack of history discounting in CCID-3—and demonstrates that they can significantly impair the performance of typical DCCP use cases such as live video streaming. Solutions are proposed to tackle all these problems, and it is shown that they considerably improve the performance and the flexibility of applications.

I. INTRODUCTION

DCCP was designed to provide tailored support for time-critical applications such as voice chat and video conferencing, which prefer timeliness over reliable delivery [1]. Although it has already been some years since its IETF standardization back in 2006 [2], DCCP is still hardly used by potential applications. This can be partially attributed to the fact that most popular operating systems (OS) still do not natively support DCCP, and that NAT-traversal can still not be guaranteed. Recently, a short term solution to that misery was proposed, which encapsulates DCCP by UDP [3].

The session description protocol (SDP) is used at connection startup to probe whether end points either natively support DCCP or whether to fall back on a user-level DCCP-over-UDP implementation. Although this workaround appears to be promising as transition mechanism because it enables support for DCCP in devices that support UDP but do not yet natively support DCCP, as long as most major OS players do not incorporate DCCP into their protocol stacks, the protocol’s long-term success is highly questionable. For Linux, there exists a pretty stable kernel implementation of DCCP1 that covers almost all mandatory functionality of the respective RFCs, and although there is still much work in progress with respect to experimental congestion control algorithms (CCIDs), the core module, paired with CCID-2 (TCP-like congestion control) or CCID-3 (TCP-friendly rate control (TFRC)), is fully operational. A good starting point for C and Python application developers is [4].

We have carried out research on what we believe to be a typical use case for DCCP: live video streaming, i.e. streaming with tight timing constraints, which means that bandwidth fluctuations (which would have to be compensated for by buffering) should be minimized. In particular, we have developed a selective ARQ method that can significantly improve the video quality in the face of packet loss (details can be found in [5]), and tested it over DCCP in a local testbed. In this context, we thoroughly reviewed DCCP’s native Linux implementation from a user perspective with a special emphasis on CCID-3, conducted several experiments under diverse channel conditions, discovered some limitations, identified potential for improvement, and extended the implementation accordingly. We believe that our findings are not only useful in connection with this specific implementation, but that they are also of immediate benefit for future OS-level implementations (hopefully) to come as they will certainly consider the Linux DCCP code as primary reference.

In particular, we extended the socket API to enable applications to learn which packets were successfully transmitted and which ones were lost (Section II), thus providing the foundations of beneficial use cases such as supporting selective reliability. In addition to that, we found problems with the implementation of TFRC, which is probably the most important congestion control mechanism available in DCCP. TFRC derives its sending rate from the well-known TCP steady-state equation, which it uses to calculate how much a TCP sender would send under the current network conditions [6]. This yields a behavior that is TCP-friendly while exhibiting a “smoother” overall behavior (i.e. less rate fluctuations), which is deemed beneficial to time-critical applications that DCCP is trying to support.

First, we have found that the current design of the TFRC equation calculation gets increasingly inaccurate when a connection’s loss event rate decreases beyond some point. As a consequence, we propose to replace the linear lookup table that is currently in place with a non-linear lookup strategy that adapts the samples’ granularities based on their impact on the TFRC equation (Section III). Second, we have implemented the history discounting extension, which was proposed to improve rate recovery during error-free phases, yet is not currently included in the Linux DCCP implementation. We experimentally verify its benefits, and show its positive impact in connection with the live video streaming use case. Furthermore, we show that by empowering a user application to control the number of weights involved in the loss event calculation process, that application is able to adapt, based on that application’s specific needs, a connection’s degree of agility/responsiveness (Section IV). In Section V, related work

---

1http://www.linuxfoundation.org/collaborate/workgroups/networking/dccp
is discussed that is either directly connected to the modifications proposed in this paper, or that discusses discovered problems of TFRC with regard to fairness and responsiveness. Finally, in Section VI, we conclude by recapitulating the major improvements proposed in this paper.

II. IMPROVING TRANSPORT LAYER TRANSPARENCY

From an application’s perspective, using TCP or UDP is straightforward: after connection establishment (needed by TCP), an application uses the socket API provided by the OS to send and receive data without the need to be aware of what is actually going on at lower layers. This isolation makes perfect sense for TCP as this protocol guarantees reliable delivery, and information about occasional packet loss is therefore of little interest for upper layers. Similarly, the UDP functionality at the transport layer does not hold any information worthwhile to be provided to respective applications because a UDP sender is completely unaware of what happens to datagrams along the delivery path due to the absence of feedback. In contrast to that, providing applications that use DCCP with the opportunity to query whether packets have successfully made it to the receiver or not via the socket API is both technically feasible and reasonable; especially multimedia applications can profit from knowledge of lost data and react accordingly.

Currently, there exist several implementations of DCCP, most of which are user-level implementations and/or their development was discontinued. The most advanced implementation will be referred to throughout the remainder of this article has been integrated in the official Linux kernel development tree and has frequently been extended since version 2.6.14. Unfortunately, none of the kernel implementations allow programs that run in user space to learn whether a packet has successfully been received by the other endpoint or not. Therefore, we propose to extend the existing socket API (see Fig. 1) of the Linux DCCP implementation by additional functionality that allows applications to query the sequence numbers of outgoing as well as of acknowledged packets. Thus, sender-side applications are enabled to deduce which data segments were reported as lost by the receiver side. More specifically, we added the two getsockopt() parameters DCCP_SOCKOPT_TX_SEQ and DCCP_SOCKOPT_TX_ACK which return up to \( n \) of the most recently sent and acknowledged sequence numbers. Furthermore, the setsockopt() parameter DCCP_SOCKOPT_SEQTRACK was used to enable or disable the internal sequence number tracking and to adjust the parameter \( n \), which is the size of the kernel-space ring buffer used for storing the sequence numbers. \( n = 0 \) is the default setting, which is implicitly used by applications that do not make use of the sequence number tracking extension, i.e., the tracking overhead is avoided.

With these modifications in place, all that remains to be done when implementing a DCCP application is to frequently poll the sequence number information and to map these numbers to the corresponding data ranges. When performing selective retransmissions, it is important to note that DCCP sequence numbers cannot be reused, and that a fresh sequence number is always assigned to retransmitted content. Moreover, it may happen that acknowledgements are delayed and that they may therefore arrive after the respective retransmission event. Consequently, the mapping from DCCP sequence numbers to data segments should be implemented in such a way that surjective associations are supported. With regard to ease of deployment, we underline that all modifications of this section are limited to the sender side, i.e., the kernel functionality of the receiving system remains untouched, which substantially facilitates the realization of the sequence number tracking extension. In particular, the DCCP specification is not violated as the sequence number tracking patch is exclusively implementation specific.

Up to this point, we have assumed that receiver feedback (acknowledgements) is reliable and covers the entire sequence number range that was used. This condition holds for CCID-2, DCCP’s TCP-like congestion control algorithm [7], because the receiver periodically sends ACK vectors that in turn get acknowledged by the sender. This is crucial as it both helps the receiver to clean up old ACK vector space (thus reducing the size of future ACK vectors) and it enables the window-based congestion control mechanism to function properly because the current window size can be calculated. In contrast to that, CCID-3, DCCP’s TCP-friendly rate control (TFRC) algorithm [8], [9], does not use ACK vectors per default. Incoming acknowledgements mandatorily contain the elapsed time option (indicating the time between the reception of the last packet and sending the acknowledgement), the receive rate option (the rate measured at the receiver), and the loss intervals option (to signal the most recent loss intervals). Feedback does not need to be reliable as no congestion window is maintained, and the absence of feedback is penalized by halving the maximum sending rate.
In order to realize consistent sequence number tracking with CCID-3, there are two options: either enable ACK vectors via feature negotiation or implement immediate ACKs. The first solution is straightforward but is currently not feasible as the CCID-3 Linux implementation does not support ACK vectors. We considered adding this functionality but finally decided to use immediate ACKs as, although rather being a workaround, this was much easier to implement. It can be enabled by using the `netem` development, the test systems at node 1 and node 4 were em-

 extended DCCP eth0
 node 1 (sender) virtual machine Linux 3.0.0

 node 2
 Netem insertion of loss and delay

 node 3
 network switch

 node 4 (receiver) virtual machine Linux 3.0.0 extended DCCP

 Fig. 2. Testbed setup. Both intermediate nodes act as Layer-2 bridges and there is only one IP net involved. All links are wired and run at 100 Mbit/s.

 the diagrams. Moreover, the transmission rates never exceed 6 MB/s and 1.2 MB/s although no losses occur towards the end of the transmissions, i.e., the bitrate should actually keep on increasing until the channel capacity of approximately 12 MB/s is reached.

 The step effect clearly contradicts the intention of Kohler et al. [1] to design DCCP as a multimedia friendly protocol characterized by smooth rate changes. The reason for the effect can be found in the implementation where a part of the TFRC equation was realized as lookup table. More specifically, the simplified version of the TFRC equation from [9] where the Retransmit Timeout (RTO) is set to $1 \cdot RTT$ and $b$, the number of packets acknowledged by an ACK, is set to 1, was split into $X_{Bps} = \frac{s}{R(p) \cdot RTT}$ with $f(p)$ given in Equ. 1:

 $f(p) = \sqrt{2/3 \cdot p + 12 \cdot \sqrt{3/8 \cdot p \cdot (p + 32 \cdot p^3)}}$  

 As in the original equation, $p$ is the “loss event rate” (defined in [9]) and $s$ is the packet size. Using a lookup table for $f(p)$ is essential to avoid severe rounding errors as floating point arithmetic is prohibited for code running in kernel space. The current implementation contains two lookup tables, each having 500 entries, that cover the ranges $[0.2\%; 100\%]$ and $[0.01\%; 5\%]$ with even distances between the sampling points, i.e., the resolutions are 0.2% and 0.01% respectively. Assuming an average packet size of 1000 B and a $RTT$ of 100 ms analogous to the lower plot of Fig. 3, a sample calculation where $p$ is 0.05% and 0.04% (i.e., two neighboring

 III. SMOOTHING OUT BITRATE JUMPS

 To test the functionality of the Linux DCCP implementation and the extensions proposed in this paper, a test environment was used as depicted in Fig. 2. To facilitate kernel-module development, the test systems at node 1 and node 4 were em-

 embedded in virtual machines that physically shared the network adapter with its host system. Node 2 was used to introduce artificial delay and loss using the queuing policy `netem` [10]; the network adapters of node 2 were bridged. Node 3 was a 5-port switch, which had no special purpose during the test runs; it solely acted as gateway to enable software deployment from an external network.

 When using CCID-3, bitrate measurements revealed that, as expected, the send rate largely correlates with the results obtained from the TFRC equation. However, at high bitrates when the loss event rate gets small ($\lesssim 0.2\%$), bitrate changes become increasingly unsteady. This can be seen in Fig. 3, which depicts single transmissions from node 1 to node 4 with packet loss probabilities of 1% and 0% before and after the points denoted by “recovery”. A “step effect” (abrupt, choppy bitrate changes) can be observed in the right part of

 Fig. 3. The coarse granularity of the TFRC lookup table leads to bitrate jumps at small loss event rates (i.e., high bitrates). The measurements were done using an average packet size of 1000 B and average RTTs of 20 ms (upper diagram) and 100 ms (lower diagram) with deviations of at most 5%.
entries in the lookup table) yields bitrates of 545 kB/s and 610 kB/s; at such low values of \( p \), the implementation prohibits smooth bitrate transitions due to the low table resolution.

To tackle that problem, we propose to replace the current lookup table with a non-linear table in such a way that the sampling points are more dense at regions that are more likely to be queried. As an example, packet loss rates of around 1% are certainly more likely to occur than rates above 50%. Moreover, reducing the density of sampling points at high loss rates is reasonable as they have a lower impact on variations of the bitrate \( X_{\text{bps}} \); in contrast to the previous sample calculation, the bitrate decrease from, e.g., \( p = 50\% \) to \( p = 50.2\% \) is just 0.005 kB/s.

The concept of a non-linear lookup table is similar to that of the gamma correction in digital image processing [11]. Let \( S_1 \) be a vector that contains sampling points covering the entire domain at even distances, i.e., \( S_1 = [0.1\%, 0.2\%, \ldots, 100\%] \). Furthermore, let \( S_1 \) be \((S_1)^\gamma\) where the power function is calculated element-wise. When evaluating \( f(p) \), we first determine the index \( i_p \) of the entry of \( S_1 \), being closest to \( p \). \( f(p) \) is then approximated by \( L[i_p] \) where \( L \) is a second vector that contains the values of \( f \) at the sampling points. A \( \gamma \)-value of 1 corresponds to a linear lookup table whereas values above 1 imply that low loss rates are finer-grained whereas high loss rates are of lower granularity.

As depicted in Fig. 4, using a non-linear step width with \( \gamma = 3 \) leads to an increased accuracy at \([5\%; 54.4\%]\) and, more importantly, at \([0\%; 0.7\%]\). Increasing the number of table entries is not required. The decrease in accuracy regarding the remaining two ranges is reasonable as packet loss rates above 50% are extremely unlikely, and the step width increase of the range \([0.7\%; 5\%]\) is at most 0.03% (at a loss rate of 5%). Although other values of \( \gamma \) may also lead to an improved coverage of \( f(p) \), after checking various settings, we consider 3 as a good choice as the accuracy-increase at low loss rates and the decrease at higher rates is well balanced.

After modifying the module implementation to support non-linear lookups of \( f(p) \), the measurements of Fig. 3 were repeated. The results, depicted in Fig. 5, illustrate the advantages of our proposal: The step effect can be heavily reduced, and there no longer is a realistic transmission rate upper bound. The only disadvantage of the non-linear table modification is that, in contrast to the original implementation, also forward lookups (from \( p \) to \( f(p) \)) have to be done using a binary search, which increases the computational effort for this operation from \( O(1) \) to \( O(\log(n)) \). However, when considering the small size of the lookup table used (\( n = 1000 \)), the overhead is marginal. Regarding the residual step effect, it is hard to eliminate because, on the one hand, increasing \( n \) would enlarge the module and therefore the kernel’s size, and on the other hand, increasing \( \gamma \) would further reduce the resolution of less frequently used values of \( p \) and therefore jeopardize the smoothness of rate changes at lower bitrates. For the rare use case that smooth rate changes are required even at very low values of \( p \), we propose to define a threshold \( t_p \) (e.g. \( t_p = 0.01\% \)), and to apply the lookup strategy only if \( p > t_p \).
Otherwise, \( f(p) \) is approximated by \( \sqrt{\frac{2}{3}} \cdot \sqrt{p} \) (as the second addend of \( f(p) \) gets negligibly small for \( p < t_p \)) and calculated using Newton iteration.

**IV. INCREASING AGILITY TOWARDS CHANGED TRANSMISSION CONDITIONS**

Due to the fact that with TFRC the maximum allowed send rate primarily depends on the loss event rate, it is important to determine an appropriate estimate of the current rate based on the loss event history. The loss event rate is the inverse of the weighted average length of the \( l \) most recent loss intervals. The lossy fraction of loss intervals is defined to be at most one \( RTT \). As a result, the longer the loss-free fractions of loss intervals are, the smaller the loss event rate gets. The current Linux kernel implementation uses the algorithm specified in section 5.4 of [9]. Consequently, the loss event rate is computed based on the weighted sum of the lengths of the \( l \) most recent loss intervals. It is suggested that the weights \( w_i \) are equal if \( i \leq 1/2 \) and linearly decrease with \( i \) when \( i > 1/2 \). Common values for \( f \) and the weights are \( 8 \) and \( [1, 1, 1, 1, 0.8, 0.6, 0.4, 0.2] \), where the first weight is associated with the most recent loss interval. This weighting approach works well when the distances between loss events do not deviate too much from each other. However, it does not appropriately take long, loss-free periods into account and thus causes the maximum allowed send rate to recover slowly. More specifically, the most recent loss interval is only weighted by the factor \( 1/3t \) regardless of its actual length. During experiments, this turned out to be a limiting factor especially in test scenarios with large \( RTT \)’s. After periods with packet loss, the maximum allowed send rate kept staying below the bitrate required by the user application even under error- and congestion-free channel conditions for a significant amount of time.

Therefore, we replaced the original loss event rate algorithm in the kernel module with the one proposed in section 5.5 of [9], which extends the former by a concept known as history discounting, to see whether unused bandwidth can be saturated faster. This approach makes the loss interval weights dynamic and focuses on the increase of the weight of the most recent loss interval when that interval becomes twice as large as the computed average. The algorithm and its implications are described in depth in [12] and [6]. To give the reader a concrete idea about the performance differences, both mechanisms were additionally implemented in a software simulator. The artificial sender application transmitted packets of size 1.45 kB over a 1 MB/s connection with fixed \( RTT \)’s of 50 ms, 100 ms, and 200 ms to a receiving application. All packets were transmitted at the maximum allowed send rate without idle periods in between. The arbitrarily chosen vector \( v = [16, 77, 23, 54, 110, 21, 224, 43, 53] \) denotes the numbers of consecutive correctly received packets. These sequences of packets were separated by exactly one lost packet to properly initialize the loss event history. After the last (the \( 9^{th} \)) lost packet, 2000 further packets were transmitted without losses. Fig. 6 shows the results of this experiment.

It can be observed that doubling the \( RTT \) leads to an increase of the required transmission time by the factor two, which is reasonable because the \( RTT \) is a factor in the TFRC equation. Furthermore, the history discounting mechanism causes the maximum allowed send rate to increase faster when the most recent loss interval exceeds the average (weighted) loss interval by the factor two. This was also experimentally confirmed in [6] where Floyd et al. derive upper bounds on the rate increase of 0.14 and 0.22 packets per squared \( RTT \) without and with history discounting respectively. In comparison to that, the calculated rate increases while sending the last packets in Fig. 6 for the selected \( RTT \)’s in increasing order are 0.121, 0.121, 0.122 (without history discounting) and 0.320, 0.321, 0.322 (with history discounting). The deviation of the results with history discounting from the derived upper bound is due to the fact that a minimum discount factor of 0.25 was taken as recommended in Section 5.5 of [9] whereas Floyd et al. based their calculations on a threshold value of 0.5. When using that threshold, the respective results are 0.211, 0.213, and 0.214.

Besides confirming the validity of the implementation and the experiment and underlining the advantages of history discounting in terms of rate-responsiveness, these considerations lead to another conclusion: with history discounting, the smoothness property of a connection can be controlled to a certain degree by changing the minimum discount factor \( DF_{\text{min}} \). In such a way, rate increases of up to 0.75 packets per squared \( RTT \) can be reached. This is possible when \( DF_{\text{min}} \) is set to zero: as a consequence, during loss-free periods, the general discount factor \( DF \) gets close to zero because the most recent loss interval significantly exceeds the weighted
average of all recent loss intervals. To derive the maximum rate increase per $RTT^2$, we need to calculate the increase of the average loss interval: when the current average loss interval is $1/p$ packets, then the average loss interval after one $RTT$ is at most $1/p + w_0 \cdot 1/f(p)$ packets where $w_0$ is the weight of the most recent loss interval (also factoring in the discount factor weights); $1/f(p)$ is the maximum allowed send rate in packets per $RTT$. Based on these considerations, the sending rate increases by at most

$$\Delta X = \frac{1}{f(\frac{1}{p + w_0 / f(p)})} - \frac{1}{f(p)} \quad (2)$$

packets per $RTT^2$; for $p \approx 0$ and $w_0 \approx 1$ (when $DF$ gets close to zero, $w_0$ gets close to one), $\Delta X$ is approximately 0.75.

Another possibility to change the agility of the bitrate, i.e., to increase the responsiveness to changed channel or congestion conditions is to modify $l$, the number of weights involved in the loss event rate calculation process. We conducted an experiment in the testbed shown in Fig. 2 and set $l$ to 4, 8, and 16. The weights were calculated using the formula given in [9]. The results, depicted in Fig. 7, indicate that reducing $l$ improves the bitrate agility. This means that the CCID-3 rate control reduces the rate more drastically after experiencing short loss intervals, but, on the other hand, unused bandwidth is faster saturated during error-free periods. The increase of $l$ has the opposite effect: The occurrence of multiple short loss intervals has a lower impact on the bitrate, but it generally takes longer to recover from phases characterized by severe packet loss. Based on the findings presented in this section, it seems that the addition of these tuning features and the provision of a corresponding user-level interface is highly desirable for application developers as this empowers them to decide which kind service to choose based on the specific application requirements. This even makes it possible to adjust the aggressiveness and responsiveness of flows on the fly.

To demonstrate the advantage of history discounting in the context of live video streaming, we transmitted four different test sequences over a loss-affected channel (see Fig. 8). Between second 5 and 15, we increased the packet loss rate so that the maximum allowed send rate of DCCP occasionally exceeded the bitrate of the video stream. This prohibited some packet (re)transmissions because the respective playback deadline was exceeded, thus leading to a decrease in video quality. The difference in video quality of the decoded streams with and without history discounting in place indicates that the faster rate recovery caused by the history discounting mechanism in some cases enables the sender to transmit a higher number of urgent packets, i.e., the negative impact of a slowly growing rate after periods of packet loss on future frames is mitigated.

V. RELATED WORK

There already exist some publications that aim to improve the performance and usability of DCCP, e.g., by extending some of its congestion control mechanisms to maximize TCP-fairness [13], by analyzing the impact of queuing [14], or by proposing cross-layer extensions in heterogeneous environments [15]. In particular, Sarwar et al. investigate the performance of CCID-3 when the connection delay is very long such as over satellite links (when the $RTT$ is in the range of seconds) [16]. They propose an algorithm that aims to optimize the number of feedback messages per $RTT$ based on the observed link delay and present results from both ns-2 and actual satellite transmissions to demonstrate its benefit. In particular, they state that deploying their algorithm leads to an improved slow start performance due to shorter intervals between feedback messages. They further claim that shorter feedback intervals cause the sender rate to recover more quickly. However, after reconstructing their approach, we were unable to experimentally confirm this. In fact, it contradicts the basic idea of the loss event rate calculation as loss events are calculated by the receiving side solely based on the exponentially weighted moving average of the observed $RTT$s and independent of the sender side’s number of received feedback messages per $RTT$.

Somewhat similar to the concepts discussed in Section II, Lai et al. propose a partial reliability extension to DCCP, PR-DCCP, and introduce a new protocol option called retransmitted packet (RP) vector [17]. PR-DCCP allows the sender to specify for each outgoing packet whether to reliably transmit it or not, and, similar to our approach, it requires the endpoints to support ACK vectors. RP vectors are used to help the receiver to identify retransmitted packets, i.e., received packets that contain the payload of packets that have previously been lost, and they further enable the recognition of the reliable fraction of packets.

$^2$These belong to the commonly used video test sequences for qualitative investigations, see http://media.xiph.org/video/derf/
Although the idea of providing application programmers with a convenient way to introduce partial reliability is reasonable, the approach has some drawbacks: RP vectors are used to get the data in the receive buffer in the correct order because, according to Lai et al., the payload may not contain any identifiers such as RTP headers which would allow a receiving application to reconstruct the data stream. As a side effect, in-order delivery must be enforced by PR-DCCP, which has a negative impact with respect to application-level end-to-end delay. Data fragments may be unnecessarily delayed while waiting for retransmitted reliable fragments as opposed to conventional DCCP. Besides sacrificing out-of-order delivery which has repeatedly been mentioned as an attractive protocol property for multimedia applications (e.g. [18]), it seems unlikely that DCCP will ever be used by applications that do not provide identifiers within the payload. Moreover, due to the fact that the reliability issue is handled within the kernel space, the simplicity of the application programming interface comes at the cost of a decreased application flexibility, which is also the reason why selective reliability was deliberately omitted while designing DCCP [1]. For example, it is not possible to query the internal send queue and to place urgent or important packets at its start.

The limitations of TFRC with respect to its compatibility with TCP are discussed by Rhee et al. [19]. They show that the different techniques used to calculate the RTO may lead to an initial sending rate gap, which in turn results in a higher loss event rate for slowly responsive flows such as TFRC. The authors additionally show that a feedback effect caused by that difference further widens the initial throughput difference, and they report cases where TFRC consumes twenty times more bandwidth than TCP, and other cases where TFRC’s bandwidth consumption is only a tenth of that of TCP. To mitigate the amplifying effect of different loss event rates, Rhee et al. propose to apply a modified RTO calculation policy to fix much of the throughput imbalance. They report that when the RTO values of TFRC are always kept larger than those of TCP by a constant factor, the throughput difference can be kept within 20%, which seems to be a reasonable trade-off.

One alternative to the TFRC congestion control mechanism is TCP-friendly window-based congestion control (TFWC), which aims to provide comparably smooth sending rates and claims to improve fairness with competing TCP flows especially in environments with a low level of statistical multiplexing such as Digital Subscriber Lines (DSLs) with drop-tail queuing [20], [21]. TFWC is also equation-based but factors out the term \( \frac{1}{RTT} \) and uses \( 1/f(p) \) (as given in Eq. 1) to calculate the congestion window size \( cwnd \). To use a window-based congestion control mechanism, reliable feedback is required similar to TCP, and therefore, TFWC uses the same ACK vector concept as DCCP’s CCID-2, thus increasing the signaling overhead compared to CCID-3. Unlike TCP, which waits for the packet with the lowest sequence number to be ACKed before sliding the send window, TFWC slides the window either when that packet gets positively ACKed or when its RTO timer expires, i.e., packet timeouts

Fig. 8. The sequences CITY, CREW, HARBOUR, and SOCCER were encoded at an average bitrate of 110 kbps using H.264/MPEG-4 AVC and were streamed over DCCP with a fixed RTT of 50 ms and a packet loss probability of 0.1%. Between second 5 and 15, the packet loss probability was increased to 3% to cause a bandwidth shortage. Selective reliability was turned on as discussed in Section II, and decoding-timestamp earliest-first was used as scheduling policy.
do not lead to retransmissions (as TFWC is also unreliable) but enable sending new packets (assuming that *cwnd* has not decreased in the meantime).

To demonstrate that TFRC starves TCP over links that exhibit the same characteristics as a DSL, Choi et al. present results from ns2 simulations. The starving effect can be mainly attributed to TFRC’s lack of a fine-grained congestion avoidance mechanism (TCP decreases its rate earlier due to ACK-clocking). In contrast to that, TFWC supports ACK-clocking and is therefore fairer in such scenarios. The authors further argue that TFWC is easier to implement than TFRC due to clock granularity issues of the host systems. Although this is to some degree true for user libraries (however, workarounds exist [22]), since 2006, it does no longer apply to kernel code due to the availability of high-resolution timers [23].

Besides, the authors argue that TFWC reclaims bandwidth quicker than TFRC because of the ACK-clocking mechanism, which seems questionable as the send rate of both mechanisms is ultimately connected to \( f(p) \), and although TFWC allows the sender to calculate \( p \) earlier, \( p \) does not decrease faster as the calculation of \( p \) is still connected to the RTT estimate. Moreover, it remains to be seen whether TFWC really improves the fairness in real-world scenarios as only ns2 simulation results have been published so far. It would be especially interesting to see whether the cwnd-jitter-hack mentioned in the paper really preserves TFWC’s smoothness and self-fairness. In connection with the findings that we have presented in this paper, we underline that all extensions are also applicable to TFWC. More precisely, TFWC also benefits from finer grained values of \( f(p) \) to suppress bitrate jumps, it is also desirable to have a means of adjusting rate change characteristics, and it is even easier to implement the feature discussed in Section II because TFWC, unlike CCID-3, requires reliable feedback.

VI. CONCLUSION

In this paper, we have proposed a number of standard-compliant changes to DCCP and implemented them using the implementation that was adopted by the Linux kernel and that is to be seen as reference code for future OS-level implementations to come. In particular, we have developed three extensions from which diverse use cases can profit:

1. The linear lookup table that is used for calculating the TFRC equation (in CCID-3) should be replaced with the non-linear one described in Section III. While the specification [9] mentions the possibility of a lookup table, it does not go into any detail about how this table should be constructed. This change therefore affects only the implementation but not the specification.
2. History discounting should be implemented. Since this mechanism is defined as optional in the specification [9], this change also only affects the implementation.
3. Moreover, applications should be allowed to change and adjust the number of weights involved in the loss event calculation process.
4. The API should allow an application to keep track of outgoing sequence numbers and incoming acknowledgments. Since there is no API specification, this change also only affects the implementation. The reason for us to propose these changes is that they were needed in order to attain a good quality with our live video streaming system, which heavily relies on selective ARQ. It was therefore a necessity to track sequence numbers to even implement our system, the full benefits of which are described in [5].

We believe that the lack of a specified API is a significant shortcoming, as future DCCP-based applications should ideally be able to use a uniform interface irrespective of the OS. Socket API extensions for the Stream Control Transmission Protocol (SCTP) have recently been specified [24], and it seems illogical and inconsistent to not do the same for DCCP. Such an API specification should build upon the lessons that we have learned, and include the extensions that we have defined in this paper.

This paper has focused on effects appearing with only one single DCCP flow. When multiple TFRCs share a bottleneck in parallel, they show some deficiencies—TFRC’s reactivity and its smoothness are degraded. We have addressed this issue with MulTFRC, which utilizes a new equation that can represent not only one but \( N \) flows, where \( N \) is a positive rational number and can even be smaller than one [25]. We believe that including MulTFRC as a new CCID in DCCP would therefore constitute another improvement of the protocol; the requirement of avoiding floating point calculations in the Linux kernel however makes implementing the MulTFRC equation in DCCP a non-trivial task. This is planned as future work.

REFERENCES