SCHOOL OF ENGINEERING - STI
SIGNAL PROCESSING INSTITUTE
Christophe De Vleeschouwer and Pascal Frossard

CH-1015 LAUSANNE

Telephone: +4121 6932601
Telefax: +4121 6937600
e-mail: christophe.devleeschouwer@epfl.ch

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

EXPLICIT WINDOW-BASED TRANSPORT CONTROL PROTOCOLS IN LOSSY ENVIRONMENTS

Christophe De Vleeschouwer and Pascal Frossard

Swiss Federal Institute of Technology Lausanne (EPFL)

Signal Processing Institute Technical Report

TR-ITS-2004.011

May 7th, 2004

Part of this work has been submitted to IEEE ICNP 2004

Explicit window-based transport control protocols in lossy environments

Christophe De Vleeschouwer and Pascal Frossard
LTS4 - Signal Processing Institute - EPFL
1015 Lausanne, Switzerland

***Abstract*— This paper addresses efficient packet loss recovery by retransmission in window-based congestion control protocols. It builds on explicit congestion control mechanisms to decouple the packet loss detection from the congestion feedback signals. Implicit algorithms alternatively infer congestion from losses (which yields to window size reduction), and therefore do not allow to evaluate the performance of window-based transmission algorithms in lossy environments. We first propose a simple modification of TCP that offers the possibility for explicit congestion control. Different retransmission strategies applicable to window-based congestion control protocols are then discussed in the framework of explicit congestion control. We introduce a new early retransmission timer that significantly improves the error resiliency when combined with explicit congestion control. Extensive simulations then compare the error recovery mechanisms generally used in recent TCP implementations, and the new loss monitoring and recovery strategies, combined with explicit congestion control protocols. Performances are analyzed in a simple network topology where a bottleneck link is shared by loss-free, and respectively lossy connections. Retransmissions triggered by the proposed accurate loss monitoring mechanism are shown to end up in a fair share of the bottleneck bandwidth between all connections, even for high loss ratios and bursty loss processes. The link utilization is in the same time close to optimal. Explicit congestion control, combined with efficient error control strategies, can therefore provide a valid solution to reliable and controlled connections over lossy network infrastructures.**

I. Introduction

Congestion control is imperative for the well-being of the network. It prevents any one connection from swamping the links and switches between communicating hosts with an excessive amount of traffic. In essence, congestion control prevents overwhelming the network with too many packets. A natural way to achieve this goal is to limit the number of packets that are in transit between the sender and the receiver. Window-based congestion control mechanisms follow this idea by limiting, for each connection, the number of transmitted but yet to be acknowledged packets. Window-based protocol achieve network stability by forcing the connection to obey a 'packet conservation' principle, which means that a new packet is not push into the network until an old packet leaves [18].

TCP is the window-based congestion control protocol used for the Internet. TCP is an implicit end-to-end congestion control [1] in the sense that the network components, i.e. the routers, provide no explicit support to the transport layer for congestion-control purposes. Congestion in the network must be inferred by the end-systems based only on the observed network behavior, e.g. packet losses and delay. However, loss is known to be a poor signal of congestion, because (i) congestion is not the only source of losses, (ii) a loss is a binary signal that do not provide precise congestion feedback, (iii) the decision that a packet was lost cannot be made quickly. For these reasons, TCP becomes inefficient and prone to instability when the delay-bandwidth product increases, or when packets are subject to non-congestion related losses.

Explicit congestion control mechanisms, where routers provide explicit feedback to the sender regarding the congestion state, have been proved to solve the inefficiency and instability problem related to (ii) and (iii) [11],[17]. As an example, Katabi and al. [11] have designed an eXplicit Control Protocol, XCP, where the network uses precise and explicit congestion signaling to tell the senders how to react to the state of congestion. The resulting protocol is both more responsive and less oscillatory than conventional TCP, which becomes especially beneficial when the delay-bandwidth product increases.

In this paper, we focus on the problem related to (i), i.e. to the fact that all losses are not necessarily caused by congestion. To illustrate the problem and motivate our study, we now survey the efforts made to improve TCP performance over lossy links. Because TCP interprets any kind of losses as a congestion notification, without any precaution, TCP results in flow starvation over lossy links. The problem is well-known, especially in wireless environments. In the past decade, three approaches have been considered to circumvent it [4]. The first one consists in increasing link-layer reliability to hide link-related losses from TCP sender [5], [2], [16]. The second one splits the end-to-end connection, and terminate the TCP connection at the base station to hide the lossy link from the sender [3], [6]. The third approach attempts to give the TCP sender the capability to handle non-congestion related losses appropriately. Some schemes refine the TCP acknowledgments to allow the TCP sender to recover from multiple losses without resorting to a coarse timeout [15], [8], [14], while others distinguish between congestion-related losses and other forms of losses, either based on explicit loss notification [4], or based on end-to-end bandwidth estimation [7].

Our work is related to the third approach in the sense that we do not attempt to hide losses to the sender, but rather want to give the sender the capability to handle them. However, our work is quite different from the approaches described in [4], essentially because our goal is not to improve TCP performance, but rather to explore the limitations of the window-based congestion control paradigm in lossy environments.

This work has been funded by the Swiss NSF and by the Belgian NSF.

The two main sources of inefficiency for window-based congestion control in presence of losses are (a) the erroneous interpretation of a loss as a signal of congestion, which wrongly ends up in congestion window deflation, and (b) the fact that the congestion window is linked to the highest fully acknowledged sequence of data segments, which prevents any new transmission before a previously lost segment has been successfully retransmitted. To circumvent the first issue, we promote the use of explicit congestion control mechanisms. For these protocols, as long as explicit information about congestion is received by the sender, there is no reason for the sender to infer the congestion state from losses or delay measurements. Hence, losses are not specifically interpreted as a congestion signal anymore, and do not cause congestion window deflation. A major contribution of our work consists in the evaluation of the potential of explicit congestion control to address the problem related to non-congestion related losses in a window-based transport paradigm. To address the second issue, we envision the combination of retransmission mechanisms with temporary increase of the sender window in response to duplicate acknowledgments. In principle, the approach is similar to the fast retransmit and fast recovery mechanisms used in TCP Reno [1] or NewReno [9]. We show however, and this is another important contribution of our work, that an implementation dedicated to the explicit congestion control framework ends up in substantial performance improvements in comparison with a direct transposition of TCP retransmission mechanisms. In addition, we quantify the benefit brought by an accurate monitoring of losses, such as offered by selective acknowledgment [15], or the identification of the packet that triggered an acknowledgment [12].

The paper is organized as follows. Section II introduces the framework of our study. It defines the state variables that characterize a window-based connection, and presents two explicit congestion control algorithms. The first one is the eXplicit Control Protocol (XCP), introduced by Katabi and al. [11]. The second one is an original proposal to make TCP explicit. To make these protocols robust against losses, i.e. to preserve their transport efficiency, Section III and IV identify two kinds of loss-resilient mechanisms, depending on the feedback information provided by receiver acknowledgments. In Section III, similar to basic TCP, the receiver only informs the sender about the latest data segment received in-sequence. It does not identify the additional segments that might have been received out-of-order, and cached at the receiver. In that case, our contribution consists in exploiting the specificities of an explicit control framework to improve the loss-resilience mechanisms proposed in the TCP context. On the contrary, in Section IV, the sender learns from the receiver feedback which exact packets have reached the receiver. This additional information is exploited to derive an original, and even more efficient recovery mechanism. Section V and VI validate and compare the two approaches based on NS simulations. They demonstrate the potential of explicit window-based congestion control algorithms in lossy environments. In addition, the proposed eXplicit TCP protocol is shown to be able to co-exist fairly with TCP, in a single queue of an XTCP-enabled router. This is attractive w.r.t. its deployment. Section VII concludes.

II. Explicit window-based congestion control framework

As mentionned in the introduction, the goal of our paper is to explore the ability of window-based congestion control protocols to support transmission losses. This section introduces the envisioned framework. It first defines the main state variables that characterize a window-based connection. Then, it introduces two window-based congestion control algorithms. These protocols are explicit, which means that the information about the state of congestion is explicitly transmitted by the routers to the sender via receiver acknowledgments. Explicit protocols are of particular interest for the rest of our study because they offer the possibility to decouple the congestion control task from the loss detection (and recovery) process. On the contrary, implicit algorithms, which infer congestion from losses, are dominated by the reduction of the congestion window size associated to losses, and consequently, have no chance to be efficient in lossy environments.

A. Window-based protocols: Definitions

This section introduces a number of state variables that characterize a window-based connection.

We limit our study to window-based control protocol based on positive acknowledgment (ACK), i.e. for which the receiver sends feedback information in response to correctly received packets. The feedback information is based on the data sequence number and, possibly, on the packet sequence number, read in the received packet header. These sequence numbers are defined as follows.

The **data sequence number** associated to a packet, and denoted $dseqn$, identifies the data segment conveyed by the packet. Two transmissions of the same data are thus characterized by the same data sequence number.

The **packet sequence number**, denoted $pseqn$, identifies each packet sent by the sender. The packet sequence number monotically increases with the time at which the packet is sent. In practice, it corresponds to a counter incremented by one each time a new packet is sent. Two packets that (re)transmit the same data at different time instant have thus the same data sequence number, but distinct sender sequence number. The benefit that can be drawn from the packet sequence number is described in Section IV.

Based on the data sequence number definition, a **cumulative acknowledgment** equal to N indicates that all the data segments with a data sequence number up to and including N have been correctly received at the receiver. At any time, the ***lack*** state variable records the largest cumulative acknowledgement ever received by the sender.

By definition, window based congestion control limits the number of transmitted-but-yet-to-be-acknowledged packets. This number of permissible packets is referred to as the **congestion window size**, and is generally denoted ***cwnd***. This variable is the one that constraints the rate of the connection based on the network state of congestion. It is adjusted based on the information explicitly received from the routers, or implicitly inferred from the observation of the network behavior (losses and delay).

In addition to the congestion window, we define the **send-out window**, denoted ***swnd***, so that all data with sequence number lying between $lack$ and $lack + swnd$ are eligible for being sent. In practice $swnd \geq cwnd$, and the difference between $swnd$ and $cwnd$ corresponds to data packets that have already left the network, and are stored at the receiver. Note that $swnd$ is upper bounded by the receiver advertised window, denoted $rwnd$, which reflects the buffer capacity at the receiver.

The data sequence number of the next segment to be considered for transmission by the sender is stored in the ***nextseq*** state variable. We describe in Sections III and IV how $lack$, $swnd$ and $nextseq$ are updated upon reception of receiver acknowledgments, or upon timer expiration. This determines the sender behavior as, after every update, the data segments to send are identified as the ones whose data sequence numbers lie between $nextseq$ and $last + swnd$.

B. The eXplicit Control Protocol (XCP)

This section briefly reviews the eXplicit Control Protocol (XCP) proposed in [11]. XCP is window-based, and controls the size of the congestion window based on explicit and accurate feedback from routers. In short, XCP is based on a few bytes of control information conveyed in the packet headers. To control the link utilization, routers inform the senders about the degree of congestion at the bottleneck. In a router, the feedback about congestion to be conveyed by a packet is computed based on the mismatch between the aggregate traffic rate and the link capacity, and is adjusted according to the feedback delay expected for the packet. Fairness is achieved based on reallocation of bandwidth between individual flows. Extensive simulations demonstrate that XCP maintains good utilization and fairness, while maintaining small standing queue size. In particular, [11] shows that XCP outperforms TCP when the per-flow delay-bandwidth product becomes large.

C. Our proposed eXplicit TCP (XTCP)

In this section, we propose an explicit congestion control protocol that adopts the exact same additive increase - multiplicative decrease behavior as TCP. The proposed protocol is named eXplicit TCP (XTCP). By defining an explicit protocol that is as close as possible to TCP, we plan to measure the gain provided by an explicit feedback, independently of the congestion control mechanism itself. Another advantage of a TCP-like explicit protocol is related to deployment issues. This will be further discussed in Section VI-A.

In short, the only difference between the proposed eXplicit TCP and conventional TCP is the way the decision to decrease the congestion window is taken. The TCP sender implicitly infers congestion from a lost data [13]. On the contrary, XTCP requires an explicit feedback from the routers to decrease its congestion window. We have chosen a simple binary feedback mechanism that, similar to TCP, indicates whether there is congestion or not in the router. Hence, in absence of accurate feedback, the sender has to probe the network to the point of congestion before backing off, just as TCP.

In practice, the binary feedback is provided through a *congestion flag* contained in the XTCP packet header. The flag is initialized to zero by the sender, and is set to one when the packet encounters a congested router. When received by the receiver, the flag is copied in the ACK header, and returned to the sender. Upon ACK reception, the sender decides whether the congestion window should be decreased or increased based on the congestion flag. Similar to TCP, the congestion window is incremented each time an ACK with congestion flag set to zero is received, and is divided by two when the sender infers a congestion event based on the returned congestion flags. By definition, a congestion event occurs when an ACK with congestion flag set to one is received, and when the latest congestion event is older than one RTT. This is to avoid multiple backoffs during one RTT. In practice, an exponential weighted average of the RTT samples is used to estimate RTT. We can now explain how routers update the congestion flag.

Formally, a *congestion counter* is associated to every queue in the network. Each time a queue drops a packet due to congestion, the congestion counter associated to the queue is incremented by one. When an XTCP packet leaves the queue to be sent out to the output link, if the counter is positive, the congestion flag of the XTCP packet is set to one, and the counter is decremented by one[1]. It is worth noting that the packet whose congestion flag is set to one does not necessarily belongs to the same flow as the packet whose drop has caused incrementation of the congestion counter. So, there is no need to maintain per flow state in the router. Note also that XTCP does not make any assumption about the queue management discipline used in routers. In concrete terms, XTCP can be used both with Droptail or RED policies.

Before exploring the performance of XCP and XTCP in lossy environments, Section III and IV introduce the mechanisms that are expected to make explicit control protocols robust in presence of losses.

III. Loss-resilience based on cumulative acknowledgment

Section III and IV present mechanisms that are expected to preserve window-based connection efficiency in presence of losses. In both sections, we consider an explicit congestion control framework that offers the possibility to decouple the congestion control task from the loss recovery process.

In this section, we assume that the receiver sends out the largest possible cumulative acknowledgment upon reception of a packet. Based on this limited feedback, we envision four mechanisms to trigger the retransmission of a lost data packet. The ideas behind the three mechanisms described in Sections III-A, III-B, and III-D are familiar to TCP designers [1], [9]. However, the fourth mechanism, described in Section III-C, and its impact on the implementation of the three other mechanisms, are specific to the explicit congestion control framework envisioned in this work. The NS simulations presented in Section V-B and VI-B reveal that this

[1]To make sure that we do not run in a situation where the counter is positive, and the queue is empty, we only increment the counter if its current value is smaller than the number of packets present in the queue.

timer significantly improves the performance of the transport protocol in presence of losses. That is the reason why we find important to explain in detail how these mechanisms work and interact. To highlight our contribution, Section III-E summarizes the differences between the TCP implementations of fast retransmission mechanisms, and the approach described in this paper.

A. Retransmission based on duplicate acknowledgment

This section describes how the principle underlying the fast retransmit/fast recovery mechanism of TCP Reno are adapted to an explicit congestion control framework. We observe that, on the contrary to TCP Reno, the explicit congestion control framework (i) does not need to halve the congestion window in response to multiple duplicate ACKs; (ii) only partially deflate the send-out window upon reception of a new ACK.

Readers that are familiar with TCP probably understand the meaning and implications of these two points. Hence, they can directly move to Section III-B. Other readers might be interested in the following description.

In the absence of loss or packet reordering, any acknowledgment indicates the correct reception of novel data at the receiver, and is referred to as a **new ACK**. In presence of losses, this is not the case. The reception, at the receiver, of data that have a higher data sequence number than not-yet-received data triggers a duplicate acknowledgment. This **duplicate ACK** actually re-acknowledges data for which the sender has already received an acknowledgment. Upon reception of a duplicate ACK, the sender infers that the data immediately following the largest acknowledged data, i.e. the $(lack+1)^{th}$ data segment, has either been delayed or lost by the network. In practice, similar to the principle adopted by TCP Reno, our sender waits for several duplicate ACKs before concluding the packet has been lost, and retransmitting it. We denote $dupackthreshold$ the number of duplicate ACKs beyond which the sender infers the loss of the $(lack+1)^{th}$ data segment. However, conversely to TCP Reno, our sender can rely on the explicit feedback provided by ACKs about congestion, and consequently, does not need to halve the congestion window in response to a detected loss.

Regarding the implementation, a counter, denoted $dupacks$ is initialized to zero and incremented by one every time a duplicate acknowledgment reaches the sender. This counter is reset to zero each time new data are acknowledged, or when the connection is reset (see Section III-D). In addition, the arrival of a duplicate acknowledgment at the sender indicates that a packet has reached the receiver, and has thus left the network. In accordance with the congestion window definition, which limits the number of sent-but-not-yet-received packets, a novel data packet can be sent out by the sender in response to the arrival of a duplicate acknowledgment. This is done by incrementing the send-out window $swnd$ by one. Specifically, $swnd$ is monitored as the sum of $cwnd$ and $dupwnd$, where $dupwnd$ denotes a counter initialized to zero and incremented by one every time a duplicate ACK reaches the sender. The $dupwnd$ counter is reset to zero when the connection is reset. Upon reception of new data acknowledgment, the head of the send-out window is moved to the novel largest acknowledged data segment, possibly overstepping a number of data segments whose earlier reception had triggered duplicate acks, and caused $dupwnd$ incrementation. To maintain the number of sent-out-but-not-yet-received packets in the network equal to the congestion window, $dupwnd$ has thus to be decremented by the amount of these segments that are now acknowledged, but which triggered duplicate acks in the past. Let $olack$ and $nlack$ respectively denote the last acknowledged data segment before and after the reception of the new ACK. $dupwnd$ should then be decremented by $nlack - (olack + 1)$ upon reception of the new ACK. Note that the way $dupwnd$ is decremented differs from the classic implementation proposed by TCP Reno [1], which simply resets $dupwnd$ to zero upon reception of a new ACK. This is further discussed in Section III-E.

In the following, we present 3 other retransmission mechanisms, and consider their impact on the duplicate ACK mechanism. In particular, Section III-B explains that the way the $dupacks$ and $dupwnd$ counters are managed has to be modified in presence of retransmission mechanisms based on partial acknowledgments. Moreover, in Section III-D, we state that $dupacks$ and $dupwnd$ should not be incremented in a period immediately following the expiration of the connection recovery timer.

B. Retransmission based on partial acknowledgment

This retransmission mechanism has been proposed by the NewReno version of TCP [9], and is expected to help when multiple packets are lost from a single window of data. Readers familiar with TCP NewReno can immediately move to Section III-C.

Among new data acknowledgments, NewReno distinguishes between **complete** and **partial acknowledgments**. Let $olack$ and $nlack$ respectively denote the largest data sequence number acknowledged before and after the reception of a new ACK. A new ACK is defined to be a complete ACK if it acknowledges all the intermediate data segments that have been sent between the initial transmission of the $(olack+1)^{th}$ segment and its last (re)transmission. On the contrary, a new ACK is a partial acknowledgment if it only indicates the correct reception of a subset of these segments. From a practical point of view, the reception of a partial ACK means that the $(nlack + 1)^{th}$ data segment has not been received at the receiver at the time the $(olack + 1)^{th}$ was received. In absence of packet reordering in the network, we interpret this information as the loss of the $(nlack + 1)^{th}$ data segment.

To decide whether a new ACK is a partial or a complete acknowledgment, the sender uses a state variable, denoted ***recover***, to remember the largest sequence number sent out before the last retransmission of the next data to acknowledge, i.e. of the $(olack+1)^{th}$ data segment. Specifically, every time a data segment is retransmitted, either because a timer expires or a retransmission mechanism becomes active, $recover$ records the largest data sequence number ever sent out by the sender. When receiving a new ACK, the sender compares the novel largest acknowledged data sequence number, i.e. $nlack$, with

the $recover$ value. If $nlack$ is strictly smaller than the $recover$ value, the sender infers that the new ACK is a partial acknowledgment, and retransmits the $(nlack+1)^{th}$ data segment.

It is worth noting that in presence of the partial acknowledgment retransmission mechanism, the $dupacks$ counter defined in Section III-A should only be reset upon reception of a complete acknowledgment. So, the acknowledgment of new data by a partial acknowledgment should not cause a reset of the $dupacks$ counter. This is to avoid multiple retransmissions of the same packet during the same round trip time, i.e. one due to the partial ACK and the other due to $dupacks$ reaching $dupackthreshold$.

C. Early retransmission timer

Retransmission mechanisms based on duplicate acknowledgments rely on the reception of several duplicate acks before inferring that a packet has been lost. Hence, they are not efficient for small congestion window sizes. Moreover, and more important, the $dupacks$ state variable is only reset to zero after the acknowledgment of new data[2]. As a consequence, retransmissions of subsequent lost segments rely on the correct reception of the initial retransmitted segment. In other words, retransmission mechanisms based on duplicate ACKs do not support multiple loss of the same segment. The same conclusion holds for the retransmission mechanism described in Section III-B, and based on the arrival of partial acknowledgments at the sender.

As retransmission mechanisms based on duplicate or partial ACKs are vulnerable, we propose a retransmission mechanism of last resort, based on a timer expiration. This timer, denoted **early retransmission timer**, is reset every time new data are acknowledged -because the candidate segment for retransmission changes as a consequence of the new ack-, and every time a data segment is retransmitted. Note however that the early retransmission timer is cancelled upon epiration of the recovery timer (see Section III-D). Note also that the early retransmission timer is not reset upon reception of a duplicate ACK, except if the duplicate acknowledgment causes a retransmission. Upon timer expiration, the $(lack+1)^{th}$ data segment is retransmitted.

Regarding the implementation, the timer expires after a timeframe called timeout. In practice, a short timeout results in a fast retransmission, and in rapid loss recovery. However, one should take care not to trigger retransmissions for packets that are not lost, i.e. that are still in transit. In particular, an important issue to consider when dealing with retransmission based on a timer, is the stability issue. The problem comes from the fact that a bad estimation of the timeout might cause the sender to inject a new packet into the network before an old one has exited, which is against the 'packet conservation principle' stated as a guarantee of stability for window-based transport protocols [18]. To avoid this problem, the early retransmission timeout has been set to a longer timeframe than the timeout of the recovery timer defined in Section III-D. This choice guarantees that, in case of trouble, e.g. due to a bad round trip time estimation, the connection ends up in a recovery phase, and does not swamp the network with inadequate retransmissions. During our simulations, the early retransmission timeout has been defined twice as large as the recovery timeout. As told in Section III-D, both timeouts are thus computed as a conservative estimation of the round trip time (RTT). To validate this choice with regards to stability, we have run a simulation in which some connections have deliberatly under-estimate the round trip time when computing the recovery and, consequently, the retransmission timeouts. We have observed that these connections did not strangle other connections, and that they were themselves strongly penalized by regular expirations of the recovery timer. We have also observed in the simulations presented in Sections V-B and VI-B that the expiration of the retransmission timer is rare and always appropriate. Hence, we conclude that defining the retransmission timeout as a larger and scaled version of the recovery timeout prevents unstable behavior of the system.

[2]If $dupacks$ was reset to zero immediately after the retransmission of a packet, subsequent duplicate acknowledgments that correspond to the same window of emission would trigger an additional and probably useless retransmission.

D. Mechanism of last resort: the recovery timer

In complement to retransmission mechanisms, a window-based transport protocol resorts to a timer as a recovery mechanism of last resort. This timer, denoted recovery timer, is different from the one introduced in Section III-C, and is present in all TCP implementations. We explain in this section that, in an explicit congestion control framework, there is an advantage to manage, i.e. to reset, the recovery timer in a different way than for TCP.

Before going into the details of the recovery timer management, we define its purpose. The **recovery timer** expiration indicates that the connection stayed idle for a while, and has to be reset. A **connection reset** consists in setting $cwnd$ to one, and $nextseq$ to $lack+1$ (see definitions in Section II-A). The timeout of the recovery timer is generally set to a conservative estimation of the round trip time (RTT). In our simulations, we have used the same timeout as the one proposed for TCP [13]. This timeout is defined as the sum of an exponential weighted average of the RTT samples, and of an estimate of how much the RTT samples deviate from this average.

We now analyze how to handle the recovery timer. From our simulations, we have concluded that an appropriate management of the recovery timer is important to preserve the connection efficiency in the presence of losses. In particular, the strategies adopted to reset the recovery timer have to be adapted to the explicit congestion control framework, and depend on the presence or the absence of an early-retransmission timer.

When an early retransmission timer is used, the recovery timer can be reset both in response to a new ACK and to a duplicate ACK. This is intuitively a good behavior in the sense that the reception of acknowledgments by the sender means that the connection is still on. Remember here that we are considering explicit congestion control mechanisms, for which information about the state of congestion is transmitted by the routers to sender. So, as long as the sender receives

acknowledgments, it receives information about the state of congestion. It should thus concentrate on recovering from losses rather than deflating the congestion window in response to duplicate ACKs.

On the contrary, in the absence of early retransmission timer, we can not adopt this strategy because, in that case, there is no way to retransmit a data segment that has been lost twice. So, if the recovery timer was reset at every duplicate ACK, we could run in a situation where $dupwnd$ would go to infinity (or at least a value that pushes $swnd$ to $rwnd$), before the recovery timer expires and permits the retransmission of the lost packet. A better behavior is to trigger the recovery phase earlier, i.e. as soon as we know that the segment has been lost a second time. For this reason, the recovery timer is not reset upon reception of a duplicate acknowledgment that does not cause a retransmission. As a conclusion, in the absence of early retransmission timer, the recovery timer is reset every time a new ACK is received, but is only reset upon reception of a duplicate ACK if this ACK triggers a retransmission. This strategy is the one adopted by TCP.

We now describe some subtle issues associated to the recovery timer expiration.

Upon expiration of the recovery timer, the connection is fully reset. This implies cancellation of the potential early retransmission timer.

After a timeout, $nextseq$ is decremented to $lack+1$, which means that data segments that have already been sent out, and possibly received, can become eligible for retransmission in the future. This particular condition lasts as long as $lack$ remains strictly smaller than the largest data sequence number ever sent out by the sender at the time the recovery timer expires. It is referred to as the recovery phase, and requires particular attention from the sender.

During the recovery phase, the sender wants to avoid retransmission of packets that have already been received. For this reason, during recovery phase, we have decided to constrain the sender to send out a single data segment, i.e. the one labeled $lack+1$, in response to a new ACK or to an early retransmission timer expiration. Note that the sender should take care to update $nextseq$ to $lack+2$ after a retransmission triggered during the recovery phase.

An even more subtle issue related to the recovery phase is referred to as "bugfix" in TCP description [9]. It is related to the fact that when the sender enters a recovery phase, there might be packets or acknowledgments flowing between the connection end-hosts. These packets trigger duplicate ACKs that do not carry any information about the current state of the connection. For this reason, during the recovery phase, the sender should simply ignore duplicate acknowledgments.

E. Comparison between our retransmission strategy and TCP

As a summary, the main differences between the retransmission/recovery mechanisms that we propose, and the ones proposed for TCP result from (i) the fact that losses are not immediately interpreted as a signal of congestion in the explicit control framework, (ii) the presence of an early retransmission timer and its impact on the recovery timer management.

As it does not interpret losses as a signal of congestion, an explicit control algorithm does not need to sharply reduce the window in response to duplicate ACKs. This permit to keep the connection active and efficient as long as sufficient ACKs are received, i.e. as long as packets can flow over the network.

As explained in Section III-D the presence of an early retransmission timer allows for a reset of the recovery timer each time a new or a duplicate acknowledgment is received. This, in turns, keeps the connection alive, and preserves transmission efficiency. This is especially beneficial when coupled with point (i), i.e. with the fact that an explicit control framework does not have to deflate the congestion window in response to duplicate ACKs.

Another difference is the way $dupwnd$ is updated upon reception of acknowledgments. In our proposal, $dupwnd$ is incremented in response to the very first duplicate acknowledgment. So, we do not wait for $dupackthreshold$ ACKs before starting inflating the send-out window. Moreover, we always consider partial deflation of $dupwnd$ upon reception of a new acknowledgment. In comparison, for TCP Reno, $dupwnd$ is simply reset to zero when a new ack is received. Our simulations reveal that the benefit drawn from the partial deflation of $dupwnd$, which had also been proposed by TCP NewReno, is significant. But it is worth noting that the importance of this benefit directly results from the explicit congestion control framework, for which the congestion window is not sharply deflated in response to duplicate ACKs. In the TCP case, because duplicate ACKs are interpreted as a signal of congestion, the congestion window gets smaller in response to duplicate ACKs, which in turns decreases the send-out window, and strongly reduces the benefit obtained from a partial deflation of $dupwnd$.

Two other specificities of our implementation are (i) the update of the congestion window in response to the explicit feedback contained both in new and duplicate ACKs, and (ii) the constraint to send out a single packet in response to a new ACK during the recovery phase, despite a potential increase of the congestion window.

IV. Loss-monitoring based on accurate receiver feedback

With the limited information available from cumulative acknowledgments, a sender can only learn about a single lost packet per round trip time. In this section, we envision more detailed feedback to be sent by the receiver to the sender. The main purpose of our study is to evaluate the benefit that can be drawn from an accurate receiver feedback when the window-based connection is subject to losses. As in Section III, we still consider an explicit congestion control framework. Ultimately, based on the mechanisms presented in this section, and on the simulations presented in Section V and VI, our goal is to discuss the conditions under which a window-based transmission protocol based on explicit congestion control is able to support connections over lossy links.

The section is split in two sub-sections. Section IV-A presents the protocol that we have implemented to improve the feedback provided by receiver acknowledgments, while

Section IV-B describes the loss retransmission mechanism that we have specifically designed to exploit the novel information available from the receiver.

A. Packet sequence number feedback

The main limitation of cumulative acknowledgments comes from the fact that upon reception of a duplicate ACK, the sender has no information to identify the packet that triggered the ACK, and consequently, has no information to identify the data segment that has reached the receiver. A simple way to circumvent this limitation is to uniquely identify every packet sent out on the network. In our simulator, this has been done by adding a field to the packet header that contains its packet sequence number. Remember from Section II-A that the packet sequence number, denoted $pseqn$, is defined based on a counter incremented by one each time a new packet is sent. Every time the source sends a packet, it writes the state of the counter in the packet header, and increments the counter by one. This concept of packet sequence number had been introduced by Keshav and Morgan [12] to design efficient retransmission mechanisms in the context of rate-based congestion control. On the contrary, we are interested in window-based congestion control, for which the emission of packets is directly related to the position of the send-out window, which itself is linked to the largest cumulative acknowledgment received by the sender. This makes the problem much more complex than in a rate-based control context, where the transmission of new packets and the loss recovery mechanisms are totally decoupled [12].

Another way for the sender to learn about the data segments that have reached the receiver is the selective acknowledgment option proposed for TCP [15]. We expect that conclusions drawn from our implementation could be generalized to a SACK implementation.

Once informed about the data received, the sender can adopt intelligent strategies to retransmit the missing data segments. This is described in the next section.

B. Retransmission based on accurate loss-monitoring

Based on the feedback provided by the receiver about the packets that have been received, both in order and out of order, the sender is able to manage a **loss-monitoring window**. This window stores information about the segments that are still waiting for a cumulative ACK, and its size is thus bounded by the largest number of out-of-sequence packets that can be buffered at the receiver, i.e. by $rwnd$. In this section, we first define how the loss-monitoring information is maintained. We then explain how the information is exploited to trigger data retransmissions.

Let $N < rwnd$ denote the size in packets of the loss-monitoring window. Given $lack$, the largest acknowledged data sequence number, the loss-monitoring window stores a state forall data sequence number j so that $lack < j < lack+1+N$. In practice, we use a circling buffer to store the state of the relevant data segments. Let $W[.]$ be an array of size N. At any time, $W[j \bmod N]$ stores the loss-monitoring window state corresponding to the j^{th} data segment. Given $lack < j < lack+1+N$, $W[j \bmod N]$ is defined as follows:

- $W[j \bmod N] = FREE$, with $FREE$ being a constant flag value, when the j^{th} data segment has not yet been sent over the network;
- $W[j \bmod N] = RECV$, with $RECV$ being a constant flag value, when the j^{th} data segment has been received out of order by the receiver;
- $W[j \bmod N] = X$, with $X > 0$ being the packet sequence number of the latest packet sent over the network and conveying the j^{th} data segment, in other cases.

In practice, the loss-monitoring window state variable is maintained as follows:

(a) First, each array position is initialized to the constant $FREE$ value, indicating that the array does not record any state variable yet. Each block of the array is thus available to store the state of future data segments.

(b) When a packet is sent out, the loss-monitoring window is updated as follows. Let d denote the data sequence number of the data segment conveyed by the packet, and p be the packet sequence number. Then, $W[d \bmod N] \leftarrow p$, indicating that all packets with a packet sequence number larger than p have been sent out after the last emission of the d^{th} data segment. Recording this information is important w.r.t. the retransmission mechanism proposed hereunder.

(c) Upon reception of a new ACK, the $lack$ state variable is updated. Let $olack$ and $nlack$ respectively denote the old and new $lack$ values. Hence, $W[j], olack < j < nlack$ is reset to $FREE$, indicating that the corresponding positions of the array are now available to store the state of future data segments. Note that before sending out a new data segment on the network, let say the n^{th} segment, the sender has to check that $W[n \bmod N] = FREE$, indicating that the new segment will not exceed the storage capacity of the loss-monitoring window.

(d) Upon reception of a duplicate ACK, let p denote the packet sequence number of the packet that triggered the ACK. Remember p can be read from the header of the received ACK, as described in Section IV-A. Let then $d = D[p]$ denote the data sequence number of the data segment conveyed by the p^{th} packet. Then, $W[d \bmod N] \leftarrow RECV$ to denote that the d^{th} data segment has been received by the receiver. Note that $D[p]$ is known at the sender, as the sender obviously knows which data segment has been sent in a given packet.

Given $W[.]$, the design of our proposed retransmission mechanism is then guided by the following rules, stated in terms of the $dupackthreshold$ parameter:

- a data segment can only be retransmitted once the sender has received $dupackthreshold$ acknowledgments triggered by packets that have been sent out later than the data segment;
- we accept a maximum of one retransmission per $dupackthreshold$ acknowledgment.

To implement a retransmission mechanism that follow these rules, we define a state variable, denoted $rseqn$, which denotes

the sequence number of the data segment that is expected to be the best candidate for a retransmission. Among all data segments monitored by the window W, $rseqn$ is defined as the one whose latest (re)transmission has been performed the furthest in the past, and for which the sender has no indication about reception by the receiver. Formally, given W[.], $rseqn$ is defined by:

$$rseqn = \underset{lack+1<j<lack+1+N,\ W[j]\neq RECV/FREE}{\arg\min} W[j] \quad (1)$$

Let $dupcount$ denote a counter which is incremented by one every time an ACK triggered by a packet whose packet sequence number is larger than $W[rseqn]$ reaches the sender, without indicating the correct reception of the $rseqn^{th}$ data segment. The fact that the ACK has been triggered by a packet with a larger sequence number than $W[rseqn]$ means that the ACK or packet corresponding to the last retransmission of the $rseqn^{th}$ data segment has either been delayed or lost. Once $dupcount$ reaches $dupackthreshold$, the $rseqn^{th}$ data segment is retransmitted. After a retransmission, or after the sender received an ACK indicating the correct reception of the $rseqn^{th}$ packet, $dupcount$ is reset to zero and $rseqn$ is updated based on equation (1).

Before ending this section, it is worth mentionning that, as in Section III-A, the send-out window is respectively inflated or partially deflated in response to a duplicate ACK or to a new ACK. The inflation/deflation process is important to keep the connection active while lost packets are retransmitted. Moreover, both the early retransmission timer and the recovery timer defined in Section III-C and III-D are used in conjunction with the retransmission mechanism based on the loss-monitoring window. Note that, by definition, the early retransmission timer is dedicated to the management of the retransmission of the $(lack + 1)^{th}$ data segment. It is thus reset upon retransmission of this segment only.

V. Loss-resilient eXplicit Control Protocol

This section uses simulations to explore the advantages and limitations of the retransmission mechanisms presented in Sections III and IV in the context of the eXplicit Control Protocol (XCP) proposed in [11]. In Section V-A, we introduce the terminology to denote the combinations of XCP with different retransmission mechanisms. In Section V-B, we discuss the results obtained based on NS simulations, for different protocols and error processes.

A. Loss-resilient XCP protocols definition

In this section we introduce the terminology used to denote the transport protocols obtained when combining XCP with loss retransmission mechanisms.

The acronyms used for the different combinations are defined in Table I. In this table, XCP is used to denote protocols that change neither the packet format, nor the receiver behavior in comparison with the reference protocol defined in [11]. On the contrary, LMXCP assumes that the packet sequence number is conveyed by the packet header to allow for efficient loss monitoring, as described in Section IV-B. We use the prefix LR to distinguish the implementations with early retransmission timer (see Section III-C), and the postfix PA to indicate that partial acknowledgments are used to trigger fast retransmissions (see Section III-B). Eventualy, XCP_dumb refers to the implementation proposed in [11]. It simply relies on the mechanisms implemented by TCP Reno ($cwnd$ decrease in response to duplicate ACKs, and $dupwnd$ reset to zero in response to a new ACK) to recover from losses, without exploiting the advantages provided by an explicit congestion framework w.r.t. loss resilience.

Acronym	Definition
XCP_dumb	eXplicit Control Protocol with retransmission and recovery mechanism implemented as in TCP Reno, similar to [11]
XCP	eXplicit Control Protocol with retransmission and recovery mechanisms adapted to the explicit congestion control framework (see Section III-A)
XCP_PA	XCP + retransmissions based on partial ACKs (see Section III-B)
LR-XCP	XCP + early retransmission timer (see Section III-C)
LR-XCP_PA	LR-XCP + retransmissions based on partial ACKs
LR-LMXCP	XCP + retransmissions based on accurate loss-monitoring (see Section IV)

TABLE I

Acronyms for loss resilient XCP protocols.

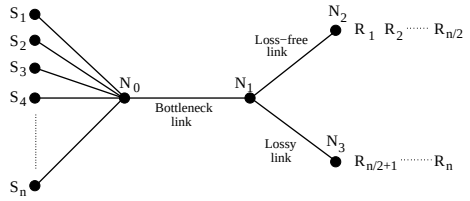
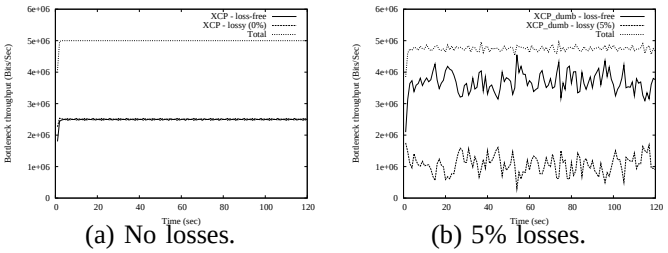


Fig. 1. Network topology reflecting different users accessing a bottleneck through links with different loss characteristics, e.g. wired and wireless.

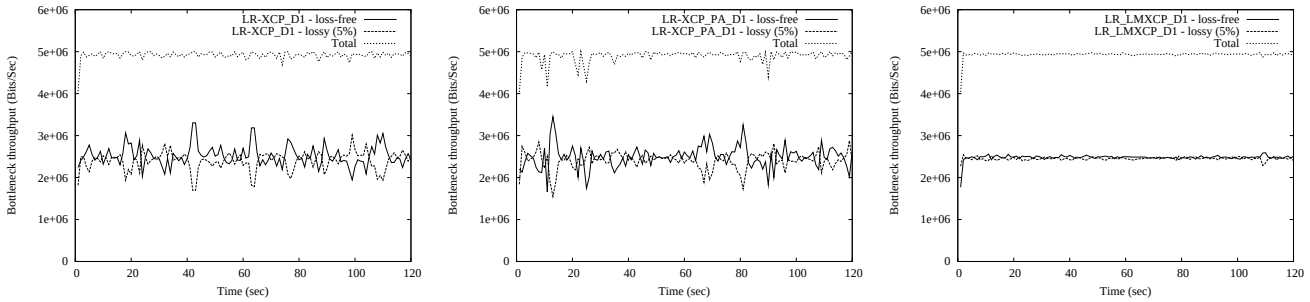


(a) No losses. (b) 5% losses.

Fig. 2. Sums of throughputs measured respectively for loss-free and lossy flows. Throughputs are measured on the bottleneck link as a function of time. By comparing (a) and (b), we observe that the presence of randomly distributed losses on the lossy link causes starvation of the corresponding flows.

B. Loss-resilient XCP simulations

This section presents results obtained based on NS simulations. For simplicity, a single topology is considered throughout the paper. This topology is represented in Fig. 1, where n sources share a bottleneck link. Half of the flows ends up in node N2, through a loss-free link, while the other half ends



(a) LR-XCP (b) LR-XCP_PA (c) LR-LMXCP

Fig. 3. Sums of throughputs measured respectively for loss-free and lossy fbws. Sums of throughputs are plotted as a function of time. In all graphs, the $dupackthreshold$ parameter is set to 1 (as indicated by D1).

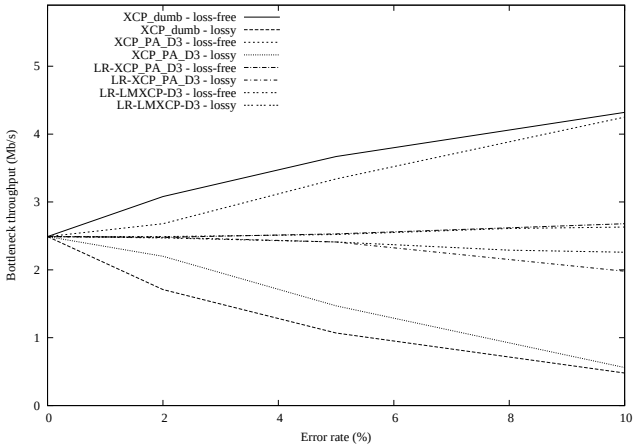


Fig. 4. Sums of average throughputs computed respectively for loss-free and lossy fbws. Throughputs are averaged over a 120s period. Sums of average throughputs are then plotted as a function of the rate of losses generated on the lossy link. $dupackthreshold = 3$

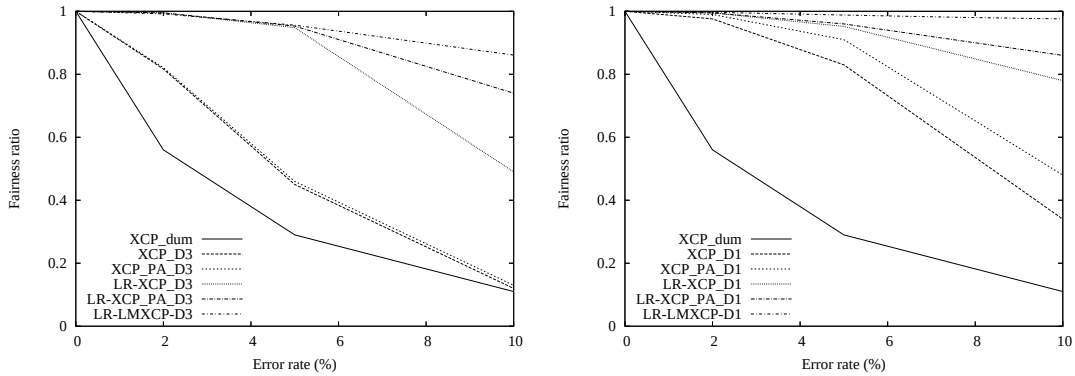
up in node N3, across the lossy link. We refer to the flows that go through the lossy (loss free) link as lossy (loss free) flows. All links have a bandwidth equal to 5 Mbits/sec, and are characterized by the same delay of 10ms, except when explicitely mentioned. In the rest of the paper, the number of sources n is equal to 10, and losses generated on the lossy link are either randomly distributed (=default case), or bursty (see Fig. 9). For most simulations, the performance of loss-resilient protocols are estimated in terms of fairness between lossy and loss-free flows, by comparing the sum of bottleneck throughputs measured for lossy and loss-free flows. Fair usage of the bottleneck reflects good performance of the loss-resilience mechanisms.

Fig. 2 illustrates the problem encountered by the reference implementation of XCP [11] in lossy environments. We observe that the presence of losses causes starvation of the lossy flows. This is because, by default, the XCP loss recovery mechanisms are the one optimized for TCP, and do not exploit the explicit framework specificities. In Fig. 3 and 4, we analyze how the presence of loss-resilient mechanisms dedicated to the explicit control framework prevents this drawback.

In Fig. 3, we observe that all loss-resilient XCP protocols significantly improve the fairness in comparison with Fig. 2(b). We also observe that the accurate monitoring of losses performed by LR-LMXCP mitigates the throughput fluctuations.

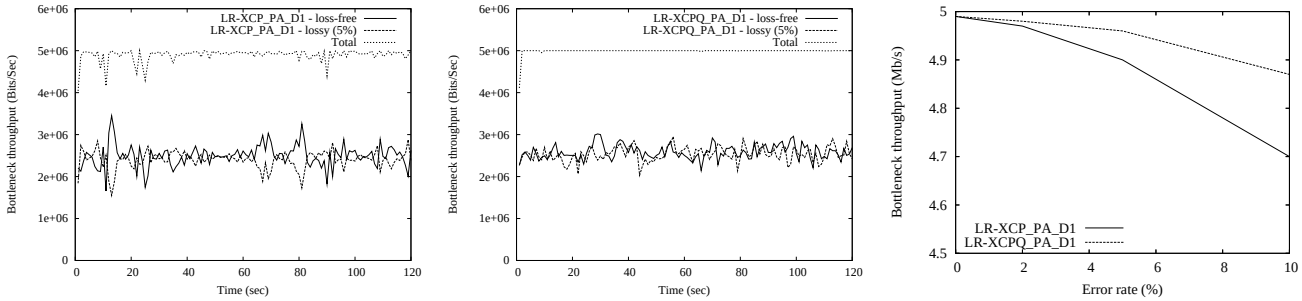
Fig. 4 and 5 compare the performance of the different retransmission mechanisms in details. Fig. 4 compares, as a function of the loss rate, and for the protocols defined in Table I, the sum of bottleneck throughputs corresponding to lossy flows with the one corresponding to loss-free flows. Fig. 5 presents, for the protocols defined in Table I and for two values of the $dupackthreshold$ parameter, the fairness ratio measured between lossy and loss-free flows as a function of the loss rate. The fairness ratio between lossy and loss-free flows is defined as the ratio between the sum of throughputs measured respectively for lossy and loss-free flows on the bottleneck link. We observe on both figures that LR-XCP_PA performs better than XCP_PA. We conclude that the presence of a retransmission mechanism based on a timer brings a significant benefit. Fig. 5 also quantifies the benefit obtained when partial acknowledgments (PA) are used in addition to duplicate ACKs to trigger retransmissions. By comparing (LR-)XCP with (LR-)XCP_PA, we observe that partial acknowledgments mainly help at high loss rates, i.e. when more than one packet is likely to be lost in a single RTT. We also note that, in the absence of retransmission timer, i.e. for XCP and XCP_PA, and when $dupackthreshold = 3$, partial ACKs do not help. This is because in that case, the sender has little chance to enter the fast recovery mode, i.e. to reset the $recover$ parameter (see Section III-B). In addition, Fig. 4 and 5 confirm that a precise monitoring of losses, such as performed by LR-LMXCP, is beneficial. Comparing graphs (a) and (b) in Fig. 5 shows that this is even more true when the $dupackthreshold$ parameter is set to 1. In that case, retransmissions are rapidly triggered by LR-LMXTCP, and losses are rapidly recovered. If needed, several different data segments might be retransmitted in a single round trip time. On the contrary, even with a smaller $dupacktheshold$, LR-XTCP can only consider the retransmission of the $(lack+1)^{th}$ segment, and is thus limited to a maximum of one retransmission per-round trip time.

Before going deeper into the analysis of the different loss resilience mechanisms w.r.t. fairness, it is worth considering the bottleneck link utilization of XCP in the presence of



(a) dupackthreshold=3 (b) dupackthreshold=1

Fig. 5. Fairness ratio measured between lossy and loss-free flows as a function of the loss rate, and $dupackthreshold$ parameter.



(a) LR-XCP_PA (b) LR-XCPQ_PA (c) Bottleneck utilization

Fig. 6. Comparison between an XCP router that minimizes the persistent queue, and an XCP router that targets a persistent queue of 10 packets. In these graphs, the letter Q in LR-XCPQ_PA denotes the use of an XCP router maintaining a persistent queue. (a) and (b) compare for both routers the sums of throughputs measured respectively for loss-free and lossy flows. Losses are randomly generated with a 5% rate. Sums of throughputs are plotted as a function of time. In all graphs, the $dupackthreshold$ parameter is set to 1 (as indicated by D1). (c) plots the bottleneck link utilization as a function of the loss rate for both systems.

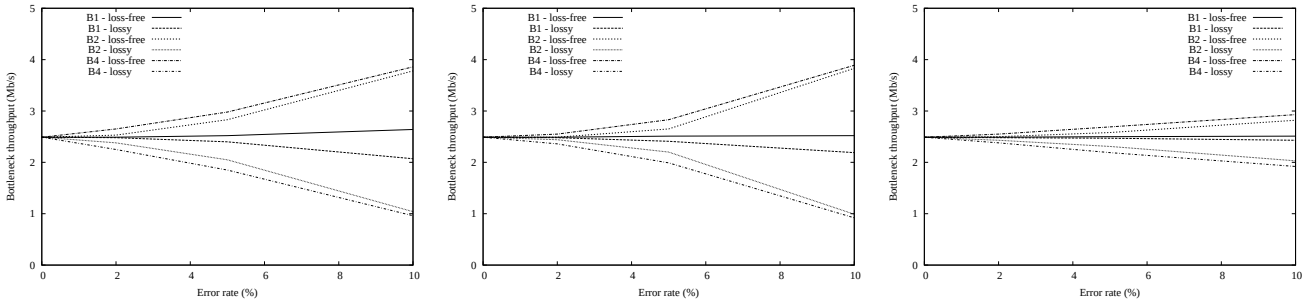
losses. We observe in Fig. 2 and 3 that the total throughput traces do not saturate at 5 Mbits/sec. We conclude that the (loss-resilient) XCP protocols fail to achieve full link utilization. This is confirmed by a deep analysis of Fig. 4. When the loss rate increases, the sum of the loss-free and lossy throughputs corresponding to a given protocol becomes smaller than 5 Mbits/sec. Hence, the total bottleneck link utilization decreases as the loss rate increases. We explain this link utilization deficiency by the small queue sizes maintained by XCP routers [11], which makes them unable to absorb rate fluctuations, e.g. due to a recovery phase caused by packet losses. A way to address this issue is to maintain non-zero persistent queues in XCP routers.

To validate that idea, we have modified equation (1) in [11] so that the efficiency controller targets both maximal link utilization and a non-zero persistent queue. Specifically, Q is replaced by $(Q_{EWA} - \gamma)$ in equation (1) of [11], where γ denotes the size in packets of the targetted persistent queue. Q_{EWA} is defined based on the Q samples as follows. In [11], a Q sample corresponds to the minimum queue seen by an arriving packet during the last propagation delay. This definition results in large fluctuations of Q along the time. To derive a stable signal from Q, we define Q_{EWA} as the exponential weighted average of the Q samples, i.e. each time a new Q sample is generated, Q_{EWA} is set to $\beta.Q_{EWA} + (1 - \beta).Q$. In our simulation, β has been set to 0.9, while the persistent queue γ has been set to 10 packets. The thresholds defining the RED queue policy [10] have been increased to take the persistent queue into account.

Fig. 6 presents the results obtained with and without persistent queues in XCP routers for the $LR - XCP\_PA$ protocol. We observe that the presence of persistent queues in routers preserves the bottleneck link utilization. We conclude that, even when accurate explicit feedback about congestion is available, it is relevant to maintain persistent queues in routers to absorb the unpredictable throughput fluctuations resulting from packet losses, which might for example cause the expiration of the recovery timer.

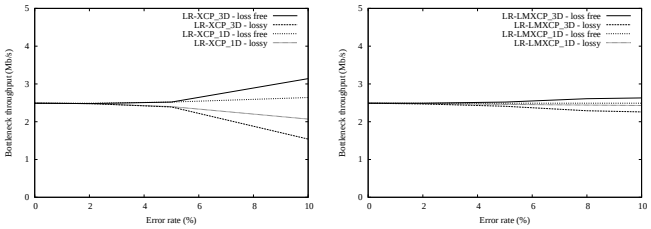
We now concentrate on the impact of protocol and simulation parameters over the performance of LR-XCP and LR-LMXCP. Again, the performance is estimated in terms of fairness between lossy and loss-free flows.

Fig. 7 analyses the impact of the $dupackthreshold$ parameter. This parameter ensures some robustness of the retrans-



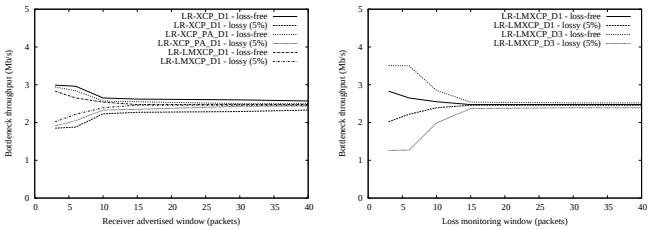
(a) LR-XCP (b) LR-XCP-PA (c) LR-LMXCP

Fig. 9. Impact on fairness of the bursty nature of losses appearing on the lossy link. The acronym BX, with X = 1, 2 or 4, means that each time a loss event happens, X consecutive packets are dropped on the link. The loss rate refers to the product of loss event with the X parameter.



(a) LR-XCP (b) LR-LMXCP

Fig. 7. Impact of the $dupackthreshold$ parameter on the fairness achieved between lossy and loss-free fbws. Traces are defi ned as in Fig. 4.



(a) LR-XCP (b) LR-LMXCP

Fig. 8. Impact on fairness of the constraint imposed on the send-out window by the receiver advertised window. $dupackthreshold$ is set to one, and loss rate is equal to 5%.

mission mechanisms against packet reordering. The larger the parameter, the more robust the protocol. However, a very large parameter would inhibit the retransmission mechanisms based on duplicate or partial acknowledgments. Hence, the choice of $dupackthreshold$ trades off increased robustness to reordering for improved transmission efficiency in presence of losses. Fig. 7 shows that increasing the $dupackthreshold$ from one to three only reduces the performance of loss retransmission mechanisms for high loss rates.

Fig. 8 measures the impact of the constraint imposed on the sender by the receiver advertised window, denoted $rwnd$. This window reflects the receiver buffer capacity, and corresponds to the largest number of out-of-sequence packets that can be buffered at the receiver [13]. It constrains the send-out window of the sender and limits the number of packets the sender can send in advance, while waiting for the recovery of a lost packet. Fig. 8 shows that the performance of the loss-resilience mechanisms only significantly degrades when the constraint on the send-out window becomes of the same order of magnitude as the congestion window. This observation is important because it demonstrates that efficient loss resilient mechanisms do not require large buffer from the end-hosts.

Fig. 9 analyzes how the burstiness of losses affects the retransmission mechanisms. In these simulations, each loss event on the lossy link causes the loss of X consecutive packets, with X ranging from 1 to 4. As expected, we observe that LR-XCP mechanisms are more sensitive to bursts of losses than LR-LMXCP. This is because LR-XCP mechanisms retransmit at most one packet per round trip time, and are thus less efficient than LR-LMXCP when multiple losses occur in the same window of data. Moreover, we observe that the retransmission mechanism based on partial ACK is already beneficial at low loss rates, when losses are bursty.

VI. Loss-resilient eXplicit TCP

In this section, we explore the behavior of our proposed eXplicit TCP in presence of losses, and consider its gradual deployment. Table II presents the acronyms that are used to denote the combination of XTCP with different loss-resilience mechanisms. The last acronym, namely LR-LMXTCP_TCPfriend, is defined in the next section.

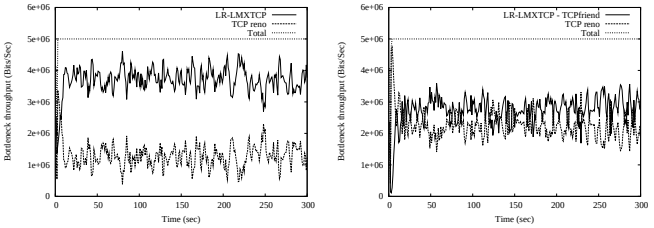
Acronym	Defi nition
LR-XTCP	eXplicit TCP with early retransmission timer (see Section III-C) and recovery mechanisms adapted to the explicit congestion control framework (see Section III-A)
LR-XTCP_PA	LR-XTCP + retransmissions based on partial ACKs (see Section III-B)
LR-LMXTCP	eXplicit TCP + retransmissions based on accurate loss-monitoring (see Section IV)
LR-LMXTCP -TCPfriend	LR-LMXTCP + mechanism to ensure TCP friendliness (see Section VI-A)

TABLE II
Acronyms for loss resilient XTCP protocols.

A. Gradual deployment: joint TCP and XTCP queuing

In this section, we describe a mechanism allowing end-to-end loss resilient XTCP flows to compete fairly with TCP flows. This mechanism allows TCP and loss-resilient XTCP to co-exist in the same network, and provides a possible path for incremental XTCP deployment. To start a loss-resilient XTCP connection, the sender has to check whether the receiver and the routers along the path are XTCP enabled. If they are not, the sender reverts to TCP. As mentioned in [11], these kind of checks can be done using TCP and IP options.

An XTCP-enabled router queues both TCP and XTCP traffics together, in a single queue, but only increments/decrements the XTCP congestion counter when dealing with XTCP packets. The change needed to make a router XTCP-enabled is thus minor. We now extend the design of loss-resilient XTCP to ensure that XTCP flows are TCP-friendly. We illustrate our approach based on LR-LMXTCP.



(a) LR-LMXTCP (b) LR-LMXTCP-TCPfriend

Fig. 10. Throughput traces showing how loss resilient XTCP competes with TCP. (a) LR-LMXTCP is unfair to TCP, (b) LR-LMXTCP with artificial backoff simulations achieves improved fairness.

Fig. 10 plots the sum of throughput measured on the bottleneck link, for TCP and LR-LMXTCP flows respectively. The topology considered for this simulation is the one described in Fig. 1. Routers obey a drop tail policy, and are XTCP-enabled. All links are loss-free. Half of the flows are TCP flows. The others are LR_LMXTCP. In Fig. 10(a), we observe that LR-LMXTCP flows achieve higher throughput than TCP flows. This unfairness is mainly due to the different behavior of LR-LMXTCP and TCP in front of (congestion) losses. LR-LMXTCP handles losses in an efficient way, while TCP generally resorts to a recovery phase when more than one loss occur in a single flight of packets [1].

To validate this interpretation, we propose a simple change to the design of the loss-resilient XTCP sender, so that it triggers an artificial connection backoff when it detects conditions for which TCP is expected to experience a timeout. We use the LR-LMXTCP_TCPfriend acronym to refer to this version of LR-LMXTCP. We now define the artificial backoff procedure, and explain when it has to be triggered.

The artificial backoff simulates the recovery process described in Section III-D. It consists in resetting the congestion window to one, but without resetting the $nextseq$ state variable to $lack + 1$. As in Section III-D, $recphase$ is updated to the largest data sequence number ever sent out by the sender, and the congestion window is not adjusted based on received acknowledgments as long as the $lack$ state variable remains smaller than $recphase$.

The artificial backoff is triggered when (i) a congestion flag is received and (ii) either the congestion window is smaller than $dupackthreshold$, or the previous congestion flag has been received less than one RTT ago. These conditions reflect the fact that a loss ends up in a recovery phase for TCP either when the connection can not enter a fast recovery phase, or when two packets are lost in a single RTT.

In Fig. 10(b), we observe that LR-LMXTCP_TCPfriend competes rather fairly with TCP.

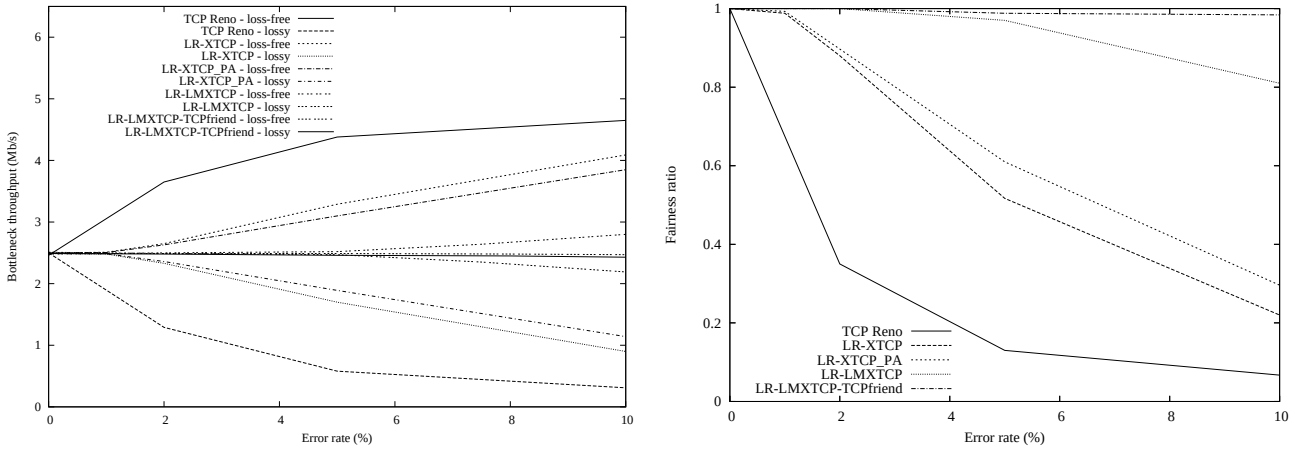
B. Loss-resilient XTCP simulations

We analyze the behavior of loss resilient XTCP protocols, based on simulations, with the topology described in Fig. 1.

Fig. 11(a) plots the throughputs measured over the bottleneck link, respectively for loss-free and lossy flows, as a function of the loss rate experienced on the lossy link. Fig. 11(b) presents the same results in terms of fairness ratio between lossy and loss-free flows. These plots are provided for the five protocols defined in Table II. We observe that TCP rapidly starves the lossy flows. We also observe that the retransmission based on partial ACKs only improves LR-XTCP beyond a sufficient loss rate. Moreover, we note that an accurate feedback, as explored by LR-LMXTCP, brings a significant improvement over approaches that are based on cumulative ACKs. Because LR-LMXTCP preserves high efficiency at high loss rates, we conclude that window-based congestion control protocols, when coupled with a precise feedback from the receiver, are able to support lossy environments. In addition to these general conclusions, we also observe in Fig. 11 that the TCP friendly version of LR-LMXTCP performs even better than classic LR-LMXTCP. The artificial backoffs introduced in the TCP friendly version improve the fairness between loss-free and lossy flows.

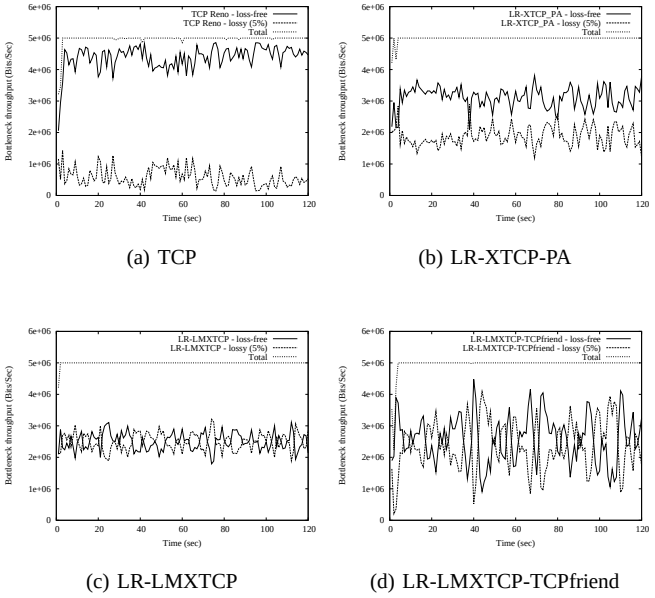
Fig. 12 traces the sum of lossy and loss-free throughputs as a function of time. We make two observations. First, from the total aggregate throughput value, we observe that all loss-resilient XTCP protocols achieve full utilization of the bottleneck link. This is confirmed by Fig. 11(a), where the sums of corresponding loss-free and lossy throughputs equal the bottleneck link bandwidth, i.e. 5 Mb/s. Second, based on graphs (c) and (d), we note that the TCP friendly throughput fluctuates more than the non friendly one. This is in accordance with what we expect from TCP-like connection backoffs.

A last interesting behavior related to the TCP friendliness is illustrated in Fig. 13. This figure plots the bottleneck link utilization as a function of the link delay parameter, when the droptail router queue size is fixed. As expected, the link utilization degrades when the bandwidth-delay product of the connection increases in comparison with the queue size. This is because the queue becomes too small to absorb the reduction of rate due to connection backoffs. We observe in Fig. 13 that LR-LMXTCP preserves higher utilization than TCP or LR-LMXTCP_TCPfriend. This was foreseeable as connection backoffs are more frequent with TCP than with LR-LMXTCP, which is able to face losses without resetting the connection.



(a) Throughput (b) Fairness

Fig. 11. (a) Sum of throughputs measured for lossy and loss-free flows over the bottleneck as a function of the loss rate. Losses are randomly distributed. The D3 postfix indicates that the $dupackthreshold$ parameter is set to 3. (b) Fairness ratio measured between lossy and loss-free flows.



(a) TCP (b) LR-XTCP-PA

(c) LR-LMXTCP (d) LR-LMXTCP-TCPfriend

Fig. 12. Sum of loss-free and lossy flow throughputs as a function of time. Losses are randomly distributed with a loss rate of 5%.

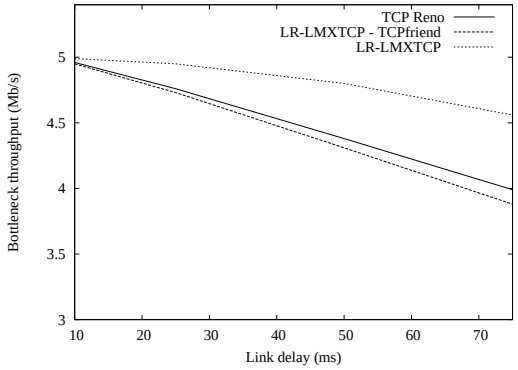


Fig. 13. Bottleneck link utilization as a function of the link delay parameter. Queue size is fixed to 50 packets.

VII. Conclusions

We have studied packet retransmission mechanisms for window-based explicit congestion control protocols. The main objective was to explore whether the window-based transport paradigm, based on the packet conservation principle, was able to prevent connection starvation in the presence of losses.

We first discussed why and how the retransmission mechanisms differ in an implicit or explicit congestion control framework. Explicit control avoids considering any loss as a congestion signal. As a result, retransmissions can be decoupled from the congestion control process. This offers increased flexibility to implement retransmission mechanisms, and allows designing algorithms that substantially improves the performance in comparison with a simple transposition of TCP recovery mechanisms. In particular, we have shown that the introduction of an early retransmission timer brings significant benefit over implementation of TCP-like mechanisms that are suited for the explicit framework. In addition, we have proposed a novel retransmission strategy based on acknowledgments indicating which exact packet triggered the ACK. The proposed scheme appears to be particularly well suited to the explicit congestion framework, and substantially improve the performance obtained based on cumulative ACKs.

As a conclusion, the proposed loss-resilience mechanisms maintain close to optimal link utilization, and bottleneck resources are fairly distributed among lossy and lossless connections. The combination of explicit control with dedicated retransmission mechanisms provides thus an interesting solution to establish reliable and controlled window-based connections in a lossy network. We have validated our approach through simulations, both for the XCP protocol proposed in [11], and for an original explicit TCP protocol. The proposed XTCP protocol has been shown to be able to co-exist with TCP in a single queue, and is thus amenable to gradual deployment.

REFERENCES

[1] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. In *RFC 2581, http://www.rfc-editor.org/rfc/rfc2581.txt*, April 1999.

[2] E. Amir, H. Balakrishnan, S. Seshan, and R. Katz. Efficient TCP over networks with wireless links. In *Fifth workshop on Hot Topics in Operating Systems*, pages 35–40, May 1995.

[3] A.. Bakre and B.R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of ICDCS*, pages 136–143, May 1995.

[4] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, December 1997.

[5] H. Balakrishnan, S. Seshan, and R. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks*, 1(4):469–481, February 1995.

[6] K. Brown and S. Singh. M-TCP: TCP for mobile cellular networks. *IEEE/ACM SIGCOMM Computer Comm. Review*, 27(5):19–43, Oct. 97.

[7] C. Casetti, M. Gerla, S. Mascolo, M. Sanadidi, and R. Wang. TCP Westwood: end-to-end bandwidth estimation for enhanced transport over wireless links. *Wireless Networks*, 8(5):467–479, September 2002.

[8] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *SIGCOMM Comp. Comm. Rev.*, 26(3):5–21, July 96.

[9] S. Floyd and T. Henderson. The Newreno modification to TCP's fast recovery algorithm. In *RFC 2582*, April 1999.

[10] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[11] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for high bandwidth-delay product environments. *ACM SIGCOMM*, pages 89–102, Pittsburgh, August 2002.

[12] S. Keshav and S.P. Morgan. SMART retransmission: performance with overload and random losses. *INFOCOM'97*, 3:1131–1138.

[13] J. F. Kurose and K. W. Ross. Computer Networking: a top-down approach featuring the Internet. Addison Wesley, 2001.

[14] D. Lin and H.T. Kung. TCP fast recovery strategies: analysis and improvements. In *INFOCOM*, volume 1, pages 263–271, March 1998.

[15] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. In *RFC 2018*, October 1996.

[16] C. Parsa and J.J. Garcia-Luna-Aceves. Improving TCP performance over wireless networks at the link layer. *Mobile Networks and Applications*, 5(1):57–71, March 2000.

[17] K.K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification to IP. In *RFC 2481*, January 1999.

[18] Jacobson V. Congestion avoidance and control. In *ACM SIGCOMM*, pages 314–329, Stanford, CA, September 1988.