A Performance vs. Trust Perspective in the Design of End-Point Congestion Control Protocols*

Aleksandar Kuzmanovic† and Edward W. Knightly
Rice University
{*akuzma,knightly*}*@rice.edu*

Abstract

Receiver-driven TCP protocols delegate key congestion control functions to receivers. Their goal is to exploit information available only at receivers in order to improve latency and throughput in diverse scenarios ranging from wireless access links to wireline and wireless web browsing. Unfortunately, in contrast to today's sender-driven protocols, receiver-driven congestion control introduces an incentive for misbehavior. Namely, the primary beneficiary of a flow (the receiver of data) has both the means and incentive to manipulate the congestion control algorithm in order to obtain higher throughput or reduced latency. In this paper, we study the deployability of receiver-driven TCP in environments with untrusted receivers which may tamper with the congestion control algorithm for their own benefit. Using analytical modeling and extensive simulation experiments, we show that deployment of receiver-driven TCP must strike a balance between enforcement mechanisms, which can limit performance, and complete trust of endpoints, which results in vulnerability to cheaters and even DoS attackers.

1. Introduction

Recent advances in TCP congestion control design have demonstrated the ability to significantly improve TCP performance in a variety of scenarios, ranging from high-speed (e.g., [9, 15]) to mobile and wireless networks (e.g., [2, 4]). However, each such advance introduces the following dilemma: if a user can obtain a significant increase in throughput via an optimized congestion control algorithm, how can the network or the other end point distinguish among (i) users with optimized protocol stacks, (ii) "cheater's" that have modified protocol stacks that maximize their own throughput without regard to fairness or network stability, and (iii) attackers that seek only to transmit at a high rate in order to deny service to others. More precisely, the question becomes how can misbehavior be detected in the presence of widely variable protocol performance profiles? And most importantly, protocol innovations often introduce novel security challenges, which, if not considered *a priori*, may have devastating consequences once such innovations become deployed.

TCP variants that are widely deployed today are sender-centric protocols in which the sender performs important functions such as congestion control and reliability, whereas the receiver has minimum functionality via transmission of acknowledgements to the sender. Yet, it is becoming evident that increasing the functionality of *receivers* can significantly improve TCP performance [5, 10, 20, 29, 30, 31]. Indeed, a key breakthrough in this design philosophy is represented by fully receiver-centric protocols in which *all* control functions are delegated to receivers [12, 14]. The benefits that are being established for this innovative design include improved TCP throughput and an array of other performance enhancements: (i) improved loss recovery; (ii) more robust congestion control; (iii) improved power management for mobile devices; (iv) a solution to the handoff problem in wireless networks; (v) improved behavior of network-specific congestion control; (vi) easy migration to a replicated server during handoffs; (vii) improved bandwidth aggregation; and (viii) improved web response times.

However, both sender- and receiver-centric protocols implicitly rely on the assumption that both endpoints cooperate in determining the proper rate at which to send data, an assumption that is increasingly invalid today. With sender-centric TCP-like congestion control, the sending endpoint may misbehave by disobeying the appropriate congestion control algorithms and send data more quickly. Fortunately, the lack of a strong incentive for selfish Internet users to do so (uploading vs. downloading) appears to be the main guard against such misbehavior. Moreover, while it has been discovered that misbehaving *receivers* can perform DoS at-

* This research is supported by NSF Grants ANI-0099148 and ANI-0331620.

† The first author will join the CS Department of Northwestern University in December 2004.

Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP'04)
1092-1648/04 $ 20.00 IEEE

tacks or steal bandwidth even with sender-centric protocols [28], it has been shown that it is possible to modify TCP to entirely eliminate this undesirable behavior [6, 28].

On the other hand, receiver-centric congestion control presents a perfect match for a misbehaving user: the receiving endpoint performs *all* congestion control functions, and has both the incentive (faster web browsing and file downloads) and the opportunity (open source operating systems) to exploit protocol vulnerabilities. In this paper, we explore the tradeoffs and tensions between performance and trust for receiver-centric transport protocols. In particular, given the above benefits (i)-(viii), and clear vulnerabilities, *our goal is to evaluate whether it is possible for HTTP, file, and streaming servers in the Internet to deploy receiver centric transport protocols while striking a balance between performance enhancements and protection against misbehavior.* We focus on the class of receiver-driven protocols because their deployment introduces a set of novel security challenges that can have devastating effects on the widely-deployed HTTP, file, and streaming servers in the Internet. Moreover, we show that *none* of the existing solutions are able to efficiently protect the servers from such receiver misbehaviors.

In this paper, we first anticipate a set of possible receiver misbehaviors, ranging from classical denial-of-service attacks, e.g., receiver request flooding, to more moderate and consequently harder-to-detect resource-stealing manipulations. We analyze misbehaviors that forge the additive-increase-multiplicative-decrease (AIMD) or retransmission timeout (RTO) parameters such that flows steal bandwidth over longer time-scales. Furthermore, we develop an analytical model by generalizing [22] to predict the throughput that a misbehavior will obtain as a function of modified parameters.

Next, we evaluate and point out the main limitations of a set of state-of-the-art router- and edge-based mechanisms designed to detect and thwart denial-of-service attacks and other flow misbehaviors. We then propose and evaluate a set of sender-side mechanisms designed to detect and thwart receiver misbehavior, yet without *any* help from a potentially misbehaving receiver. We initially focus on long time-scales and develop a TFRC-based scheme in which senders (i) independently estimate RTT and loss rate without any cooperation from a potentially misbehaving receiver, (ii) dynamically compute the TCP-friendly rate, and (iii) detect out-of-profile behavior. While this end-point approach at the sender-side is able to accurately detect even slight receiver misbehaviors and strictly enforce TCP-friendliness, we show that a fundamental tradeoff arises from the fact that in the absence of trust between the sender and receiver, it becomes problematic for the sender to infer whether the receiver is misbehaving or legitimately trying to optimize its performance with an enhanced protocol stack.

Finally, we analyze short-time-scale receiver misbehaviors, and show that the performance vs. trust tension significantly magnifies over shorter time-scales. For example, we conduct a web experiment and show that a malicious client that uses excessively long initial window size and also forges exponential backoff timers, can not only significantly improve its own response time, but can also drastically degrade the response times of the background traffic. While sender-based enforcement mechanisms (e.g., rate limiting) are again successful against DoS attacks, we show that in HTTP scenarios dominated by short-lived flows, such mechanisms can often limit receiver-driven TCP performance to a level *below* that achievable by today's sender-based TCP.

2. Background

2.1. Delegating Control Functions to Receivers

One of the first transport protocols that exploits increased receiver functionality is Clark *et al.*'s NETBLT [5], which makes error recovery more efficient by placing the data retransmission timer at the receiver. In later work, an increased set of control functions appear at the receiver, either for performance or practical reasons (e.g., to decrease the computation burden at the sender). For example, Sinha *et al.*'s WTCP [29] calculates the sending rate at the receiver; Floyd *et al.*'s TFRC [10] maintains the loss history and computes the TCP-friendly rate at the receiver; Tsaoussidis and Zhang's TCP-Real [31] tracks loss events and determines the data delivery rate at the receiver; Spring *et al.* [30] and Mehra *et al.* [20] add functionality to the receiver to control the bandwidth shares of incoming TCP flows, i.e., by adapting the receiver's advertised window and delay in transmitting *ack* messages, the receiver is able to control the bandwidth share on the access link according to the client's needs.

2.2. Fully Receiver-Driven Transport Protocols

In contrast to the above protocols, *all* control functions are delegated to receivers in Web Transport Protocol (WebTP) [12] and Reception Control Protocol (RCP) [14]. Hsieh *et al.* [14] argue that the key advantage of fully receiver-centric transport protocols is that the *receiver* controls *how much data can be sent*, and *which data should be sent* by the sender. The benefits that are being established for this protocol design are listed in the Introduction and described in detail in [12, 14].

2.3. RCP Protocol

Here, we provide a brief overview of RCP, variants of which we consider for the remainder of the paper. [1]

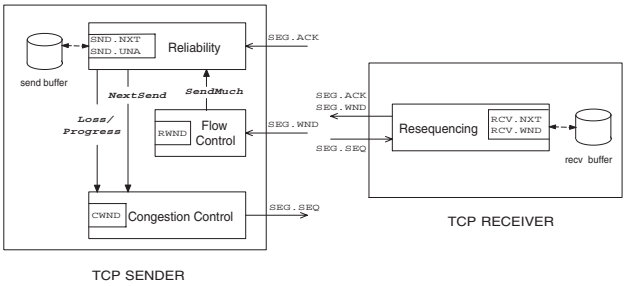

Figure 1. TCP functionalities at the sender and receiver

All TCP variants provide reliable in-sequence data delivery to the application, with protocol operations consisting mainly of four mechanisms: connection management, flow control, congestion control, and reliability. Figure 1 depicts a schematic view of the interaction between sender and receiver in TCP, together with several state variables.

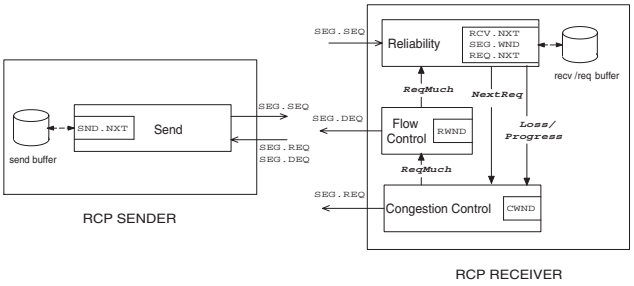

Figure 2. RCP functionalities at the sender and receiver

Observe that except for connection management, which needs to be implemented at both ends, Figure 2 indicates that RCP delegates all other control functions to the receiver. Thus, either the sender or receiver can initiate connection setup, after which the receiver becomes fully responsible for reliability, flow control, and congestion control, using the same window-based mechanisms employed in sender-driven TCP. Since RCP shifts the control of data transfer from the sender to receiver, the *data-ack* style of message exchange in TCP is no longer applicable. Instead, to achieve the self-clocking characteristics of TCP, RCP uses *req-data* exchange for data transfer, where any data transfer from the sender is preceded with an explicit request (*req*) from the receiver. Equivalently, the RCP receiver uses incoming *data* packets to clock the requests for new data. In summary, RCP represents a clone of sender-side TCP which simply transfers *all* important control functionalities to the receiver. (We interchangeably use the terms RCP and receiver-driven TCP.)

However, the fact that *all* control functions are delegated to receivers raises a fundamental security concern for misbehaving receivers that will manipulate protocol parameters (all available at the receiver) and gain significant performance benefits. This concern is amplified by the fact that receivers would have the opportunity (open source operating systems requiring a minor change), and incentive (faster web browsing and file downloads) to perform such activities.

3. Vulnerabilities

3.1. Receiver Misbehaviors

Here, we treat two classes of misbehaviors in the context of receiver-driven transport protocols: denial-of-service attacks and resource stealing. The key distinction between the two lies in the primary goal of the misbehaving client: DoS attackers aim to deny service to the background flows without necessarily achieving a particular benefit for themselves, whereas resource stealers aim to gain a performance benefit by stealing resources from the background flows (without necessarily starving them).

3.1.1. Denial of Service Attacks We begin with an extreme scenario and show that an RCP sender can become an easy target of a DoS attack. Indeed, Figure 2 shows that the RCP sender listens to the request packets from the receiver, and replies by sending data packets without *any* control, as all control functions are delegated to the receiver for performance reasons. Hence, flooding the sender with short *req* packets (the same size as the *ack* packets, $\sim$40 Bytes) may force the RCP sender to flood the reverse path (from the server to the client) with much longer *data* packets (typically $\sim$1500 Bytes), and congest the network.

To demonstrate the vulnerability of fully receiver-driven transport protocols, we simulate the above request-flood attack and show the result in Figure 3. In the experiment, seven TCP Sack flows share a link, and at time 300 sec, an RCP flow joins the aggregate (we provide the exact simulation parameters in Section 5). However, we remove the congestion control functions from the RCP flow (by re-tuning the appropriate RCP parameters at the receiver - details are given below), such that it floods the server with requests.

1 While we focus on RCP, similar receiver incentives and protocol vulnerabilities hold whether protocols delegate some or all control functions to receivers, e.g., TFRC [10] and WebTP [12], respectively.

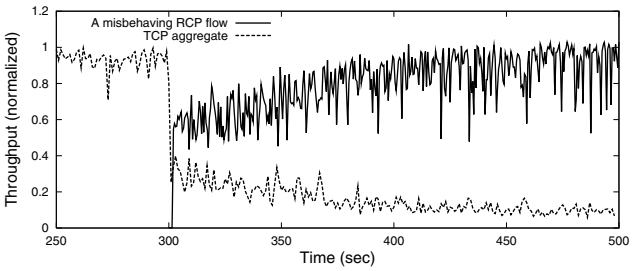

Figure 3. RCP receiver performs a DoS attack by flooding the sender with requests

Consequently, the RCP flow utilizes the entire bandwidth and denies service to the background traffic by exploiting TCP's well-known vulnerability to attacks by high-rate non-responsive flows.

3.1.2. Resource Stealing In contrast, an unscrupulous receiver may moderately re-tune its parameters in an attempt to steal bandwidth from other flows in the network while eluding detection. Indeed, we will quantify the extent to which it is harder to detect flows that moderately disobey some (but not all) congestion control rules (e.g., decrease the window size upon a packet loss, but do not halve it), than it is to detect flows that dramatically violate one or more congestion control rules.

While the space of possible receiver misbehaviors is vast, we focus on parameter-based misbehaviors simply because they are easy to implement. While receivers could clearly use other mechanisms to achieve similar rates, we demonstrate in Section 5 that this does not affect the detection problem. Furthermore, in this paper we do *not* treat the problem of application-level misbehaviors such as parallel download (where a malicious user opens multiple transport-layer connections to parallely download different partitions of a file from a server), which are easier to detect. Nevertheless, observe that the misbehaviors analyzed in this paper are much more generic: (i) they can be simply and entirely implemented at the receivers; (ii) a malicious receiver can achieve a performance benefit even in scenarios where a single transport connection is used for download (e.g., in the HTTP 1.1 web-server scenarios or in the non-partitioned FTP-download scenarios).

The first parameter of interest is the *additive-increase parameter* α, which has a default value of one packet per round-trip time. By increasing the window size more aggressively ($\alpha > 1$), a flow can achieve higher throughput.

The second parameter is the *multiplicative-decrease parameter* β which has a default value of 0.5 such that the congestion window is halved upon the receipt of congestion indication. Again, the receiver can potentially utilize more bandwidth by decreasing the window only moderately via $\beta > 0.5$.

The third parameter is the *retransmission timeout RTO*. Both TCP and RCP use a retransmission timer to ensure data delivery in the absence of any feedback from the remote peer. In both cases, this value is computed using smoothed round-trip time and round-trip time variation. RFC 2988 [27] recommends to lower- and upper-bound this value to 1 and 60 sec, respectively. Thus, a malicious receiver may easily change these values. For example, by setting the *RTO* to a small value (e.g., 100 ms), one can expect to achieve throughput improvements in high packet loss ratio environments, because the misbehaving flow would back-off significantly less aggressively than behaving flows would.

Finally, the fourth parameter of interest is *the initial window size* W. The default is two segments, whereas RFC 2414 [1] recommends increasing this parameter to a value between two and four segments (roughly 4 Kbytes) to achieve a performance improvement. A misbehaving receiver might wish to further improve its performance (without caring much about problems such as congestion collapse), and increase this parameter even more. By doing so, the receiver can maliciously jump-start the RCP flow (this is exactly what we did, among other things, in Figure 3 by setting $W = 10$) and improve its throughput. However, this parameter is expected to be crucial in improving the short file-size response times which are typical for web browsing.

3.2. Modeling Misbehaviors

We begin with the well-known TCP throughput formula (Equation (30) in [22]) that expresses average TCP rate *B* as a function of the round-trip time *RTT*, steady-state loss event rate *p*, TCP retransmission timeout value *RTO*, and number of packets acknowledged by each ack *b* (typically $b = 1$ [13]):

$$B \approx \frac{1}{RTT\sqrt{\frac{2bp}{3}} + RTO\,\min(1, 3\sqrt{\frac{3bp}{8}})p(1+32p^2)}. \quad (1)$$

Using the stochastic TCP model and methodology of [22], we generalize the above result to a scenario with arbitrary values of α and β. Denoting d as $1/\beta$, we have B approximated by

$$\frac{1}{RTT\sqrt{\frac{2bp(d-1)}{\alpha(d+1)}} + RTO\,\min(1, 3\sqrt{\frac{bp(1+d)(d-1)}{2\alpha d^2}})p(1+32p^2)}. \quad (2)$$

We provide the derivation in [16]. Note the two corner cases: for $\alpha = 1$ and $\beta = 0.5$, Equations (1) and (2) are

equivalent; when $\beta = 1$ (when $d = 1$), then $B \to \inf$, i.e., if the congestion window is never decreased upon a packet loss, the throughput will theoretically converge to infinity. We explore intermediate cases as follows.

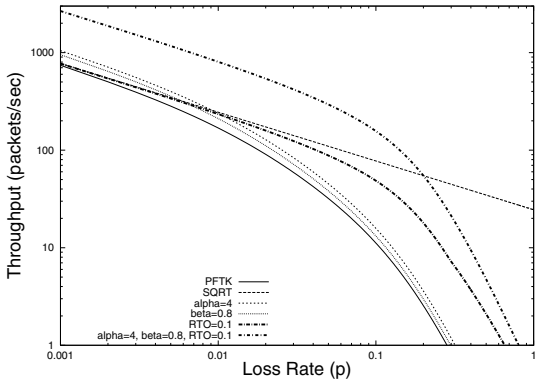

Figure 4. Long-time-scale misbehaviors - numerical results

Figure 4 shows numerical results for TCP (and hence RCP) throughput as a function of the packet loss rate. *PFTK* denotes the formula from [22] (Equation (1), with $b = 1$ and $RTO = 1$), while *SQRT* is the "square-root" formula from [19] (the same as Equation (1), only without the *RTO* part). Next, we plot the throughput that a malicious receiver can achieve, according to the Equation (2), by manipulating α, β, and *RTO* (exact values are shown in the figure).

First, observe that by re-tuning α to four, one can double the throughput (*y*-axis is in logarithmic scale), while re-tuning β to 0.8 ($d = 1.25$) one can steal somewhat less bandwidth. More generally, according to Equation (2), setting α to a value larger than one, enables a flow to achieve approximately $\sqrt{\alpha}$ higher throughput as compared to a well-behaved TCP flow and for the same packet loss rate. Second, notice that the amount of stolen bandwidth (the difference between the misbehaving and the *PFTK* curve) increases as the packet loss ratio increases in the case of the *RTO* parameter (e.g., $RTO = 100$ ms). This is because timeouts occur more frequently in higher packet-loss-ratio environments, and thus, disobeying the exponential backoff rules enables significant throughput gains in such environments. Furthermore, by re-tuning all parameters together ($\alpha = 4$, $\beta = 0.8$, $RTO = 0.1$), the model predicts significant stealing effects, where the misbehaving flow utilizes approximately ten (for $p = 0.02$) to twenty (for $p = 0.1$) times more bandwidth than behaving flows. Finally, observe that the *SQRT* formula significantly overestimates the TCP-friendly rate for higher packet loss ratios (where the exponential backoffs play a key role), hence this formula is not suitable for detection purposes.

4. Network Solutions

Here we analyze several state-of-the art network solutions (both core- and edge-based) designed to detect malicious flows. Common to all solutions is their fundamental limitation to accurately detect such flows due to their lack of the knowledge of the actual flows' parameters.

4.1. Core-Router-Based Solutions

4.1.1. RED-PD In [18], Mahajan *et al.* develop RED-PD, a scheme that uses the packet drop history at a router to detect high-bandwidth flows in times of congestion, and preferentially drop packets from these flows. In order to detect high-bandwidth flows, RED-PD sets a *target bandwidth* above which a flow is identified as malicious. The target bandwidth is defined as the bandwidth obtained by a *reference* TCP flow with the *target RTT* (default is 40 ms), and the current drop rate measured at the output router queue. The targeted bandwidth is computed using the square-root TCP-friendly formula. In other words, in the absence of per-flow RTT measurements, RED-PD sets the target RTT to 40 ms as a bound for distinguishing in- vs. out-of-profile flows.

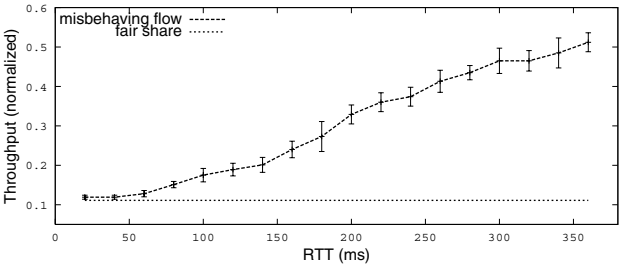

Figure 5. RED-PD is unable to detect a malicious flow

While RED-PD can protect the system against certain misbehaviors, the lack of exact knowledge of the flow's RTT fundamentally limits its ability to detect severe end-point misbehaviors as demonstrated in Figure 5. We perform *ns* experiments with nine flows sharing a RED-PD router. We vary the round-trip times of the flows from 20 to 350 ms (as shown on the *x*-axis), and plot the bandwidth of a single flow on the *y*-axis. When all flows are well-behaved, the bandwidth share is fair (the straight line in the figure). However, when one of the flows (whose normalized throughput is shown on *y*-axis) re-tunes α to 25, it can potentially steal up to five times more bandwidth than its fair share according to Equation (2). Observe that RED-PD successfully limits the malicious flow to its fair-share, but only when the RTT is less than or equal to 40 ms (recall that this

is the RTT of the *reference* flow). However, as the flows' RTT increases, the malicious flow is able to steal more and more bandwidth, up to five times more than its fair share (the maximum for this scenario) when the RTT is 350 ms.

RED-PD's limitations in detecting misbehaving flows are more general than indicated in the above example. First, it is important to notice that a misbehaving flow can steal bandwidth not only in homogeneous-RTT scenarios as in the above experiments, but also in heterogeneous-RTT environments, since the amount of stolen bandwidth depends on the RTT of a misbehaving flow. Second, while in this paper we focus on receiver-driven transport protocols, observe that the above RED-PD limitations apply equally to sender-based TCP stacks. Another problem arises from the fact that RED-PD uses a simple (and less accurate) square-root formula, which significantly overestimates the TCP-friendly rate for higher packet loss ratios because it doesn't account for retransmissions [22]. Hence, malicious TCP or RCP flows have the opportunity to steal dramatically more bandwidth as the packet loss ratio increases, e.g., 100 times more when $p = 0.3$, as indicated in Figure 4.

Finally, RED-PD's inability to determine with high confidence if a flow is malicious or not, limits its ability to punish a malicious flow (e.g., to completely starve it). Hence, "stealing pays off" for endpoints as they can freely re-tune their parameters without adverse effects: (i) they will not be completely starved; (ii) they will not utilize less bandwidth than a well-behaving TCP or RCP would; and yet (iii) they can quite often steal significant amounts of bandwidth.

4.1.2. Fair Queuing While it may appear attractive to apply some version of fair queuing (including the preferential-dropping schemes developed to enforce fairness among adaptive and non-adaptive flows, e.g., Flow Random Early Detection (FRED) [17], CHOKe [24], or Stochastic Fair Blue (SFB) [8]) to solve the above problem, observe that such schemes are also unable to detect end-point misbehaviors and to enforce the proportional fairness targeted by TCP. Moreover, in a heterogeneous RTT environment, such schemes will significantly deviate from the proportional bandwidth share, and even magnify the bandwidth-stealing effects. Below, we provide a simple, yet illustrative example. While not representative of an actual or realistic scenario, our main goal is to illustrate the difference between proportional (RTT-dependent) and max-min fairness as enforced by FQ.

Consider a link shared by three congestion-controlled flows, such that the proportional fair share is (0.9,0.05,0.05). Next, assume that flow 2 is malicious. It re-tunes its parameters and utilizes more bandwidth by stealing from flow number one, such that the bandwidth share is now (0.7, 0.25, 0.05). However, if FQ is used, all flows get their "fair-share", and the bandwidth share is now (0.33, 0.33, 0.33). Thus, FQ provides even more bandwidth to flow 2 than it could have stolen without it.

4.2. Edge-Router-Based Solutions

Here, we present two solutions whose goal is to detect non-TCP-friendly behavior at the network edge. The key advantage of an edge-based vs. a network-based scheme is the opportunity to monitor packets in both directions (*data* in forward, and *ack* in reverse).

4.2.1. D-WARD In [21], Mirkovic *et al.* develop D-WARD, an edge-router based protection scheme for detecting DoS activity. For each traffic type, they establish a baseline traffic model. For a TCP session, they measure both outgoing (*data*) and incoming (*ack*) traffic and define the maximum allowable ratio of the two. When the ratio of the number of data vs. the number of ack packets goes over a certain threshold, they conclude that the flow is out of profile and rate-limit it.

While the above scheme may indeed protect against TCP-based denial-of-service attacks (where the sender floods the network with data packets independent of the feedback from the receiver), this model clearly doesn't apply to the *receiver-driven* TCP scenario. Recall that in the receiver-based scenario, the number of requests and data packets is the same in both directions, even in the most severe denial-of-service scenarios. Moreover, the fact that the number of packets in the forward (*data*) and reverse (*req*) directions is the same is actually the core idea of the request-flood attack: the receiver floods the sender with requests, and the sender replies by transmitting the same number of data packets, yet with significantly larger size thereby congesting the network.

4.2.2. Tcpanaly In [26], Paxson presents `tcpanaly`, a tool whose initial goal was to work in *one pass* over a packet trace by recognizing *generic* TCP actions. The goal of executing only one pass stemmed from the objective that `tcpanaly` might later evolve into a tool that could monitor an Internet link in real-time and detect misbehaving TCP sessions on the link. Unfortunately, the author was forced to abandon both of the goals. Among many obstacles, the key one is that one-pass analysis proved difficult due to vantage point issues (see reference [26] for details), in which it was often hard to tell whether a TCP flow's actions were due to the most recently received packet, or one received in the distant past.

5. An End-Point Solution

5.1. Sender-Side Verification

In order to detect receiver misbehavior, the sender requires increased functionality beyond its role as a slave to the receiver's request packets (see Figure 2). Our objective is to add the minimum functionality to the sender that will enable it to robustly detect receiver misbehavior over long-time scales (we treat the short-time-scale misbehavior detection problem in Section 6.1), yet without *any* help from a potentially misbehaving receiver. While this new functionality inevitably increases the sender-side implementation complexity, we will demonstrate that it represents a general solution to the bandwidth-stealing receiver-induced misbehaviors.

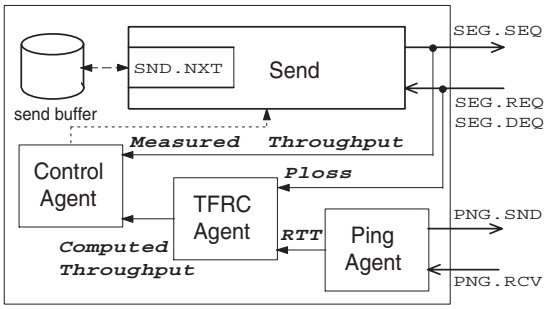

Figure 6. Secure RCP sender

Figure 6 depicts the key components of such a solution. Equation (1) indicates that knowledge of RTT and packet loss ratio is enough to compute the TCP-fair throughput, and consequently to detect out-of-profile flows. Unlike in network-based scenarios discussed in Section 4, an end-point scheme can measure RTT and the packet loss ratio, and hence enforce a more precise traffic profile than *any* network-based solution.

Because the sender must estimate RTT and packet loss ratio without any cooperation from the untrusted receiver, the sender transmits *ping* packets that the receiver has no incentive to delay, as a larger RTT implies a lower bandwidth profile.[2] Likewise, the sender must estimate the packet loss ratio and detect whether the receiver is actually re-requesting data packets that are dropped. Observe that a node performing a DoS attack need not re-request dropped packets, whereas receivers that are stealing bandwidth will be forced to re-request packets for a reliable service. In any case, one possible solution to the above problem is for the sender to purposely drop a packet to test if the receiver will re-request it as the absence of a repeated request for the dropped packet would indicate a potential DoS attack. Note that this is a backward-compatible technique that could be used instead of the proposed *nonce* technique [6]. Nevertheless, here we focus on bandwidth-stealing scenarios where the receivers are forced to re-request dropped packets for a reliable service.

Once the RCP sender estimates RTT and the packet-loss-ratio, it can compute the TCP-friendly rate. However, because these parameters can vary significantly during a flow's lifetime, we apply the methods developed for TCP-Friendly Rate Control (TFRC) [10] to estimate the TCP-friendly rate in real time. Namely, while existing use of TFRC focuses on setting the transmission rate based on RTT and loss measurements, we utilize TFRC to *verify* TCP friendliness using the actual RTT (measured via the *ping agent*) and loss measurements incurred by the RCP flow itself.

In [25], Patel *et al.* designed an end-point scheme whose goal is to verify TCP friendliness in the context of untrusted mobile code. The key difference between our scheme and the one from [25] is that our scheme aims to thwart possible *receiver* misbehaviors, and hence does not require any cooperation from a potentially malicious receiver. Moreover, in contrast to the scheme from [25], which compares the TCP sending rate to the TCP-friendly equation rate [22], our scheme applies the TFRC protocol to estimate the TCP-friendly rate in real time. This is particularly important in the presence of highly dynamic background traffic; while being an equation-based scheme, TFRC manages to adapt to relatively short time-scale available-bandwidth fluctuations [3].

Finally, by comparing the measured throughput (based on the number of packets sent) and the throughput computed by the *TFRC agent*, the *control agent* is able to detect, and eventually punish, a misbehaving receiver. We do not implement the control module in this work, as our primary goal is to explore the ability of the above scheme to accurately *detect* receiver misbehaviors. Alternatives to punish include rate-limiting and preferentially dropping packets. However, given that the scheme can indeed accurately detect misbehaving receivers (to be shown below), the sender may simply disconnect the misbehaving client, and in that way discourage potentially malicious receivers from the temptation to steal bandwidth.

5.2. Detecting Misbehaviors

5.2.1. TFRC Agent To robustly detect misbehaving receivers, it is essential to first evaluate the TFRC agent's

2 If the receiver doesn't reply to the *ping* requests, the sender may either disconnect it, or rate-limit it to a moderate rate. Moreover, to prevent the receiver to simply send a response in anticipation of a *ping* request (thus thereby simulating a smaller RTT), the sender should randomize the period between the *ping* messages.

accuracy in measuring TCP friendliness. Computed TFRC throughput may deviate from actual TCP throughput due to measurement errors (low RTT sampling resolution, *ping* packets sent once per second, etc.), system dynamics, and inaccuracies in the underlying TCP equation. Thus, to manage the detection scheme's false positives (incorrect declaration of a non-malicious flow as malicious), such inaccuracies must be incorporated into the detection process.

We conduct *ns* simulation experiments and consider a link shared by a number of TCP Sack flows (varied from 1 to 600). The link implements RED queue management and has capacity 10 Mb/s; we set the buffer length, min_thresh, and max_thresh to 2.5, 0.25 and 1.25 times the bandwidth-delay product, respectively. The round trip time is 50 ms. Unless otherwise indicated, these parameters are used throughout the paper. We perform a number of simulations, and present average results together with 95% confidence intervals.

To establish a baseline of TFRC's behavior, we first mount the TFRC agent on the sender side of a *sender-based* TCP Sack [11] flow and present the results in Figure 7. The figure depicts the ratio of measured (TCP Sack) vs. computed (by the TFRC agent) throughputs as a function of the packet loss ratio. When the measured vs. computed throughput ratio is one, this indicates that the TFRC agent exactly matches the TCP Sack throughput. Observe that this is indeed the case for low packet loss ratios (for the curve labeled as "TCP Sack"). As the packet loss ratio increases, the curve moderately increases, indicating a slight conservatism of the TFRC agent as the throughput computed by the TFRC agent is slightly lower than the measured TCP Sack throughput. The problem of TFRC conservatism has been studied in depth in reference [32]. In summary, the throughput computed by the *TFRC agent* deviates from the TCP Sack throughput, yet the deviation is moderate, even for high packet loss ratios.

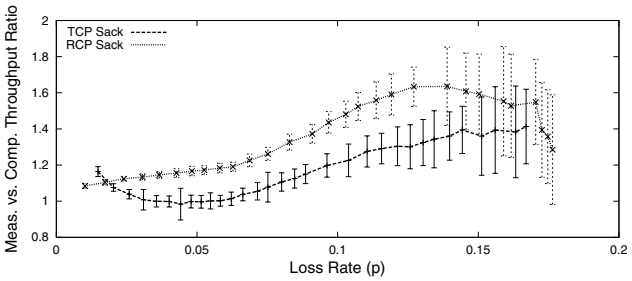

Figure 7. TFRC agent mounted on the sender side of a well-behaved (a) TCP Sack and (b) RCP Sack

Finally, we repeat the above experiment, but now mount the TFRC agent on the RCP sender as in Figure 6. Observe that the ratio of the measured (RCP Sack) vs. computed throughput is somewhat higher than in the above sender-based TCP Sack scenario. Indeed, RCP Sack has an improved loss recovery mechanism (see reference [14] for details) and consequently improves throughput. The key problem is the sender side's difficulty in determining whether the receiver is trying to optimize its performance, or is simply stealing bandwidth. We treat this problem in detail in Section 5.3. Here, we obtained the reference measurement-based profile for a behaving RCP flow, which we will next use to demonstrate the capability of an end-point scheme to detect even moderate receiver misbehaviors.

5.2.2. Detecting Misbehaving Receivers Here, we implement a misbehaving RCP node that re-tunes its congestion control parameters α, β, and *RTO* at the receiver.

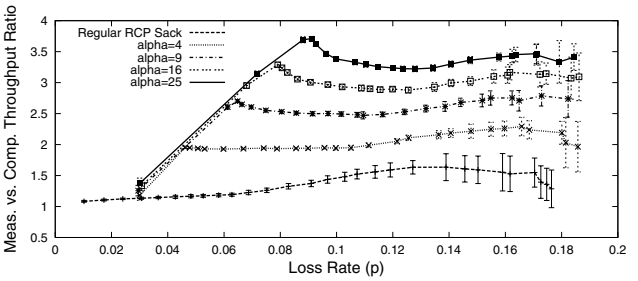

Figure 8. Misbehaving receiver re-tunes the additive-increase parameter α

We first re-tune the additive-increase parameter α and repeat the experiment above. Figure 8 depicts the measured vs. computed throughput ratio for misbehaving receivers (having α of 4, 9, 16 and 25), together with the same ratio for the behaving RCP flow having $\alpha = 1$. Recall that the left-most point on the curve corresponds to low loss and experiments in which the RCP flow competes with a single TCP Sack flow, whereas the right-most point on the curve corresponds to high loss and a single RCP flow competing with 600 TCP Sack flows. Observe first that the measured vs. computed throughput ratios for misbehaving flows clearly differ from the behaving flows' profile, indicating a strong potential for misbehavior detection (to be demonstrated below). Second, observe that the throughput ratio for misbehaving flows is approximately proportional to $\sqrt{\alpha}$ as predicted by the model except for extremely low aggregation regimes (e.g., $p = 0.03$ in which a single RCP flow competes with a single TCP Sack flow). In such low aggregation cases, while the misbehaving flow indeed takes significantly more bandwidth than the competing TCP Sack flow (not shown), it is unable to fully utilize the bandwidth due to frequent backoffs.

Next, we explore misbehaviors that re-tune the multiplicative-decrease parameter β and the retransmission timeout parameter *RTO*. Due to space constraints, we skip the results and point an interested reader to reference [16]. The most interesting result is certainly the one showing that by retuning the *minRTO* and *maxRTO* parameters simultaneously, it is possible to transform RCP (and TCP) into a powerful DoS tool.

5.2.3. Detection Threshold Here we evaluate the sender's ability to detect receiver misbehaviors and study the false-alarm probability and correct misbehavior-detection probability. Denote *meas_thr* as the throughput measured by the RCP sender, and *comp_thr* as the throughput computed by the TFRC agent (as shown in Figure 6). Next, denote k as the threshold parameter, and define $P(k)$ as

$$P(k) = Prob(\frac{meas\_thr}{comp\_thr} > k). \quad (3)$$

For example, $P(1)$ denotes the probability that the measured vs. computed throughput ratio is larger than one, whereas $P(2)$ is the probability that the the measured throughput is more than twice the computed one. If the receiver is behaving, then $P(k)$ is the *false-alarm* probability (i.e., we falsely conclude that the receiver is misbehaving with probability $P(k)$). On the other hand, if the receiver is misbehaving, then $P(k)$ is the *correct misbehavior-detection* probability (i.e., we correctly conclude that the receiver is misbehaving with probability $P(k)$).

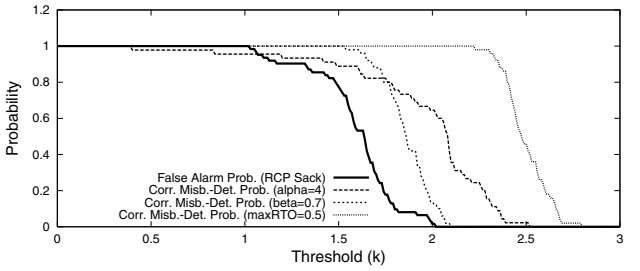

Figure 9. Detecting out-of-profile flows

Figure 9 plots the false alarm probability (for the behaving RCP flow), together with the correct misbehavior-detection probabilities for three moderately misbehaving receivers (exact parameters are shown in the figure). We set the packet loss ratio to 0.15 representing a scenario in which the throughput ratio deviates (approximately) the most as indicated in Figure 7. Consequently, the false-alarm probability for the behaving RCP flow is largest, indicating that this scenario is the most challenging from the detection point of view.

The key observations from Figure 9 are as follows. First, note the tradeoff in setting the threshold parameter k. If it is too small (e.g., $k = 1$), we are able to detect the misbehaving receivers with high probability, but the false alarm probability is also one. On the other hand, if it is set too high (e.g., $k = 3$), the false alarm probability becomes zero, but the correct misbehavior-detection probability also becomes zero. However, observe that the fact that the false-alarm probability decreases faster (for smaller k), makes it possible to set the threshold (e.g., $k = 1.8$ in this scenario), such that the false positives are acceptably small, yet we are able to detect *all* of the above cheaters with high probability. Thus, this *worst-case* scenario confirms the high precision of the end-point scheme in detecting a wide range of receiver misbehaviors. However, we will next show that setting the parameter k incurs an additional challenge when confronted with versions of TCP employing performance enhancements.

5.3. Advanced Congestion Control Mechanisms

There is a significant body of work proposed to improve the TCP performance in wireless environments, where high channel losses may disproportionately degrade TCP Sack performance. Here, we briefly explain two well-known protocols, TCP-ELN [2] and TCP Westwood [4]. TCP-ELN has been proposed to distinguish wireless random losses from congestion losses. It relies on an external trigger to classify the losses, and fast retransmits lost segments due to wireless errors without decreasing down the congestion window. It has been shown in [14] that when this mechanism is applied in the *receiver-driven* protocol scenario, the throughput improvements are quite significant (we repeat this experiment and confirm the result below). Another protocol that significantly improves the throughput over wireless links is TCP Westwood (see details in [4]), and it is expected that the same mechanism could provide further throughput improvements in receiver-driven protocols. Below, we focus on RCP-ELN and do not further consider sender- or receiver-based TCP Westwood.

We first simulate an RCP-ELN flow in a lossy wireless-like environment. Figure 10 depicts the measured vs. computed throughput ratio as a function of loss. Observe that the RCP-ELN throughput ratio increases significantly as compared to the RCP Sack profile, indicating that RCP-ELN indeed significantly improves throughput, e.g., achieving a six-fold increase for a loss ratio of 0.17. However, the key problem is that from the sender perspective, the RCP-ELN flow is difficult to distinguish from a misbehaving flow.

Figure 11 depicts the false-alarm probability for the behaving RCP-ELN flow for a packet loss ratio of 0.15. To

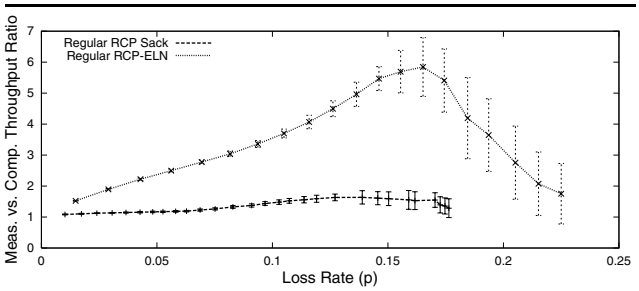

Figure 10. RCP-ELN significantly improves throughput

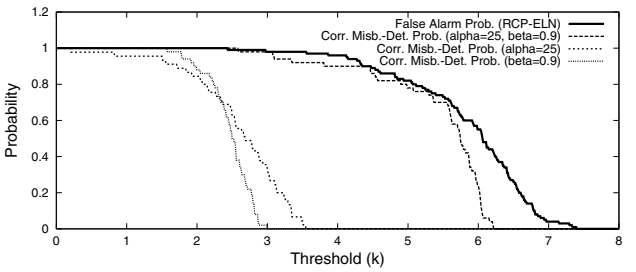

Figure 11. From the sender's perspective, RCP-ELN looks like a misbehaving flow

emphasize the detection problem, we also plot the correct misbehavior detection probabilities (without any advanced congestion control mechanisms), with maliciously re-tuned parameters (i) $\alpha = 25$, (ii) $\beta = 0.9$, and (iii) $\alpha = 25$ and $\beta = 0.9$. Observe that using a small threshold (e.g., $k = 1$) ensures a high detection probability for any of the above misbehaviors, but we also falsely detect the RCP-ELN as malicious. However, simply increasing the threshold *k* does *not* eliminate the problem. For example, for $k = 4$, the false alarm probability for ELN-RCP is still one, while the probability to detect misbehaviors (i) and (ii) has already dropped to *zero*. Finally, by using a very large *k* (e.g., $k = 7$ in this scenario), we have an acceptably small false alarm probability for RCP-ELN, but are at the same time unable to detect any of the (quite severe) receiver misbehaviors.

Thus, these experiments illustrate a fundamental tradeoff between system performance and security (the ability to detect bandwidth stealers), as both cannot be maximized simultaneously. Ironically, while advanced congestion control mechanisms at the receiver significantly improve throughput, the resulting false-alarm probability further increases, further emphasizing the tradeoff. We believe that setting the parameter *k* to a larger value strikes the best balance for the file- or streaming-servers in the Internet. A large value protects servers from severe denial-of-service attacks, while enabling innovation in protocol design by preserving the performance benefits of receiver-centric transport protocols. The downside is the fact that we are unable to detect some bandwidth stealers. In contrast, strictly enforcing today's TCP-Sack throughput profile via a lower *k* would indeed make it possible to catch even modest bandwidth stealers. However, a small *k* would remove most of the RCP benefits, and indeed remove the incentive for designing and deploying enhanced TCP stacks.

6. Short Time Scale Misbehavior

The secure RCP sender is designed to detect receiver manipulations of congestion control parameters (e.g., α, β, *RTO*) that would enable the receiver to steal bandwidth over longer time periods. Hence, these misbehaviors can be detected on longer time-scales. However, very short-lived flows transmitting up to tens or hundreds of packets are common in today's Internet due to web traffic. Below, we treat the problem of short time-scale misbehaviors and discuss possible solutions.

6.1. Initial Congestion Window

We consider web RCP flows that increase their initial congestion window in order to obtain decreased response time. The web-browsing simulation scenario consists of a pool of clients and a pool of web-servers, while the bottleneck link is 10 Mbps. We adopt the model developed in [7] in which clients initiate sessions from randomly chosen web sites (the server pool) with several web pages downloaded from each site. Each page consists of several objects, which are downloaded by either TCP or RCP, depending on the client (all the servers in the pool support both options). There is a *single* misbehaving client in the client pool, which uses a mis-configured RCP (details are given below), while the other clients from the pool behave and use unmodified TCP Sack.

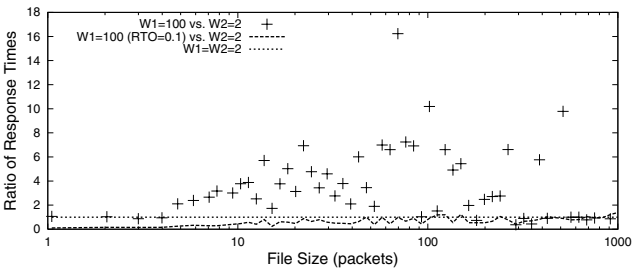

Figure 12. A greedy receiver can significantly degrade legitimate background web traffic

While it may appear attractive for a malicious client to maximally increase the initial window size parameter *W* in order to steal more and more bandwidth, this is not necessarily a good option, especially in more congested environments. This is illustrated in Figure 12, where we set the link utilization to 90%, and the malicious clients sets the initial window size parameter *W* to 100 packets. Here, this greedy user significantly degrades not only the background traffic (not shown), but also degrades its *own* response times (shown in the figure) by an order of magnitude. This degradation is due to the fact that when the malicious user sends large bursts of requests, it forces the web server to reply with large bursts of data packets, many of which are themselves lost in the congestion. These packet losses force even the RCP user to enter the exponential backoff phase and degrades its response time. To overcome the above problem, the malicious user needs to "turn off" the exponential backoff timers. We do this by re-tuning the *RTO* parameter to 100 ms. In this way, the malicious user is able both to "push-out" and significantly degrade the background traffic, and at the same time improve its own response times, as also shown in the figure.

6.2. Solutions

Here, we explore two possible solutions to the above short-time-scale misbehaviors. The first is to rate-limit flows, which while effective in thwarting cheaters, is a non-work conserving solution in which it is problematic to determine the appropriate rate. The second solution is to have a "smart" RCP client at the sender side that would enforce a "TCP-friendly" exponential window increase. It would estimate the RTT to the client, and release the *data* packets accordingly. While also effective in thwarting cheaters, this approach unfortunately mitigates some of the benefits of RCP.

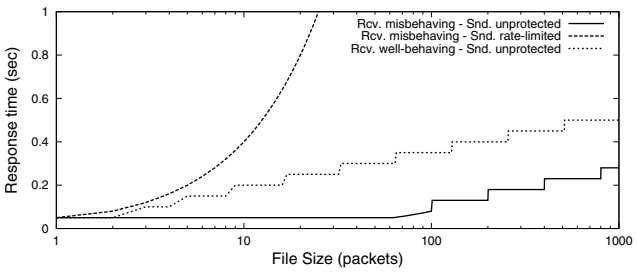

Figure 13. Protecting against short-time-scale misbehaviors

To study the performance of the above solutions, we compute and plot in Figure 13 the file-response times in three different scenarios for the RCP flow with the available bandwidth of 10 Mb/s and *RTT* of 50 ms: (i) when a malicious user sets the initial window *W* to 100 packets and the sender does not rate limit (labeled as "Rcv. misbehaving - Snd. unprotected"); (ii) the receiver sets $W = 100$, but the sender rate limits to 200 kb/s ("Rcv. misbehaving - Snd. rate-limited") and (iii) the receiver is well behaving and is not rate-limited ("Rcv. well-behaving - Snd. unprotected"). Figure 13 illustrates problems in setting the rate-limit value. Setting it to 200 Kb/s degrades the file response times significantly, as shown in Figure 13.

But the key insight from the above experiment is that using a large initial window sizes can significantly (up to *ten* times in the above scenario - and much more in larger-bandwidth networks) improve file response times. Such methodologies have been studied in depth in [23, 33, 34], but in the context of sender-based TCP, where the web-server increases the initial window size in an attempt to improve system performance. However, in the receiver-driven RCP scenario, it is hard to distinguish whether the receiver is jump-starting the TCP flow or is simply malicious. Thus, applying rate limiting or the "smart" RCP client methodology may indeed protect the system against receiver misbehavior, but at the same time prevents attempts as in [23, 33, 34] to improve performance. This illustrates the tradeoff between system security and performance in that strict enforcement of protocol rules would not only reduce performance, but would also inhibit protocol innovation.

However, either rate-limiting or a "smart" RCP client has to be *strictly* applied, because a receiver with an excessively large *W* in combination with manipulated exponential backoff timers can significantly degrade the legitimate background traffic (Figure 12). Yet, applying any of the short-time-scale protection methodologies inevitably reduces the incentive for receivers to use RCP for short-lived flows, as sender-based TCP enhanced with jump-starting methodologies is able to achieve the best response-time curve from Figure 13 without any security considerations.

7. Conclusions

Receiver-driven transport protocols delegate key control functions to receivers. While this radically new protocol design achieves significant performance and functionality gains in a variety of wireless and wireline scenarios, we showed that a high concentration of control functions available at the receiver leads to an extreme vulnerability. Namely, receivers would have both the means and incentive to tamper with the congestion control algorithm for their own benefits. We analyzed a set of easy-to-implement receiver misbehaviors and analytically quantified the substantial benefits that a malicious client can achieve.

We evaluated a set of state-of-the-art network-based solutions, and proposed and analyzed a set of end-point solutions. Our findings are as follows. (1) Network-based solutions are fundamentally limited in their ability to detect and punish even severe endpoint misbehaviors. (2) Endpoint solution can accurately detect long-time-scale receiver misbehaviors and strictly enforce the TCP-friendly rate, but such enforcement entirely *removes* the performance benefits of receiver-driven protocols. (3) In the file- and streaming-server scenarios, it is possible to strike an acceptable balance between protocol performance on one hand, and vulnerability to misbehavers on the other, due to the fact that moderate bandwidth stealers do not represent a critical threat to the system security. (4) On the contrary, short time-scale receiver misbehaviors can extremely degrade the response times of well-behaving clients in the HTTP-server scenarios; hence, such servers have to *strictly* apply sender-based short-time-scale protection mechanisms; unfortunately, such mechanisms can often limit the receiver-driven TCP performance to a level which is *below* the level achievable by sender-based TCP.

References

[1] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's initial window, 1998. Internet RFC 2414.

[2] H. Balakrishnan and R. Katz. Explicit loss notification and wireless web performance. In *IEEE GLOBECOM*, 1998.

[3] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic behavior of slowly-responsive congestion control algorithms. In *ACM SIGCOMM*, 2001.

[4] C. Casetti, M. Gerla, S. Mascolo, M. Sanadidi, and R. Wang. TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In *ACM MOBICOM*, 2001.

[5] D. Clark, M. Lambert, and L. Zhang. NETBLT: A high throughput transport protocol. In *ACM SIGCOMM*, 1987.

[6] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust congestion signaling. In *IEEE ICNP*, 2001.

[7] A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *ACM SIGCOMM*, 1999.

[8] W. Feng, D. Kandlur, D. Saha, and K. Shin. Stochastic fair BLUE: A queue management algorithm for enforcing fairness. In *IEEE INFOCOM*, 2001.

[9] S. Floyd. Highspeed TCP for large congestion windows, 2003. Internet draft draft-ietf-tsvwg-highspeed-01.txt.

[10] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *ACM SIGCOMM*, 2000.

[11] S. Floyd, J. Madhavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgement (SACK) option for TCP, 2000. Internet RFC 2883.

[12] R. Gupta, M. Chen, S. McCanne, and J. Walrand. A receiver-driven transport protocol for the web. In *INFORMS*, 2000.

[13] M. Handley, J. Padhye, S. Floyd, and J. Widmer. TCP friendly rate control, 2001. IETF Internet draft.

[14] H.-Y. Hsieh, K.-H. Kim, Y. Zhu, and R. Sivakumar. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. In *ACM MOBICOM*, 2003.

[15] C. Jin, D. Wei, and S. Low. FAST TCP: Motivation, architecture, algorithms, performance. In *IEEE INFOCOM*, 2004.

[16] A. Kuzmanovic and E. Knightly. A performance vs. trust perspective in the design of end-point congestion control protocols (extended version). *Technical Report*, 2004.

[17] D. Lin and R. Morris. Dynamics of Random Early Detection. In *ACM SIGCOMM*, 1997.

[18] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. In *IEEE ICNP*, 2001.

[19] M. Mathis, J. Semke, J. Madhavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance. *ACM Computer Comm. Review*, 27(3):67–82, 1997.

[20] P. Mehra, A. Zakhor, and C. Vleeschouwer. Receiver-driven bandwidth sharing for TCP. In *IEEE INFOCOM*, 2003.

[21] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In *IEEE ICNP*, 2002.

[22] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Reno performance: A simple model and its empirical validation. *IEEE/ACM ToN*, 8(2):133–145, 2000.

[23] V. Padmanabhan and R. Katz. TCP Fast Start: A technique for speeding up web transfers. In *IEEE GLOBECOM*, 1998.

[24] R. Pain, B. Prabhakar, and K. Psounis. CHOKe, a stateless active queue management scheme for approximating fair bandwidth allocation. In *IEEE INFOCOM*, 2000.

[25] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading transport protocols with untrusted mobile code. In *ACM SOSP*, 2003.

[26] V. Paxson. Automated packet trace analysis of TCP implementations. In *ACM SIGCOMM*, 1997.

[27] V. Paxson and M. Allman. Computing TCP's retransmission timer, 2000. Internet RFC 2988.

[28] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *ACM Computer Comm. Review*, 29(5):71–78, 1999.

[29] P. Sinha, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. WTCP: A reliable transport protocol for wireless wide-area networks. In *ACM MOBICOM*, 1999.

[30] N. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, and B. Bershad. Receiver based management of low bandwidth access links. In *IEEE INFOCOM*, 2000.

[31] V. Tsaoussidis and C. Zhang. TCP-Real: Receiver-oriented congestion control. *The Journal of Computer Networks*, 40(4):477–497, 2002.

[32] M. Vojnovic and J.-Y. L. Boudec. On the long-run behavior of equation-based rate control. In *ACM SIGCOMM*, 2002.

[33] R. Wang, G. Pau, K. Yamada, M. Sanadidi, and M. Gerla. TCP start up performance in large bandwidth delay networks. In *IEEE INFOCOM*, 2004.

[34] Y. Zhang, L. Qiu, and S. Keshav. Speeding up short data transfers: Theory, architectural support, and simulations. In *NOSSDAV*, 2000.