# TCP congestion control algorithms and a performance comparison

Yuan-Cheng Lai and Chang-Li Yao
Department of Computer Science and Information Engineering
National Cheng Kung University
laiyc@locust.csie.ncku.edu.tw

***Abstract*-The evolution of TCP congestion control algorithms and underlying concepts of each algorithm are presented herein. According to their specific strategies, TCP algorithms are classified into four groups: Reno and NewReno, SACK and the variants, SSDR, and loss-avoidance algorithms. Numerous simulations are conducted to compare the performances of these TCP algorithms, and simulation results clearly indicate the merits and limitations of them. Comments are also presented to provide users and developers with a better understanding of these algorithms.**

## I. INTRODUCTION

The first version of TCP, standardized in RFC 793, defined the basic structure of TCP i.e. the window-based flow control scheme and a coarse-grain timeout timer. The second version, TCP Tahoe, added the congestion avoidance scheme and fast retransmission [1]. The third version, TCP Reno, extended the congestion control scheme by including fast recovery scheme [2]. Reno was standardized in RFC 2001 and generalized in RFC 2581.

Today TCP Reno has become the most popular version. However, several shortcomings exist in TCP Reno. First, the multiple-packet-loss problem [3] often causes a timeout and results in low utilization. New-Reno [4] and SACK [5] try to resolve this problem with two different approaches. New-Reno remains in fast recovery until all of the data outstanding has been acknowledge. In contrast, SACK modifies the receiver behavior to report the non-contiguous sets of data that has been received or queued. FACK [6] then proposed to improve the fast recovery scheme in SACK. One year later, Rate-Halving algorithm [7] was proposed in a technical note to enhance SACK, and smooth-start and dynamic recovery (SSDR) [8] was presented and claimed to have a performance similar to SACK and FACK but a simpler implementation.

Furthermore, Brakmo and Peterson proposed a new loss-avoidance algorithm, dubbed Vegas [9], which exhibits an admirable fairness and throughput. In 1999, Pseudo-Rate [10] was proposed to improve upon Vegas. Fig. 1 exhibits the evolution of TCP congestion control algorithms. Notably, the chronology of each algorithm is specified in the latest revised time.

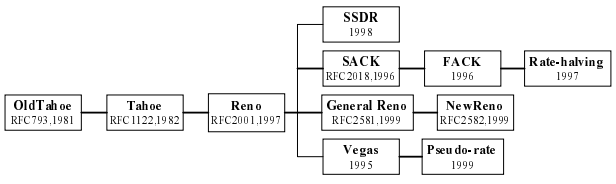


Fig. 1. Evolution of TCP congestion control algorithms.

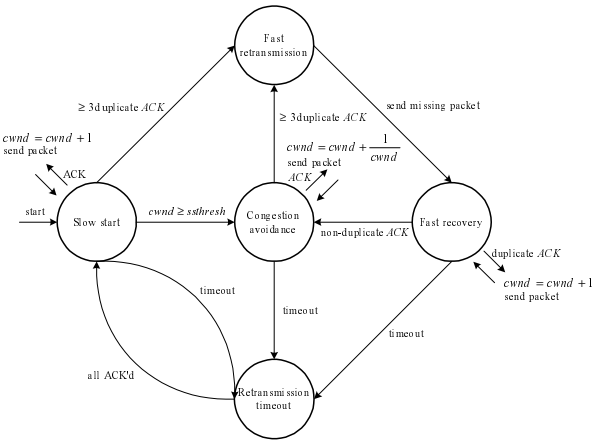


Fig. 2. Congestion control diagram of TCP Reno.

## II. TCP RENO

### 2.1 TCP Reno Congestion Controls

Reno uses a congestion window (*cwnd*) to control the amount of transmitted data in one round-trip time (*RTT*) and uses slow-start threshold (*ssthresh*) to separate its status into slow-start and congestion avoidance. When an *ACK* is received, *cwnd* is updated as follows:

$$cwnd = \begin{cases} cwnd + 1, & \text{if } cwnd < ssthresh \\ cwnd + 1/cwnd, & \text{if } cwnd \geq ssthresh \\ ssthresh, & \text{if packet loss } (ssthresh = cwnd/2) \\ 1, & \text{if timeout } (ssthresh = cwnd/2) \end{cases}$$

The control scheme of Reno can be divided into five parts: slow-start, congestion avoidance, fast retransmission, fast recovery and retransmission timeout. Fig. 2 schematically depicts the Reno specified with these parts.

### 2.2 TCP Reno Problems

Although Reno is the most popular TCP version to date, it has the following problems:

1) Multiple-packet-loss problem [3]: In Reno, the receiver always responds with the same duplicate *ACK*. When packet losses occur within one window, the sender understands at most one new loss per *RTT*, and that commonly results in a retransmission timeout. Fig. 3 illustrates the multiple-packet problem.

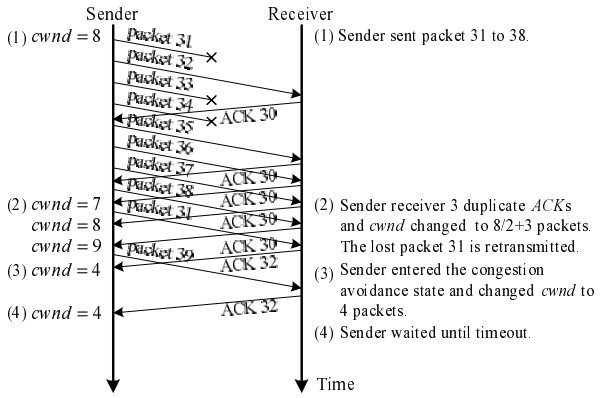


Fig. 3. Multiple-packet-loss problem.

2) Unfairness: Owing to its window control mechanism, TCP Reno has a bias against connections with long propagation delay. The connections with longer delay will obtain an obviously worse throughput. However, connections with distinct propagation delays should share the available bandwidth fairly.

## III. Enhanced TCP Congestion Control Algorithms

According to the relationship depicted in Fig. 1, these enhanced TCP congestion control algorithms are described within four subsections.

### *3.1 General Reno and NewReno*

**General Reno**: Following the essence described in RFC 2001, RFC 2581 defines the congestion control algorithms of Reno formally and generally. RFC 2581 provides two recommendations:

1) Slow-start should be employed to restart a transmission following a long idle period as a new connection is being created.

2) The receiver can use the delayed *ACK* mechanism, which generate an *ACK* when more than one packet is received.

**NewReno**: NewReno [4] modifies the fast recovery phase of Reno to alleviate the multiple-packet-loss problem. The recovery starts on detecting a packet loss and ends on when the receiver acknowledges the reception of all data transmitted before the start of fast retransmission. Thus, when multiple packets are lost within one window of data, NewReno may recover them without a retransmission timeout.

### *3.2 SACK, FACK and Rate-Halving*

**Selective ACK (SACK)**: The SACK [5] option for Reno has been introduced to further enhance TCP performance by allowing (selective) acknowledgment of packet held at the receiver. When receiving out-of-sequence packets, the receiver sends duplicate *ACK*s bearing the SACK option. The SACK option field contains a number of SACK blocks, where each reports a non-contiguous sets of data that has been received and queued. When the third duplicate *ACK* is received, a SACK TCP sender retransmits the packets starting with the sequence number acknowledged by the duplicate *ACK*s followed by subsequent unacknowledged packets.

**Forward ACK (FACK)**: FACK [6] uses the additional information provided by the SACK option to keep an explicit measure of the total number of data outstanding in the network. Since the sender may have a long wait for three duplicate *ACK*s, FACK enters fast retransmission earlier when the receiver reports that the reassembly queue is longer 3 packets.

**Rate-Halving**: Rate-Halving [7] attempts to find the correct window size following packet loss. Rate-Halving recommend a smooth reduction of *cwnd* during the first *RTT* of fast recovery, thus, the sender can continue sending a few new packets to maintain its self-clocking. In Rate-Halving, a *cwnd* adjustment during entire fast recovery is constrained to spacing transmissions at the rate of one data segment per two *ACK*s received.

### *3.3 Smooth-start and Dynamic Recovery (SSDR)*

**SSDR**: SSDR [8] was proposed to perform two Reno modifications and is claimed to be as efficient as SACK and FACK. First, Slow-start often causes multiple packet losses, which result in a retransmission timeout. Smooth-start uses a more graceful ramp of *cwnd* to relieve this problem. Second, the dynamic recovery adds a new damping phase that performs as Rate-Halving does in fast recovery. During a sender recovers from packet losses, dynamic recovery use a variable-length recovery period, while also sending new packets to probe the new equilibrium of the TCP connection.

### *3.4 Vegas and Pseudo-Rate*

**Vegas**: Vegas [9] uses the measured *RTT* to accurately calculate the amount of data packets that the sender can send to avoid packet losses. In Vegas, the sender must record the *RTT* and the sending time of each packet. The minimum round-trip time, *basertt*, must also be kept. Herein, a new congestion avoidance is proposed based on the measured *RTT*. When receiving an *ACK*, the sender calculates the difference of the expected and the actual throughputs as follows,

$$diff = (expected - actual)basertt = \left( \frac{cwnd}{basertt} - \frac{cwnd}{\text{average measured } RTT} \right) basertt$$

The idea is that the actual throughput will approach the expected throughput when the network is not congested. Vegas defines two thresholds ($\alpha$, $\beta$) as a tolerance that allows the source to control the difference between expected and actual throughputs in one *RTT*. *Cwnd* is increased by one packet if $diff<\alpha$ and decreased by one packet if $diff>\beta$. That is,

$$cwnd = \begin{cases} cwnd + 1, & \text{if } diff < \alpha \\ cwnd - 1, & \text{if } diff > \beta \\ cwnd, & \text{otherwise} \end{cases}$$

**Pseudo-Rate**: Pseudo-Rate [10] modifies Vegas in two manners.

1) New *RTT* estimation: Rather than recording the *RTT* for each packet sent in Vegas, Pseudo-Rate measures only the *RTT* for one packet.

2) Exponential increase/Direct-drop decrease: During each cycle, the window bound is calculated to constrain the maximum *cwnd* size (window bound = *basertt* × network service rate + queue length). If the window bound exceeds the

TABLE 1
SUMMARY OF TCP ALGORITHMS

| TCP algorithms | Loss-Recovery / Avoidance | Modification on Sender/Receiver | Features Enhanced |
|---|---|---|---|
| OldTahoe | Recovery | – | · Window-based flow control<br>· Timer Granularity |
| Tahoe | Recovery | – | · Slow-start<br>· Congestion avoidance<br>· Fast retransmission |
| Reno | Recovery | – | · Fast recovery |
| TCP Congestion Control | Recovery | No | · General congestion control scheme of Reno |
| NewReno | Recovery | Sender | · Response to partial *ACK*s |
| SACK | Recovery | Both | · SACK Options |
| FACK | Recovery | Both | · Better estimation of outstanding packet during fast recovery<br>· Earlier fast retransmission |
| Rate-Halving | Recovery | Both | · Smoothing window reduction during fast recovery<br>· Additional updating the TCP state variables after fast recovery |
| SSDR | Recovery | Sender | · Smooth-start<br>· Dynamic recovery |
| Vegas | Avoidance | Sender | · New congestion avoidance<br>· Modified slow-start mechanism<br>· Earlier packet loss detection |
| Pseudo-Rate | Avoidance | Sender | · New *RTT* estimation<br>· Direct-drop decrease<br>· Exponential increase |

old *cwnd*, Pseudo-Rate expands its *cwnd* to window bound exponentially. In constant, if the window bound is less than the old *cwnd*, Pseudo-Rate reduces *cwnd* to window bound directly. The exponential increase and direct decrease schemes produce a more sensitive sender within network variation.

Finally, a summary of these TCP algorithms is listed in Table 1.

## IV. SIMULATION RESULTS

The network simulator (*ns*) [11] is used as the tool to conduct our simulation. Fig. 4 shows the network model, the links are labeled with their bandwidth and propagation delay, and packet size is fixed at a length of 512 bytes.

### *4.1 Throughput*

In this subsection, mean throughput (*MT*) is investigated for each TCP algorithm, where is obtained by averaging the throughput of all connections of the same version. The experiments are conducted arranging various link error rates. The simulation results are shown in Fig. 5.

1) *The throughput rank order is Vegas > Pseudo-Rate ≈ Rate-halving > SSDR ≈ FACK ≈ SACK > NewReno > Reno*. NewReno alleviate the multiple-packet-loss problem, however, it requires numerous *RTT*s to recover lost packets. Thus, NewReno only performs slightly better than Reno does. Alternately, the SACK can process multiple packet losses within one *RTT,* thus outperform both Reno and NewReno. As FACK and Rate-Halving can more correctly estimate the outstanding packets in the network, their mean throughputs are better than that of SACK. Finally, Vegas and Pseudo-Rate use the measured *RTT* to estimate the available bandwidth. Hence their packet losses occur rarely. Notably, this results in the best performance among all algorithms.

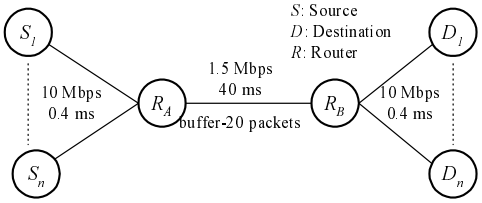


Fig. 4. Network model.

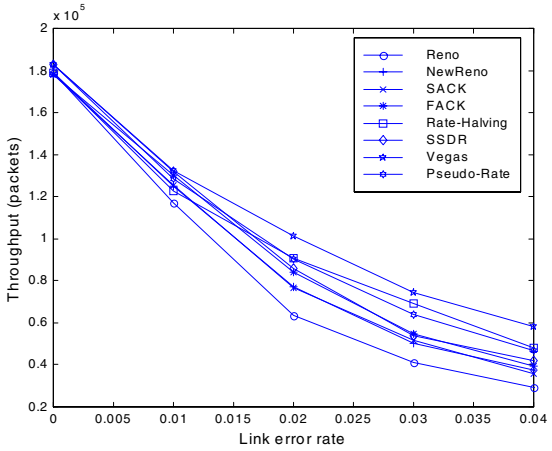


Fig. 5. *MT* for each TCP algorithm with various link error rates (1000secs).

2) *The throughput gaps among these algorithms enlarge as the link error rate increases.* As the link error rate increases, multiple packet losses occur more often, thus the individuality of each algorithm becomes more obvious.

### *4.2 Fairness*

The fairness of TCP algorithms was investigated within connections, which have distinct propagation delays. The connections are divided into short and long connections, which are 40 ms and 0.4ms respectively, between the source and router A, and between router B and the destination. Table 2 displays the simulation results, where the *UF* (Unfairness) is computed by:

$$\frac{|\,throughput\ of\ long\ connection - throughput\ of\ short\ connection\,|}{mean\ throughput\ of\ all\ connections} \times 100\%$$

1) *All TCP algorithms have a bias against long connections.* The quick response of the short connection causes its quick increase of window size either at the beginning or after congestion release, which results in a superior performance.

2) *The algorithms that use SACK option have better fairness than those that do not*. Because these fewer timeouts cause by the algorithms that employ the SACK option, result in lesser opportunity of the short connection to grab the bandwidth. Thus using the SACK option can get better fairness.

*3) Loss-avoidance schemes achieve better fairness than the loss-recovery algorithms do.*

*4) Pseudo-rate achieves the best fairness among all algorithms.*

TABLE 2
PERFORMANCE OF TCP ALGORITHMS WITH DISTINCT PROPAGATION DELAYS (50 SECS).

| Number of Connections | *n*=2 | | | *n*=4 | | |
|---|---|---|---|---|---|---|
| TCP | Short | Long | UF (%) | Short | Long | UF (%) |
| Reno | 15649 | 695 | 182.99 | 8139.0 | 330.5 | 184.39 |
| New Reno | 16335 | 289 | 193.04 | 8315.5 | 284.5 | 186.76 |
| SACK | 15271 | 1838 | 157.02 | 8245.5 | 472.0 | 178.34 |
| FACK | 14823 | 1995 | 152.55 | 7868.5 | 724.0 | 166.29 |
| Rate-Halving | 15484 | 1973 | 154.79 | 8336.5 | 621.5 | 172.24 |
| SSDR | 15640 | 824 | 179.98 | 8579.0 | 340.0 | 184.75 |
| Vegas | 14427 | 3212 | 127.16 | 5163.5 | 3828.5 | 29.69 |
| Pseudo-rate | 10229 | 7688 | 28.36 | 4990.5 | 4074.5 | 20.20 |

TABLE 3
*MT* OF TCP ALGORITHMS WHEN A NEW ARRIVAL AT 10 SECS (10~50 SECS).

| Number of Connections | *n*=1 | | *n*=2 | | *n*=3 | |
|---|---|---|---|---|---|---|
| | New | Old | New | Old | New | Old |
| Reno | 6322 | 7113.5 | 4196 | 4772.00 | 3174 | 3587.00 |
| New Reno | 6859 | 7152.5 | 4678 | 4780.66 | 3521 | 3571.75 |
| SACK | 6669 | 7110.0 | 4347 | 4739.66 | 3244 | 3543.00 |
| FACK | 6899 | 7140.0 | 4463 | 4762.00 | 3302 | 3569.75 |
| Rate-Halving | 6500 | 7200.5 | 4430 | 4818.66 | 3308 | 3627.25 |
| SSDR | 5137 | 7324.0 | 4414 | 4876.66 | 3732 | 3666.75 |
| Vegas | 5597 | 7324.0 | 5171 | 4882.33 | 3864 | 3661.25 |
| Pseudo-rate | 8309 | 7324.0 | 6317 | 4882.33 | 4816 | 3661.75 |

TABLE 4
*MT* OF TCP ALGORITHMS INTEROPERATING WITH RENO.

| Number of Connections | *n*=2 | | *n*=4 | |
|---|---|---|---|---|
| Algorithms | Reno | Other algorithm | Reno | Other algorithm |
| New Reno | 8663 | 8257 | 4181.0 | 4469.5 |
| SACK | 6517 | 11043 | 3969.0 | 4838.5 |
| FACK | 7088 | 10593 | 3757.5 | 5139.5 |
| Rate-Halving | 7665 | 10033 | 3605.5 | 5322.0 |
| SSDR | 5868 | 11721 | 3414.0 | 5592.5 |
| Vegas | 12468 | 5348 | 5696.5 | 2982.0 |
| Pseudo-rate | 12943 | 4745 | 5123.5 | 3889.0 |

### *4.3 Transient period*

Within the loss-recovery algorithm, the transient period is defined as the period that the old and new connections have the similar performance. However, within the loss-avoidance algorithm, it is defined as the period between the new arrival and the instant that the *cwnd* of all connections are stable. In our simulation, some connections begin initially and a new connection arrives after 10 seconds. Table 3 presents the experimental results, which are collected between 10 and 50 seconds.

1) *The transient period of the loss-recovery algorithms is longer than that of the loss-avoidance algorithms*. The transient period of loss-avoidance algorithm is very short due to the old connections actually decreasing its *cwnd* when the new connection arrives.

2) *For loss-recovery algorithms, the new connection obtains a bandwidth that is similar to the old ones*. As both new and old connections have the ability to contend for bandwidth, following the transient period the new connection has a throughput that is similar to that of the old connections.

3) *Within loss-avoidance algorithms, as the number of connections enlarges, the new connection usually has the better throughput*. When the number of connections is large, the queue length in the buffer also increases. Thus, the new connection overestimates the *basertt*, rather than correct minimum *RTT* (no queue delay). This overestimation provides the new connection with greater bandwidth.

4) *For Pseudo-rate, there is an obvious bias against old connections*. For Vegas, when one connection exists and a new connection arrives, the new connection receives less bandwidth than the old one does, while for Pseudo-rate, the opposite condition occurs. The reason is that there is a linear decrease of *cwnd* of the old connection in Vegas and direct drop of it in Pseudo-rate. Due to the better performance of the new connection in this case as well as comment 3) above, Pseudo-rate consistently has a bias against the old connections.

### *4.4 Interaction with Reno*

Table 4 shows that Reno robs the bandwidth of loss-avoidance algorithms.

1) *Reno robs the bandwidth of loss-avoidance algorithms*. The main reason is that Reno is aggressive in the sense that it increase the window size until packet lose, while Vegas is conservative in the sense that it try and keep to maintain a proper number of packets into the buffer at the router. Therefore, when Vegas and Reno coexist, the window size of the Reno will oscillates in the high level and Vegas almost always stabilize in the low level [12]. Similar phenomena also occur in the case where Pseudo-rate and Reno interoperate.

2) *The aggressive behaviors of loss-recovery algorithms gain more bandwidth than Reno does*. The loss-recovery algorithms including NewReno, SACK, FACK and Rate-Halving are all derived from Reno so that they have inherited its bandwidth contention ability. Thus, when interoperating with Reno, these algorithms have an advantage.

## V. CONCLUSION

In this investigation, recent TCP algorithms have been described. Based on our simulations, the advantages and drawbacks of these algorithms are exhibited clearly. Although NewReno promotes the throughput, it continues to spend much time to recover multiple packet losses. SACK and its variants increase throughput and fairness, however, they require receiver coordination. SSDR elevates the throughput, but retains an unfair problem. The loss-avoidance schemes achieve the best throughput and fairness, however, they fail to interoperate well with Reno.

## REFERENCES


[1] V. Jacobson, "Congestion Avoidance and Control," ACM SIGCOMM '88, pp. 273-288, 1988.

[2] V. Jacobson, "Modified TCP Congestion Avoidance Algorithm," mailing list, end2end-interest, 30 Apr. 1990.

[3] S. Floyd, "TCP and Successive Fast Retransmits," ftp://ftp.ee.lbl.gov/papers/fastretrans.ps, May. 1995.

[4] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC2582, Apr. 1999.

[5] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC2018, Oct. 1996.

[6] M. Mathis and J. Mahdavi, "Forward Acknowledgment: Refining TCP Congestion Control," ACM SIGCOMM '96, pp. 281-291, Aug. 1996.

[7] M. Mathis, J. Semke, J. Mahdavi and K. Lahey, "The Rate-Halving Algorithm for TCP Congestion Control," http://www.psc.edu/networking/papers/draft-ratehalving.txt, Jun. 1999.

[8] H. Wang and C. Williamson, "A New Scheme for TCP Congestion Control: Smooth-Start and Dynamic Recovery," Sixth International Symposium 1998, pp. 69-76.

[9] L. S. Brakmo and L. L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet," IEEE JSAC, pp. 1465-1480, 1995.

[10] J. R. Chen, Y. C. Chen, "Pseudo-Rate TCP: a congestion avoidance scheme with nearly optimized fairness and throughput," Computer Communications, pp. 1493-1501, 1999.

[11] K. Fall and S. Floyd, ns–Network Simulator, http://www-mesh.cs.berkeley.edu/ns.

[12] Y. C. Lai, "Performance Improvement of TCP Vegas in a Heterogeneous Environment," submitted to Computer Communications.