An Implementation of Scalable, Vegas, Veno, and YeAH Congestion Control Algorithms in ns-3

Truc Anh N. Nguyen[*], Siddharth Gangadhar[*], Md Moshfequr Rahman[*], and James P.G. Sterbenz[*†§]
[*]Information and Telecommunication Technology Center
Department of Electrical Engineering and Computer Science
The University of Kansas, Lawrence, KS 66045, USA
[‡]School of Computing and Communications (SCC) and InfoLab21
Lancaster University, LA1 4WA, UK
[§]Department of Computing
The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong
{annguyen, siddharth, moshfequr, jpgs}@ittc.ku.edu

ABSTRACT

Despite the modern advancements in networking, TCP congestion control is still one of the key mechanisms that ensure the stability of the Internet. Given its principal role, it is a popular research topic revisited every time TCP and its variants are studied. Open-source network simulators such as ns-3 are important tools used by the research community to gain valuable insight into existing TCP congestion control algorithms and to develop new variants. However, the current TCP infrastructure in ns-3 supports only a few congestion control algorithms. As part of the ongoing effort to extend TCP functionalities in the simulator, we implement Scalable, Vegas, Veno, and YeAH based on the original literature and their implementations in the Linux kernel; this paper presents our implementation details. The paper also discusses our validation of the added models against the theories to demonstrate their correctness. Through our evaluation, we highlight the key features of each algorithm that we study.

CCS Concepts

•Networks → Transport protocols; Network simulations;

Keywords

TCP Vegas, NewReno, Veno, YeAH, Scalable, transport protocols, ns-3 network simulator, performance evaluation, congestion control, loss-based, delay-based, hybrid, Future Internet

1. INTRODUCTION

TCP has been proven to be a crucial component of the Internet due to its ability to sustain its performance (although not optimal) with the evolution of networking during the past decades. One of TCP elements that is principal in ensuring the Internet stability and widely studied with numerous research is its congestion control algorithm. TCP congestion control is revisited on almost every attempt to study the Internet transport layer. With the modern advancements in networking, comes the emergence of new network environments such as Gigabit Ethernet or satellite links with challenging characteristics: high bit-error rate, long propagation delay, high link capacity, and asymmetric channels. Standard TCP has been enhanced for the Future Internet, resulting in many variants such as those studied in our paper (Scalable, Vegas, Veno, and YeAH). The current effort to employ load balancing at the transport layer by incorporating multi-path feature into TCP producing MPTCP [8] also requires a thorough understanding of the standard congestion control. This enables design of new algorithms that can operate efficiently in a more complex system, one with multiple coexistent, but heterogeneous subflows simultaneously transferring data through a bottleneck with high congestion probability.

The study of existing TCP algorithms and the development of any new enhancements gain substantial benefits through the use of open-source network simulators such as ns-3 [2]. However, the current ns-3 standard release only consists of NewReno (default), Westwood, Westwood+, Hybla, and HighSpeed congestion control algorithms. In order to extend the supported ns-3 TCP functionalities, we implement additional protocols, including Scalable, Vegas, Veno, and YeAH. This paper presents our implementation details of the added models. The paper also discusses our validation of these contributions against the original papers to demonstrate the correctness of the models. Through our evaluation, we highlight the key features of each algorithm.

The remainder of the paper is organized as follows: Section 2 briefly provides the theoretical background of the congestion control algorithms studied in our paper followed by a short survey on related work. Section 3 explains the implementations and how the new models interact with the rest of the TCP framework in ns-3. In Section 4, the correctness of our implementations is verified. Finally, Section 5 concludes our paper with directions for future work.

2. BACKGROUND AND RELATED WORK

This section provides the theoretical background of different congestion control algorithms studied in our paper, including the standard NewReno, Scalable, Vegas, Veno, and YeAH, followed by a brief survey of related work.

2.1 TCP Congestion Control Algorithms

TCP congestion control algorithms can be classified into four categories: loss based, delay based, hybrid, and explicit notification [5]. Loss-based algorithms treat the occurrence of a packet loss as an indication of congestion. Delay-based algorithms infer congestion based on the increasing delay due to queue build-up when traffic load exceeds network capacity. Hybrid algorithms take advantage of both loss- and delay-based mechanisms, while explicit congestion notification (ECN) relies on explicit signals from network elements to learn about congestion. Our paper covers the first three groups, in that Scalable is a loss-based algorithm, Vegas is delay-based, and Veno and YeAH are hybrid. The loss-based NewReno (default TCP in ns-3) is used in our paper as the baseline for comparison.

Most TCP congestion control variants are derivatives of the standard defined in RFC 5681 [4], which is known as the Reno algorithm introduced by Jacobson [13] as a revision of his original Tahoe [12]. The standard specifies four intertwined algorithms that together play a principal role in the stabilization of the Internet and the prevention of congestion collapse: slow start, congestion avoidance, fast retransmit, and fast recovery. The implementation of these algorithms requires the definition and maintenance of three state variables: cwnd, rwnd, and ssthresh. Congestion window (cwnd) determines the amount of data a sender can transmit before it receives an ACK to prevent network overflow. The receiver window (rwnd) indicates the amount of data a receiver is willing to accept. The actual sending window is the minimum of cwnd and rwnd. Slow start threshold (ssthresh) provides the transition point between slow start and congestion avoidance phases.

The slow start allows TCP to gradually probe for network bandwidth when TCP starts its data transmission or after an expiration of its retransmission timer. The slow start algorithm prevents TCP from suddenly throttling the network with a large burst of traffic. In addition, slow start initiates TCP ACK clocking that determines when new data should be placed into the network to sustain equilibrium during a connection lifetime. During slow start, cwnd is incremented by 1 for every new ACK received, resulting in an exponential increase of the sending rate until a loss happens as shown in Equation 1.

$$\text{cwnd} = \text{cwnd} + 1 \tag{1}$$

The congestion avoidance algorithm continues to allow TCP to increase its sending rate (cwnd), but at a slower speed than when it is in the slow start phase to prevent congestion after the sending rate reaches ssthresh. Specifically, cwnd is incremented by 1 for every RTT, resulting in a linear increase over time until the experience of a loss. This is equivalent to the cwnd modification per Equation 2 upon a new ACK receipt.

$$\text{cwnd} = \text{cwnd} + \frac{1}{\text{cwnd}} \tag{2}$$

The fast retransmit algorithm is responsible for promptly detecting and recovering lost data by observing the number of received duplicate ACKs (dupACKs), with the arrival of three dupACKs signifying the loss of a segment. Fast retransmit was developed as an alternative to the original retransmission timer in detecting packet losses. The fast recovery governs data transmission after fast retransmit until a new ACK arrives informing the recovery of the loss. The occurrence of a loss requires Reno to halve its slow-start threshold and sending rate according to Equations 3 and 4, respectively.

$$\text{ssthresh} = \frac{\text{cwnd}}{2} \tag{3}$$

$$\text{cwnd} = \text{ssthresh} + 3 \tag{4}$$

2.1.1 NewReno

NewReno [11] modifies the Reno fast recovery algorithm explained above by introducing a mechanism for responding to *partial acknowledgments* to enhance TCP's ability by recovering more efficiently from multiple losses occurring in a single sending window. NewReno defines an additional state variable named recover to keep track of the highest sequence number transmitted before the sender enters fast retransmit, and it only leaves its fast recovery state upon the receipt of a full ACK, which is an ACK that acknowledges all sent data up to and including recover. In case a partial ACK arrives with acknowledgment number less than recover, the algorithm remains in fast recovery trying to retransmit the next in-sequence packet while sending a new segment if cwnd and rwnd allow. The NewReno algorithm is an alternate solution for multiple data loss recovery in the absence of TCP selective acknowledgment (SACK) [16].

2.1.2 Scalable

Scalable (STCP) [14] improves TCP performance for bulk transfers in high-speed wide-area networks that are characterized by long delay and high link bandwidth, by altering TCP congestion window update algorithm. The goal is to shorten TCP recovery time following a transient congestion by using a different additive increase and multiplicative decrease factors from those employed in Reno. While operating in congestion avoidance phase, STCP increments its cwnd by 0.01 for every new ACK received until a loss occurs as shown in Equation 5. On its detection of a congestion, the ssthresh value is reduced by a factor of 0.125 as in Equation 6 instead of 0.5 as in Reno (Eq. 3).

$$\text{cwnd} = \text{cwnd} + 0.01 \tag{5}$$

$$\text{ssthresh} = \text{cwnd} - \lceil 0.125 \times \text{cwnd} \rceil \tag{6}$$

2.1.3 Vegas

Vegas [6] implements a proactive congestion control algorithm that tries to prevent packet drops by keeping the backlog at the bottleneck queue small. During a connection lifetime, Vegas continuously samples the actual throughput rate and measures the RTT since these metrics reflect the network condition when it approaches congestion. The actual sending rate is computed using Equation 7. The difference diff (Equation 9) between this throughput value and the expected throughput calculated in Equation 8 reflects the number of packets enqueued at the bottleneck, i.e. the amount of extra data sent because the Vegas sender has been transmitting at a rate higher than the available network bandwidth. Vegas tries to keep this diff value between two predefined thresholds, α and β by linearly increasing and decreasing cwnd during its congestion avoidance phase. The diff value and another predefined threshold γ are used to determine the transition between slow-start and linear increase/decrease mode.

$$\text{actual} = \frac{\text{cwnd}}{\text{RTT}} \tag{7}$$

$$\text{expected} = \frac{\text{cwnd}}{\text{BaseRTT}} \tag{8}$$

$$\text{diff} = \text{expected} - \text{actual} \tag{9}$$

In Equation 8, BaseRTT represents the minimum RTT observed during a connection lifetime.

2.1.4 *Veno*

TCP Veno [9] enhances Reno algorithm to deal with random loss in wireless access networks by employing the Vegas algorithm for estimating the current network condition to identify the actual cause of a loss. Specifically, Veno does not use the estimated number of packets enqueued (backlog N calculated in Equation 10) at the bottleneck to proactively detect congestion, but to distinguish between a corruption-based loss and a congestion-based loss. When Veno learns that a loss is non-congestive, instead of halving ssthresh (Eq. 3), it reduces ssthresh by a smaller amount using Equation 11. Veno also refines the Reno congestion avoidance algorithm by increasing the sending rate by 1 every 2 RTTs if the backlog exceeds its predefined threshold β, allowing it to operate longer in the stable state during which network capacity is fully utilized.

$$N = \text{actual} \times (\text{RTT} - \text{BaseRTT}) = \text{diff} \times \text{BaseRTT} \tag{10}$$

$$\text{ssthresh} = \text{cwnd} \times \frac{4}{5} \tag{11}$$

2.1.5 *YeAH*

Yet Another Highspeed (YeAH) [5] is a heuristic aimed to fully exploit the capacity of high bandwidth-×-delay product (BDP) networks with a small number of induced congestion events, while trying to balance among various constraints such as fair competition with standard Reno flows and robustness to random losses. YeAH-TCP operates between its *Fast* and *Slow* modes. While in the Fast mode when the network link is not yet fully utilized, YeAH increases its cwnd according to STCP rule (Eq. 5). After full link utilization is achieved, it switches to Slow mode and behaves as Reno with a precautionary decongestion algorithm. The transition between the two modes is determined based on the backlog at the bottleneck queue calculated in Equation 12 and the estimated level of network congestion shown in Equation 13. Note that Equation 12 is basically the same as Equation 10.

$$Q = \text{RTT}_{\text{queue}} \times G = (\text{RTT}_{\text{min}} - \text{RTT}_{\text{base}}) \times \frac{\text{cwnd}}{\text{RTT}_{\text{min}}} \tag{12}$$

$$L = \frac{\text{RTT}_{\text{queue}}}{\text{RTT}_{\text{base}}} \tag{13}$$

To fairly compete with Reno flows, YeAH ensures that it only executes its decongestion algorithm if its current cwnd is greater than the cwnd of the competing Reno flows that it estimates, denoted by $\text{count}_{\text{reno}}$ in the algorithm.

Upon the receipt of three dupACKs, YeAH adjusts its ssthresh based on the current value of Q as in Equation 14 if it is not competing with Reno flows. Otherwise, ssthresh is halved as in Reno.

$$\text{ssthresh} = \min\{\max\{\frac{\text{cwnd}}{8}, Q\}, \frac{\text{cwnd}}{2}\} \tag{14}$$

2.2 Related Work

There are three ns-3 research works that are most relevant to our paper. The first implements TCP Westwood and Westwood+ protocols [10] and compares their performance against existing variants, including Tahoe, Reno, and NewReno under some selected network conditions. The second presents an implementation of TCP CUBIC [15], which is the default congestion control algorithm in the Linux kernel. The authors validate their implementation by comparing their model with the one in Linux using Network Simulator Cradle (NSC) and the corresponding implementation in ns-2 [1]. In the most recent work, Window Scaling and Timestamp Options together with other congestion control algorithms including Hybla, Highspeed, BIC, CUBIC, and Noordwijk are introduced into ns-3 TCP infrastructure [7].

3. IMPLEMENTATIONS

In this section, we first explain the TCP congestion control classes and their main operations in the new ns-3 TCP framework. We follow this with the implementation details of STCP, Vegas, Veno, and YeAH algorithms.

3.1 TCP Congestion Control Classes in ns-3

TCP implementation in ns-3 resides in the Internet module and consists of multiple classes interacting with each other to perform the supported TCP functionalities. The current standard release contains multiple TCP variants including NewReno as the default congestion control algorithm, Hybla, Highspeed, Westwood, and Westwood+. They are pluggable components implemented as child classes of `TcpNewReno`, which is in turn derived from the congestion control abstract class `TcpCongestionOps`. The main methods currently utilized in the base classes are described in ns-3 documentation [3] and summarized below. An extended version of this paper with class diagram is available on our ResiliNets wiki [17] [1].

- `TcpCongestionOps::GetSsThresh()` and `TcpNewReno::GetSsThresh()`: These methods compute ssthresh after a loss event.
- `TcpCongestionOps::IncreaseWindow()` and `TcpNewReno::IncreaseWindow()`: These methods determine the current congestion phase by comparing cwnd and ssthresh and call the corresponding functions.
- `TcpNewReno::SlowStart()`: This method adjusts cwnd during slow-start phase.

[1]http://www.ittc.ku.edu/resilinets/reports/Nguyen-Gangadhar-Rahman-Sterbenz-2016-extended.pdf

- `TcpNewReno::CongestionAvoidance()`: This method modifies cwnd during congestion avoidance phase.
- `TcpCongestionOps::PktsAcked()`: This method manipulates timing information carried by received ACKs.

3.2 STCP

`TcpScalable` is a derived class of `TcpNewReno` that inherits `TcpNewReno::IncreaseWindow()` and `TcpNewReno::SlowStart()`. Because STCP modifies NewReno additive increase and multiplicative decrease factors used in the congestion avoidance and fast retransmit modes, respectively, `TcpScalable` replaces `TcpNewReno::CongestionAvoidance()` and `TcpNewReno::GetSsThresh()`. The implementation of these methods requires three class members to be declared: m_aiFactor that represents the increase factor with a default value of 50, m_mdFactor that represents the decrease factor with a default value of 0.125, and m_ackCnt that keeps track of the number of segments acknowledged. Following the Linux implementation of STCP, we use 50 for m_aiFactor instead of 100 suggested in the literature to account for delayed ACKs. Listing 1 shows a code snippet of `TcpScalable::CongestionAvoidance()`.

```
uint32_t w = std::min (cwndInSegments,
    m_aiFactor);
if (m_ackCnt >= w)
  {
    cwndInSegments += m_ackCnt / w; }
cwnd = cwndInSegments * m_segmentSize;
```

Listing 1: TcpScalable::CongestionAvoidance().

`TcpScalable::GetSsThresh()` updates ssthresh after the receipt of three dupACKs following STCP rule as shown in Listing 2.

```
uint32_t ssThresh = std::max (2.0,
    cwndInSegments * (1.0 - m_mdFactor)) *
    m_segmentSize;
```

Listing 2: TcpScalable::GetSsThresh().

3.3 Vegas

The key components of our Vegas implementation in `TcpVegas` are its RTT sampling performed in `TcpVegas::PktsAcked()` method upon the receipt of an ACK, its calculation of the diff rate based on the RTT measurements in `PktsAcked()`, and the implementation of its linear increase/decrease mode. The latter is performed inside `TcpVegas::IncreaseWindow()` method, in which a code snippet is presented in Listing 3.

```
if (diff > beta)
  {
    /* We are going too fast, we need to
    slow down by linearly decreasing cwnd
    for the next RTT */
  cwnd = cwnd - m_segmentSize;   }
else if (diff < alpha)
  {
    /* We are going too slow, we need to
    speed up by linearly increasing cwnd
    for the next RTT */
    cwnd = cwnd + m_segmentSize; }
else
  {
    /* We are going at the right speed,
     * cwnd should not be changed */ }
```

Listing 3: TcpVegas::IncreaseWindow().

3.4 Veno

Similar to Vegas, `TcpVeno` requires an implementation of the `PktsAcked()` method to perform RTT sampling needed for the calculation of backlog N at the bottleneck queue. `TcpVeno::IncreaseWindow()` modifies cwnd following Veno additive increase rule as shown in Listing 4, where m_inc is a boolean variable that is only set to True every other RTT.

```
if (N < beta)
  {
    /* Available bandwidth is not fully
    utilized, we increase cwnd by 1 every
   RTT as in NewReno */
     TcpNewReno::CongestionAvoidance (tcb,
    segmentsAcked); }
else
  {
    /* Available bandwidth is fully
    utilized, we increase cwnd by 1 every
    other RTT */
      if (m_inc)
        {
          TcpNewReno::CongestionAvoidance (
    tcb, segmentsAcked); }
      else
        {
          m_inc = true; } }
```

Listing 4: TcpVeno::IncreaseWindow().

`TcpVeno::GetSsThresh()` implements the Veno multiplicative decrease algorithm as shown in Listing 5.

```
if (N < beta)
  {
    /* Random loss is most likely to have
    occurred, we reduce cwnd by only 1/5 */
    return std::max (cwnd * 4 / 5, 2 *
    m_segmentSize); }
else
  {
    /* Congestion-based loss is most likely
     to have occurred, we reduce cwnd by
    1/2 as in NewReno */
     return std::max (cwnd / 2, 2 *
    m_segmentSize); }
```

Listing 5: TcpVeno::GetSsThresh().

3.5 YeAH-TCP

`TcpYeah` also implements the `PktsAcked()` method to measure the RTT values required for its calculation of Q, which is used by `TcpYeah::IncreaseWindow()` to determine YeAH's operation mode (Fast or Slow) during its congestion avoidance phase. A code snippet of `TcpYeah::IncreaseWindow()` is presented in Listing 6. Following the Linux kernel implementation of YeAH, we use 80 and 8 as the default values of the two thresholds maxQ and phy, respectively.

```
if (Q < maxQ & L < (1 / phy))
  {
    // We are in Fast mode; cwnd is
    incremented based on STCP rule
    TcpScalable::CongestionAvoidance (tcb,
    segmentsAcked); }
else
  {
    /* We are in Slow mode, determine if we
     need to execute the precautionary
    decongestion algorithm */
    if (Q > maxQ && cwndInSegments >
    renoCount)
      {// Precautionary decongestion
        cwndInSegments -= Q;
        cwnd = cwndInSegments *
    m_segmentSize; } }
```

Listing 6: TcpYeah::IncreaseWindow().

In `TcpYeah::GetSsThresh()`, the reduction of `cwnd` depends on whether YeAH competes with Reno flows as shown in Listing 7, where the threshold `rho` is the minimum number of RTTs required to consider the presence of Reno flows.

```
if (doingRenoNow < rho)
  {// YeAH does not compete with Reno flows
    return std::min (std::max (cwnd / 8,Q),
    cwnd / 2); }
else
  {// YeAH competes with Reno flows
    return std::max (cwnd / 2, 2 *
    m_segmentSize); }
```

Listing 7: Yeah::GetSsThresh().

4. VERIFICATION AND VALIDATION

In addition to writing various unit tests that are mandatory in ns-3 to ensure the correctness of our new models before they can be merged into the standard release, we also try to simulate them under various network conditions and validate their performance against the corresponding literature. Given that the testing scenarios in the original papers are varied, the exact parameters used are not explicitly described in some sources, and to be consistent in our paper, we use the dumbbell topology illustrated in Figure 1 to fulfill our validation purpose. The goal of this section is to demonstrate that despite the simulation topology that we use, our models still exhibit the key characteristics of the protocols. Our simulations are conducted with ns-3.24-dev.

4.1 Simulation Topology

At each edge of the dumbbell topology in Figure 1 are two nodes serving as the sources on one end and the sinks at the other end. The endpoints communicate through a single bottleneck link that connects two network routers. All traffic across the network are generated using `BulkSendApplication` with an MTU size of 1500 bytes. Drop Tail queues are used at the bottleneck link with size set to the bandwidth-×-delay product. Each of the access link that connects an endpoint with one of the two routers has a bandwidth of 10 Mb/s with a negligible delay of 0.1 ms. The bandwidth and delay of the bottleneck link are varied depending on the simulated scenarios. The one-way delay value of this link ranges from 50 ms to 300 ms to cover the different delays for various network environments. Given that the protocols studied in this paper focus on improving the standard NewReno linear increase and multiplicative decrease phases, we set both the initial congestion window and slow start threshold to 15 packets to eliminate the slow start phase. Timestamps and window scaling options are both enabled. The duration of each simulation is 200 to 400 seconds. We use NewReno as the baseline for all of our comparisons. Simulation parameters are summarized in Table 1.

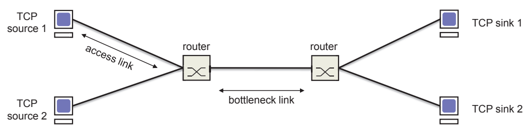


Figure 1: Simulation topology.

Table 1: Simulation parameters.

Parameter	Values
Access link bandwidth	10 Mb/s
Bottleneck link bandwidth	varied
Access link delay	0.1 ms
Bottleneck link delay	varied
Packet MTU size	1500 B
Delayed ACK count	2 segments
Application type	Bulk send application
Queue type	Drop tail
Queue size	BDP
Simulation time	200 s – 400 s

4.2 Robustness to Random Loss

To study the impact of random packet losses on the congestion control algorithms, we set the bottleneck bandwidth to 10 Mb/s and delay to 100 ms. Using `ns3::RateErrorModel`, we introduce a packet error rate (PER) of 10^{-3} into the unreliable bottleneck link. For a clearer presentation of the plots, each simulation is run for 200 seconds. We show the congestion window dynamics of a single connection with one sender and receiver on each network edge.

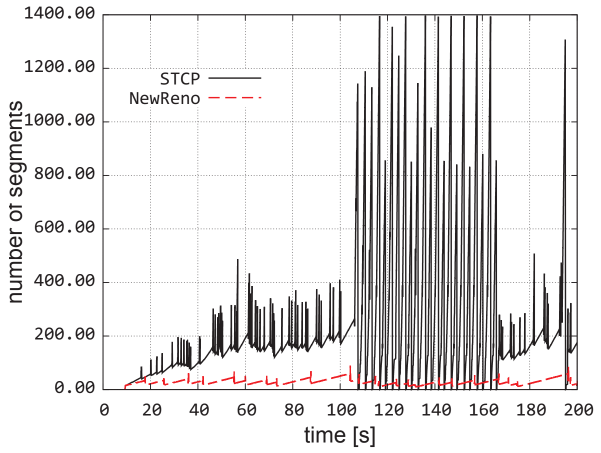


Figure 2: cwnd dynamics of STCP and NewReno.

Figure 2 shows the `cwnd` of STCP in comparison with NewReno's as they evolve over time. Overall, STCP's `cwnd` values are much higher than NewReno's, resulting in about

7 times more average throughput than the standard algorithm. As seen in the plot, NewReno takes $C/2$ seconds to recover where C is the cwnd that NewReno reaches following a loss, while the packet loss recovery time for STCP is independent of the connection's window size. In addition, NewReno halves its cwnd upon the receipt of three dupACKs. STCP, on the other hand, only reduces its window by $b \times C$ with b equal to 0.125. However, similar to other loss-based congestion control algorithms that were designed for high BDP network environments, the high throughput achieved by STCP comes at a price of higher chance of experiencing multiple retransmission timer timeouts (RTO) due to its aggressive increasing rule, as shown in the plot at time between 110 seconds and 160 seconds.

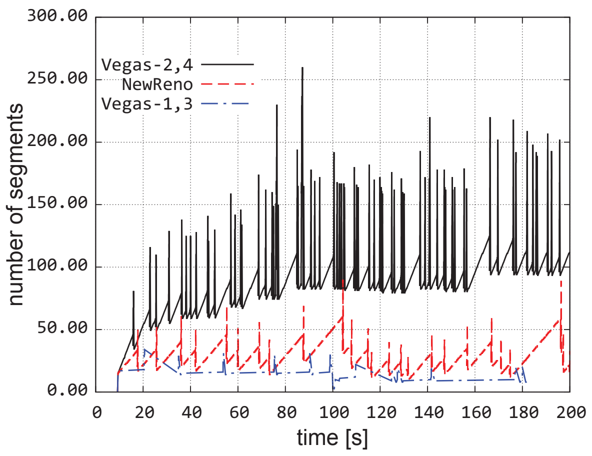


Figure 3: cwnd dynamics of Vegas and NewReno.

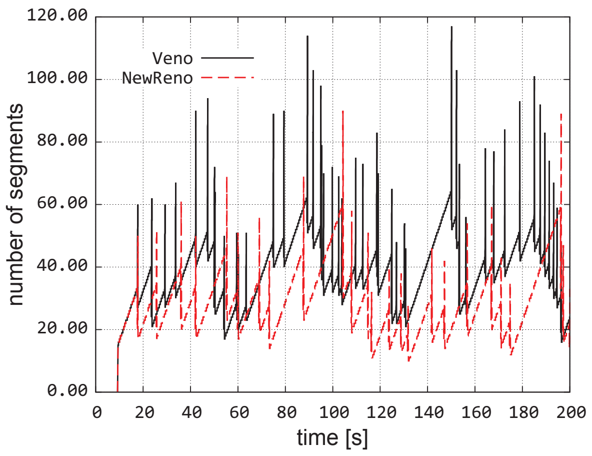


Figure 4: cwnd dynamics of Veno and NewReno.

Figure 3 shows the cwnd evolution of two versions of Vegas against NewReno. Following the notation used in the original Vegas paper, Vegas-1,3 sets α to 1 and β to 3, while Vegas-2,4 uses 2 and 4 for α and β, respectively. Overall, with our setup for this simulation, Vegas-2,4 outperforms Vegas-1,3, which even has a lower sending rate than the standard NewReno. Vegas-2,4 is able to achieve a throughput of 4.6 Mb/s, which is 3 times higher than the 1.5 Mb/s throughput achieved by NewReno. On the other hand, Vegas-1,3 is unable to utilize the available network capacity, resulting in a throughput of only 0.7 Mb/s. When we use 1 and 3 for α and β as the default values, most of the time, the calculated diff rate falls in between the two thresholds, causing Vegas to remain sending at the same rate without modifying its cwnd. For Vegas-2,4, the cwnd is increased by 1 segment size every RTT when the diff value is less than α, resulting in a linear increase until three dupACKs are received due to the random packet loss we introduce into the channel, which causes Vegas to reduce its cwnd by the maximum of the current ssthresh and 1/4 of congestion window. The linear increase/decrease mode then governs the sending rate after the reduction of cwnd. In our other simulations, we use Vegas-2,4 for a better throughput of Vegas, and this is also the default Vegas version in the Linux kernel.

Figure 4 shows the cwnd dynamics of Veno and NewReno. In this case, when the protocols are unable to fully utilize the available network bandwidth due to packet corruptions, the Veno increase rule is the same as NewReno. The only difference is its decreasing algorithm when a loss is detected. Since Veno is able to distinguish between congestive and non-congestive losses, for most of the time, Veno only reduces its cwnd by 1/5, resulting in a better throughput than NewReno.

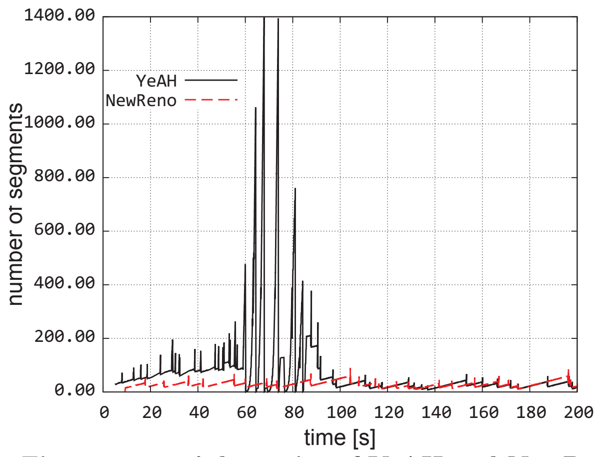
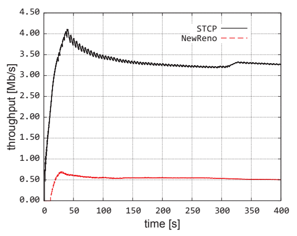


Figure 5: cwnd dynamics of YeAH and NewReno.

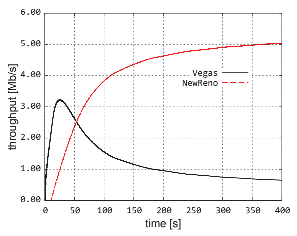
Figure 5 shows the cwnd dynamics of YeAH and NewReno. The congestion avoidance phase of YeAH is the switching between its Fast and Slow mode. With our simulation parameters and the default value of maxQ set to 80 segments, YeAH does not execute its precautionary decongestion algorithm because Q is less than maxQ although it detects that it does not compete with "greedy" Reno flows. Thus, the cwnd is only updated when YeAH enters its Fast mode. The increment rule during Fast mode follows STCP, but at a slower speed than the result for STCP presented in Figure 2 because we set m_aiFactor to 100 for YeAH implementation. Upon the detection of a loss through the receipt of three dupACKs before the occurrence of several RTOs, YeAH reduces its window by 1/8. The RTOs trigger a false alarm of the presence of Reno flows, which causes YeAH to halve its window afterward. All of these factors result in a low throughput of YeAH when comparing with STCP.

4.3 Friendliness to NewReno

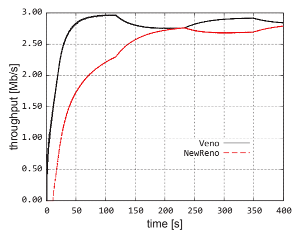
For a new TCP congestion control algorithm to be widely deployed in practice, it must be friendly with the standard Reno traffic. When it shares the network capacity with Reno



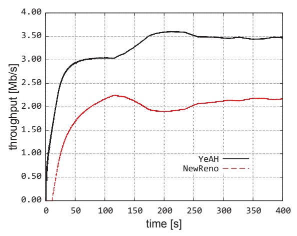
(a) STCP



(b) Vegas

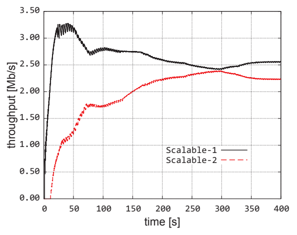


(c) Veno

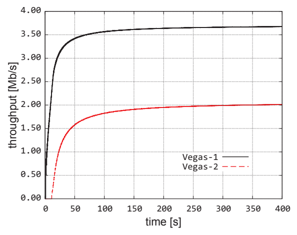


(d) YeAH

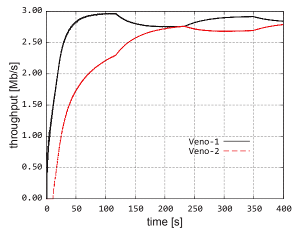
Figure 6: Instantaneous throughput with NewReno traffic.



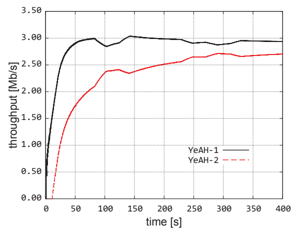
(a) STCP



(b) Vegas



(c) Veno



(d) YeAH

Figure 7: Instantaneous throughput with second flow using same algorithm.

flows, it should avoid starving the competing flows while being able to exploit the link bandwidth. So, in our second scenario, we study the friendliness of STCP, Vegas, Veno, and YeAH. We use the same dumbbell topology, but with two senders and two receivers at each network edge. Each 400-second simulation generates two traffic flows; one of them is NewReno while the other is one of the protocols studied in the paper. The bottleneck link has a bandwidth of 6 Mb/s and a delay of 100 ms with no random losses. The NewReno flow starts 10 seconds later than the other one.

Figure 6 shows the instantaneous throughput of each variant against NewReno's. Overall, Veno is the most TCP-friendly among all algorithms. The aggressive STCP puts the NewReno flow in starvation with the ratio of STCP throughput to NewReno throughput being 3.26:0.5 Mb/s. Basically, STCP is a NewReno derivative with higher increase, but smaller decrease factor than the standard algorithm's. Vegas exhibits its well-known behavior of a pure delay based algorithm as being the least aggressive among all protocols studied in our paper. While NewReno continues to increase its cwnd until a packet loss occurs, Vegas executes its proactive window adjustment that tries to send data at a moderate rate to prevent any packet drops at the bottleneck queue. As soon as the Reno flow enters the network, Vegas throughput starts to reduce. NewReno quickly obtains the same throughput as Vegas about 40 seconds after it starts and continues to steal the network bandwidth away from Vegas. Unlike Vegas, Veno does not use the extra number of packets at the bottleneck queue to control its sending rate. When the available bandwidth is not fully utilized and no random loss presents, Veno behaves exactly as NewReno during its additive increase and multiplicative decrease phase, resulting in a fair share of network capacity at 200 seconds. Because YeAH does not execute its precautionary decongestion control algorithm due to the presence of NewReno flow, it behaves like STCP during congestion avoidance. The ratio of YeAH to NewReno throughput is 3.47:2.17, which is smaller than the ratio of STCP to NewReno due to the higher m_aiFactor used in YeAH implementation as explained previously.

4.4 Intra Fairness

In addition to TCP friendliness, a congestion control algorithm is required to be internally fair: it should be friendly to itself. We study intra fairness of the protocols by simulating the same scenario as in Section 4.3, except that the second flow uses the same TCP variant as the first flow.

Figure 7 plots the instantaneous throughput of each variant in the competition with a second flow. While STCP, Veno, and YeAH try to converge to a fair share of the network resource after some time, Vegas maintains a constant gap between the throughput values of its two flows throughout the whole simulation period. This is because both Vegas flows have the tendency of attempting to prevent any queue drops. The first flow has the advantage of entering the network 10 seconds before the other, so it can obtain more bandwidth. By the time the second flow starts, it just attempts to utilize the remaining capacity.

4.5 Impact of Channel Delay

In this scenario, we study the impact of link delay on the performance of our congestion control algorithms. Each simulation generates a single flow of traffic through the bottleneck link that has a bandwidth of 6 Mb/s and delay varying from 50 ms to 300 ms. No random losses are introduced into the link.

Figure 8 plots the average throughput achieved by each algorithm when the bottleneck delay is varied. Overall, all variants are affected by high link delay, resulting in a decreasing of throughput with increasing RTT. STCP exhibits the most interesting behavior as it initially performs worse

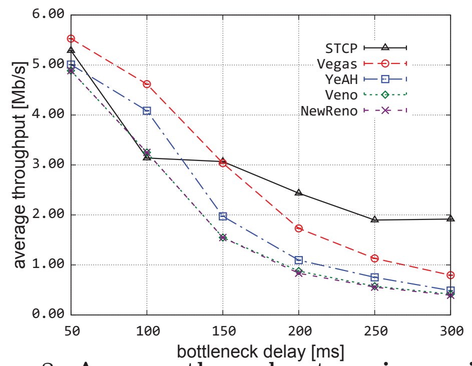


Figure 8: Average throughput vs. increasing delay.

than Vegas and YeAH, but starts to improve the throughput at the 150-second delay. Considering Vegas and its variants (Veno and YeAH), Vegas performs the best while Veno performs the worst. As explained above, with no random loss and available bandwidth not fully utilized (the high delays prevent all protocols from efficiently exploiting network capacity), Veno behaves exactly like NewReno. The non-aggressive behavior of the pure delay-based Vegas is an advantage in this case as its sending rate is more stable, resulting in fewer RTO occurrences.

5. CONCLUSIONS

We have presented our implementations of STCP, Vegas, Veno, and YeAH congestion control algorithms in ns-3 and studies of their behavior under various network conditions using a variety of metrics (robustness to random loss, TCP friendliness, intra-protocol fairness, and the impact of link delay) while verifying the correctness of the models. The results show that STCP is the most robust to random bit errors, and Vegas outperforms Veno and YeAH in the presence of non-congestive packet drops. While STCP and YeAH are the most aggressive algorithms, Vegas is the least when they have to share the bottleneck link capacity with a standard NewReno flow. The proactive congestion control mechanism that Vegas employs also prevents it from achieving intra fairness, although the same mechanism helps Vegas to better sustain its throughput than other protocols in a high propagation delay network.

For future work, we plan to study these variants under additional network scenarios to have a more complete picture of the algorithms' characteristics. We are also interested in experimenting with different values for the thresholds used in our implementations as we have seen from Section 4 that the different default values for α and β affect Vegas performance. In addition, we plan to continue to contribute to the ns-3 community with more TCP models, inclduing CTCP.

6. ACKNOWLEDGMENTS

We would like to acknowledge the members of the ResiliNets research group for their useful discussions and suggestions that helped us with this implementation. We would like to thank the anonymous reviewers for their helpful feedback on this paper. Finally, we would also like to thank Tom Henderson, Natale Patriciello, and the ns-3 development team for their timely responsiveness to guidance and issues with the ns-3 platform. This work was funded in part by NSF grant CNS-1219028 (Resilient Network Design for Massive Failures and Attacks).

7. REFERENCES

[1] The Network Simulator: ns-2. http://www.isi.edu/nsnam/ns, December 2007.
[2] The ns-3 Network Simulator. http://www.nsnam.org, July 2009.
[3] The ns-3 Network Simulator Doxygen Documentation. http://www.nsnam.org/doxygen, July 2012.
[4] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.
[5] A. Baiocchi, A. P. Castellani, and F. Vacirca. YeAH-TCP: yet another highspeed TCP. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.
[6] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, 1994.
[7] M. Casoni, C. A. Grazia, M. Klapez, and N. Patriciello. Implementation and Validation of TCP Options and Congestion Control Algorithms for ns-3. In *Proceedings of the 2015 Workshop on ns-3*, WNS3 '15, pages 112–119, New York, NY, USA, 2015. ACM.
[8] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), Jan. 2013.
[9] C. P. Fu and S. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on Selected Areas in Communications (JSAC)*, 21(2):216–228, 2003.
[10] S. Gangadhar, T. A. N. Nguyen, G. Umapathi, and J. P. Sterbenz. TCP Westwood Protocol Implementation in ns-3. In *Proceedings of the ICST SIMUTools Workshop on ns-3 (WNS3)*, Cannes, France, March 2013.
[11] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582 (Standards Track), 2012.
[12] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, 1988.
[13] V. Jacobson. Modified TCP congestion avoidance algorithm, April 1990.
[14] T. Kelly. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 33(2):83, Apr. 2003.
[15] B. Levasseur, M. Claypool, and R. Kinicki. A TCP CUBIC Implementation in ns-3. In *Proceedings of the 2014 Workshop on ns-3*, WNS3 '14, pages 3:1–3:8, New York, NY, USA, 2014. ACM.
[16] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.
[17] T. A. N. Nguyen, S. Gangadhar, M. M. Rahman, and J. P. Sterbenz. An Implementation of Scalable, Vegas, Veno, and YeAH Congestion Control Algorithms in ns-3 (extended). ITTC Technical Report ITTC-FY2016-TR-69921-04, The University of Kansas, Lawrence, KS, April 2016.