TCP Westwood(+) Protocol Implementation in ns-3

Siddharth Gangadhar, Truc Anh N. Nguyen, Greeshma Umapathi,
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
{siddharth, annguyen, greeshma, jpgs}@ittc.ku.edu

ABSTRACT

The poor performance of conventional TCP protocols in error prone channels is a well studied issue. Numerous optimizations to TCP to address this problem have been proposed. TCP Westwood is one such protocol engineered for use in wireless networks and employs a novel bandwidth estimation algorithm to determine the amount of data sent into the network in the presence of packet drops. In this paper, we present the implementation of the TCP Westwood and Westwood+ protocols in ns-3 and compare them against other existing ns-3 TCP implementations, TCP Tahoe, Reno, and NewReno. We validate our implementation by comparing performance of our implementation to the original work that introduced the Westwood protocols. In addition to validation, this paper also contributes as a performance evaluation of all existing ns-3 TCP protocols over selected network conditions.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: General, Model Development, Model Validation and Analysis; C.2.2 [**Computer-Communication Networks**]: Network Protocols — *transport protocols*

General Terms

Implementation, Analysis, Testing, Verification

Keywords

TCP Westwood, Westwood+, transport protocols, ns-3 network simulator, Tahoe, Reno, NewReno, performance evaluation, congestion and corruption loss

1. INTRODUCTION

TCP (Transmission Control Protocol) has been the dominant protocol for the Internet since the inception of the Internet. It had evolved for wired networks and with the emergence of wireless networks, been extended for use to the wireless domain. However, the sporadic wireless channel characteristics come with unique challenges such as frequent disconnects in the presence of mobile nodes and high errors. Temporary disruptions result in disconnecting existing TCP connections and the consumption of extra Round Trip Time (RTT) delays for new connection establishments. In cases of short duration disconnects, TCP consumes a much larger time to return to normal operation [17, 19] depending on the delays experienced by the networks. The high BER (bit error rate) environments of wireless channels also impose a major challenge to the functioning of TCP. Due to the inability of the protocol to distinguish between corruption based losses and losses due to network congestion [12], packet drops due to bit errors are incorrectly classified as congestion losses causing TCP to incorrectly invoke its congestion control algorithm. These factors cause a decrease in end-to-end goodput and the throughput of the network resulting in performance loss.

Over the years, numerous modifications have been added to the TCP protocol addressing how the protocol would react to packet losses caused by corruption. This has resulted in the emergence of multiple variants of TCP specifically engineered for wireless networks such as TCP Jersey [18], TCP Illinois [13], TCP Peach [2], TCP Veno [7], and TCP Westwood [14]. Techniques such as bandwidth estimation have been an integral part of most of these optimizations used to accurately estimate the bandwidth of the wireless channel in case of packet drops. Various mechanisms have been proposed in the literature that accurately estimate bandwidth. One such algorithm is the packet pair algorithm that relies on the difference in the inter-arrival times between a pair of packets sent during the same time interval to estimate bandwidth [5]. A second method calculates bandwidth based on the time of ACK receptions. Data is then transmitted as a function of the estimated bandwidth resulting in much higher performance when compared to the conservative nature of conventional TCP protocols.

A popular protocol specifically engineered for wireless networks is TCP Westwood [14]. TCP Westwood is a sender side modification that intelligently estimates the bandwidth and uses the calculated bandwidth to set the congestion window and the slow start threshold in case of packet losses. TCP Westwood+ is a modification of Westwood to handle overestimation of bandwidth in the presence of congestion [15]. This paper presents our implementation of the TCP Westwood and Westwood+ protocols in ns-3. The motivation behind choosing the protocols was to simulate pro-

Wns3 2013 March 5, Cannes, France.

tocols that tackle corruption based losses (TCP Westwood) and congestion (TCP Westwood+) in ns-3. This contributes to transport protocol simulation analysis in ns-3. The Westwood family are one of the early proponents of sophisticated techniques for higher performance such as bandwidth estimation.

The performance of the implemented protocols are compared with existing TCP protocols in ns-3, namely TCP Tahoe [9], TCP Reno [10], and TCP NewReno [6]. Validation with the original Westwood work [14, 15] is also carried out to verify the ns-3 implementation of our protocols. Furthermore, the effects of various network characteristics such as error rates, bottleneck bandwidth, and propagation delay are studied using throughput as our performance metric for the protocols.

The remainder of the paper is organized as follows: Section 2 provides a brief background on the various TCP protocols that are implemented in ns-3. Section 3 provides the algorithms and the implementation details of our TCP Westwood and Westwood+ protocols in ns-3. Section 4 presents the ns-3 simulations along with the validation of the implementation and a performance comparison of these TCP protocols in ns-3.

2. BACKGROUND AND RELATED WORK

In this section, we summarize the various TCP variants that are used in our simulations. Except for TCP Westwood and Westwood+, the implementations of TCP Tahoe, Reno, and NewReno already exist in the standard ns-3 release distribution.

2.1 TCP Tahoe

TCP Tahoe is the earliest version of TCP that introduces the well-known congestion control mechanism developed by Van Jacobson consisting of three main algorithms: slow-start, congestion avoidance, and fast retransmit [9]. The idea is for a TCP source (sender) that has no knowledge about the network status to probe the path by gradually increasing its sending rate dictated by the congestion window. At the beginning of a connection, the source employs the slow-start phase in which the congestion window is increased exponentially. Once the window reaches a certain threshold known as the slow start threshold, to prevent congestion, the TCP source transfers from the slow-start phase to the congestion avoidance phase during which the window is increased linearly. The source continues to stay in its congestion avoidance until a timeout occurs or a certain number of duplicate acknowledgments (DUPACK) are received that is interpreted as a signal for congestion over the path. Upon such an event, the TCP source enters the fast retransmit, halves its slow-start threshold, resets the congestion window to one full-sized segment, and immediately retransmits the missing segment. The source ends its congestion control cycle by going back to the slow-start phase and transmits to refilling the pipe.

2.2 TCP Reno

The TCP Reno protocol improves upon the performance of TCP Tahoe's congestion control algorithm, particularly for high bandwidth $\times$ delay product networks. TCP Reno uses a novel mechanism called fast recovery [10]. Based on the TCP's ACK based method, the receipt of any ACK with the inclusion of a DUPACK on the sender's side indicates the arrival of data at the receiver and is interpreted as bandwidth available in the network. To take advantage of the available bandwidth, instead of reseting to the slow-start phase after leaving the fast retransmit phase as in TCP Tahoe, TCP Reno employs the fast recovery mechanism. This allows the source to transmit a new segment on the receipt of each DUPACK assuming the source is not constrained by the sender's congestion window nor the receiver's advertised window. When a new ACK receives, the source transfers back to the congestion avoidance phase eliminating the reseting of the congestion window and the slow start stall times in the process.

2.3 TCP NewReno

TCP NewReno [8, 6] consists of a slight modification to Reno's fast recovery algorithm resulting in higher performance, especially in the presence of multiple segment losses per sending window. Unlike Reno, NewReno distinguishes between a full new ACK and a partial new ACK during its fast recovery phase. A full new ACK or a full ACK is an ACK that acknowledges all the outstanding segments before the sender enters the fast recovery while a partially new ACK or a partial ACK acknowledges only a fraction of the sent data. TCP NewReno treats a partial ACK as an indication that the segment following the ACK has been lost. It remains in the fast recovery retransmitting all of the missing segments until a full ACK is received. This optimization allows TCP NewReno to recover from multiple losses faster than TCP Reno by not having to wait for a retransmission timeout or re-enter fast retransmit.

2.4 TCP Westwood

TCP Westwood [14] is a sender side modification to TCP Reno that adjusts the sending rate based on a novel bandwidth estimation algorithm to improve performance in heterogeneous networks. The bandwidth estimation algorithm constantly monitors the rate of ACK reception while keeping an account on the number of DUPACKs and new ACKs. TCP Westwood allows the inclusion of DUPACKs in the calculation as each DUPACK indicates some data segments have reached the receiver. The amount of data acknowledged between ACK receptions is then used to compute the bandwidth of the link for the considered ACK interval. The calculated bandwidth is then passed through a Tustin approximation low pass filter to filter out the high frequency components. A simplified form of the filter [14] that is used in the implementation of the protocol is given as:

$$\hat{b}_k = a\hat{b}_{k-1} + \frac{1-a}{2}[b_k + b_{k-1}] \quad (1)$$

where a is a weighting parameter set to 0.9 and $\hat{b}_k$ is the filtered measurement of the available bandwidth at time instant t_k.

2.5 TCP Westwood+

TCP Westwood+ [15] is a modified version of Westwood with a slightly different bandwidth estimation algorithm to reduce Westwood's aggressiveness in the presence of ACK compression. ACK compression is an Internet phenomenon in which ACKs arrive at a much closer spacing than when they were generated due to queuing delay in the network [16]. Because Westwood estimates the network's bandwidth based on ACK inter-arrival times, ACK compression causes West-

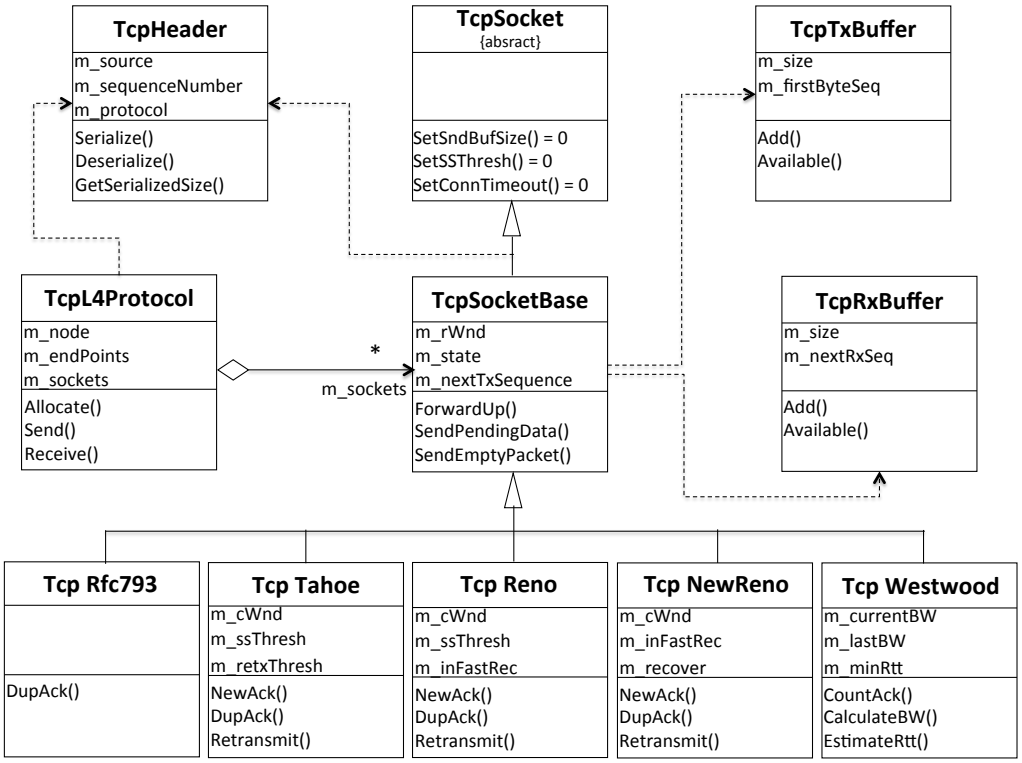


Figure 1: TCP class diagram

wood to overestimate the bandwidth, resulting in fairness problems in the existence of multiple TCP flows. To alleviate the effect of ACK compression, Westwood+ modifies the bandwidth estimation mechanism to perform the sampling every RTT instead of every ACK reception. The result is a more accurate bandwidth measurement that ensures better performance when comparing with Reno or NewReno, but still be fair when sharing the network with other TCP connections.

3. WESTWOOD IMPLEMENTATION

In this section, we first explain the basic TCP structure in ns-3 on which our implementation of TCP Westwood and Westwood+ is based. We follow this with the ns-3 implementation details of the Westwood and Westood+ algorithms.

3.1 TCP Class Interaction

Residing in the Internet module along with network protocols such as IPv4 and IPv6, the implementation of TCP consists of multiple classes communicating with each other to accomplish its designed goal – interacting with the network layer to offer reliable data transfer to applications. The main classes concerning the TCP implementation of ns-3 are listed below and their interaction is illustrated in Figure 1 [1].

- `TcpSocketBase`: This class provides key TCP features and a sockets interface for the application layer. Inherited from `TcpSocket`, `TcpSocketBase` serves as the base for all TCP variants.
- `TcpSocket`: This abstract class contains the essential attributes for a TCP socket.
- `TcpHeader`: This class implements the header for a TCP segment.
- `TcpTxBuffer`: This class provides a buffer for the sender to buffer any data received from the application before sending and acknowledgment.
- `TcpRxBuffer`: This class implements a buffer for the receiver to buffer data received from the network layer before passing it onto the application.
- `TcpL4Protocol`: Serving as an interface between the TCP socket and the network layer, `TcpL4Protocol` class is responsible for sending and receiving packets to and from the network layer. It is also responsible for checksum validation for incoming data.

Given the above main TCP classes, ns-3 contains multiple TCP variants implemented as child classes of `TcpSocketBase`. The existing variants are TCP Tahoe, TCP Reno, and TCP NewReno that we use in this paper to compare our TCP Westwood and Westwood+ implementation.

3.2 Westwood/Westwood+ Implementation

Similar to other existing TCP variants, both Westwood and Westwood+ are implemented in the same class that is inherited from `TcpSocketBase`.

3.2.1 Global Variables

The implementation consists of the following key global variables as information holders that assist the congestion control algorithm and the bandwidth estimation procedure:

- `m_cWnd` is a variable of type `uint32_t` to represent the congestion window. It is used by the sender to

determine the number of bytes that can be placed into the network without overloading the pipe. When a loss occurs, m_cWnd is determined using the estimated bandwidth.

- m_ssThresh is a variable of type `uint32_t` to represent the slow start threshold that marks the end of the slow start phase and the start of the congestion avoidance phase. Similar to m_cWnd, the value of m_ssThresh depends on the estimated bandwidth upon a loss.
- m_initialCWnd is an `uint32_t` variable that specifies the initial value of m_cWnd.
- m_inFastRec is a `bool` variable signaling the start and the end of the fast recovery phase.
- m_prevAckNo is of type `SequenceNumber32` that holds the last received ACK number.
- m_accountedFor is a variable of type `uint32_t` that keeps track of the number of DUPACK segments when a loss occurs and is used in the bandwidth estimation process.
- m_lastAck is a variable of type `double` that specifies the arrival time of the previous ACK.
- m_currentBW is a variable of type `double` that holds the current estimated bandwidth.
- m_minRtt is a variable of type `Time` that specifies the minimum RTT.
- m_lastBW is a variable of type `double` that holds the last estimated bandwidth after passing through the Tustin filter.
- m_lastSampleBW is a variable of type `double` that holds the value of the last sample of the measured bandwidth.
- m_ackedSegments is a variable of type `int` that holds the total number of acknowledged segments during the current RTT.
- m_IsCount is a `bool` variable signaling the beginning of m_ackedSegments counting process.
- m_bwEstimateEvent is a variable of type `EventId` that specifies a bandwidth sampling event.

3.2.2 Main Methods

Figure 2 depicts the flow of the Westwood and Westwood+ algorithms through the following main methods:

ReceivedAck: Taking the segment and its header as the two arguments, this method is invoked when an ACK is received by the sender. In the Westwood/Westwood+ class, it is responsible for calling the **CountAck** method. The **CountAck** call is followed by an **EstimateBW** call if it is TCP Westwood or by an **UpdateAckedSegments** call if it is Westwood+. The control is then transferred back to the **ReceivedAck** method in the `TcpSocketBase` in which the ACK is classified and corresponding functions such as **DupAck** or **NewAck** are invoked.

CountAck: As mentioned in the previous section, in order to estimate the bandwidth, the sender has to keep a count of the amount of data bytes that have been received by the receiver in consecutive ACK receptions for Westwood or during the last estimated RTT for Westwood+. The counting is accomplished by the **CountAck** method. It inspects incoming ACKs to infer the amount of data received and stores the value in a local variable cumul_ack that is further used in the bandwidth estimation process. At the beginning of the method, cumul_ack is set to the number of segments acknowledged by the current ACK. It is computed by subtracting the m_prevAckNo from the current ACK number retreived from the header. The **CountAck** procedure

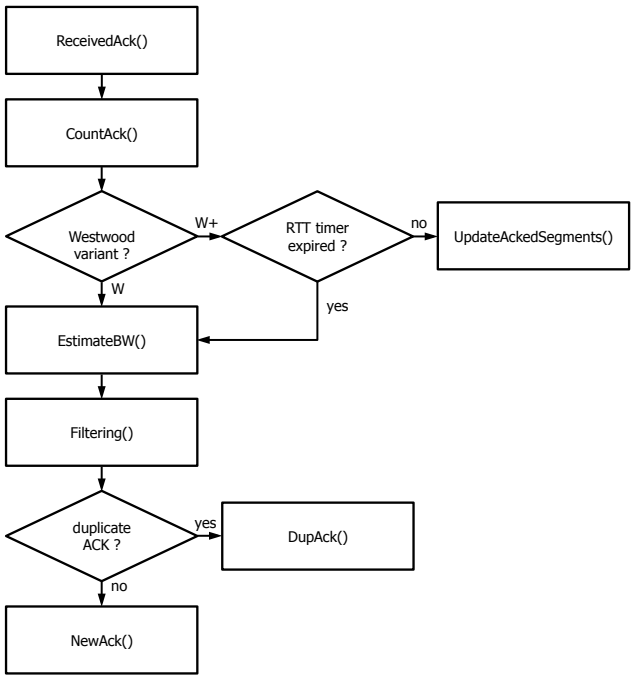


Figure 2: Westwood/Westwood+ flow chart

needs to keep track of the number of DUPACKs and makes sure they are correctly accounted for in the ACK counting process. If the cumul_ack is 0, the incoming ACK is a DUPACK, and cumul_ack is set to 1 indicating one segment counted towards the bandwidth estimation. The total number of DUPACKs, m_accountedFor is also incremented by 1. If the incoming ACK acknowledges more than one segment, the cumulative ACK is then compared against the DUPACK count. If the ACK acknowledges more segments than the DUPACK count, the difference is passed to the bandwidth estimation algorithm and the DUPACK count is reset to zero. Otherwise, all segments acknowledged by the ACK were already counted towards bandwidth calculation , and they should not be accounted again.

UpdateAckedSegments: In TCP Westwood+, the **EstimateBW** is only called when the RTT timer expires indicating the end of an RTT. While the timer is still running, upon the receipt of an ACK, the number of acknowledged segments (cumul_ack) returned by **CountAck** will be accumulated in m_ackedSegments. At the end of a sampling event (m_bwEstimateEvent), m_ackedSegments is reset back to zero to prepare for another event.

EstimateBW: This method uses the cumul_ack returned from the **CountAck** to estimate the bandwidth by employing Equation 2 for Westwood and Equation 3 for Westwood+:

$$\text{m\_currentBW} = \frac{\text{cumul\_ack} \times \text{m\_segmentSize}}{\textbf{Simulator::Now()} - \text{m\_lastAck}} \quad (2)$$

$$\text{m\_currentBW} = \frac{\text{m\_ackedSegments} \times \text{m\_segmentSize}}{\text{m\_lastRtt}} \quad (3)$$

In Equation 2, m_segmentSize is defined in `TcpSocketBase` and holds the size of each TCP segment. The method **Sim-**

ulator::Now() is invoked to measure the time of current ACK reception. In Equation 3, m_lastRtt specifies the last estimated RTT and is calculated using the **EstimateRtt** method in `TcpSocketBase`.

Filtering: The implementation allows the user to enable or disable the Tustin filter through the **Filtering** method. There are two modes of operation defined as `EnumValues`. In the default mode, the filter is disabled and the measured bandwidth is assumed to be the current bandwidth calculated from Equations 2 and 3. In the second mode, the filter is enabled and uses the Equation 4 for its computation. A local variable sample_bwe is used to store the measured bandwidth. Using a local variable α as a weighting parameter, the current bandwidth is calculated using the equation

$$\text{sample\_bwe} = \text{w1} \times \text{w2} \tag{4}$$

where

$$\text{w1} = \text{m\_currentBW} \times \alpha \tag{5}$$

and

$$\text{w2} = \frac{1-\alpha}{2} \times (\text{sample\_bwe} + \text{m\_lastSampleBW}) \tag{6}$$

NewAck: Following the same rules as in TCP Reno, **NewAck** adjusts m_cWnd and m_ssThresh values depending on the current sender state (slow-start, congestion avoidance, or fast recovery).

DupAck: Upon receiving a certain number of DUPACKs, **DupAck** adjusts the m_ssThresh based on m_currentBW using Equation 7. In case the current m_cWnd is greater than the m_ssThresh, the congestion window is set to m_ssThresh.

$$\text{m\_ssThresh} = \text{m\_currentBW} \times \text{m\_minRtt} \tag{7}$$

Retransmit: In the case of a retransmit timeout, this method is invoked to set the m_ssThresh and m_cWnd. The m_ssThresh is set in a similar manner as in the **DupAck** method and follows the Equation 7. The minimal allowed m_ssThresh is 2. The m_cWnd, on the other hand, is set to one m_segmentSize.

4. WESTWOOD EVALUATION

In this section, we present a performance validation of the TCP Westwood family of protocols based on the original work [14, 15] that introduced the protocols. The simulation setup is discussed in detail in Section 4.1. To show the effectiveness of the TCP Westwood family over other TCP variants, we make use of all existing TCP variants in ns-3, namely TCP Tahoe, Reno and NewReno[1]. The simulation results are presented and analyzed in Section 4.2.

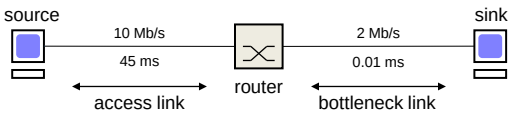


Figure 3: Corruption based simulation topology

[1]TCP SACK [3] is not in the standard ns-3 release but is currently under development by the ResiliNets group [11]

4.1 Simulation Setup

For our simulation topology that concentrates on corruption based losses, we use the same setup that was used in the original work [14] as shown in Figure 3. The topology uses a single source and sink interconnected using a gateway. We refer to the source-router link to be the access link and the router-sink link as the bottleneck link. The `PointToPointHelper` class was used for both the access and the bottleneck links along with the `RateErrorModel` class used for error generation to simulate the wireless bottleneck link. The bandwidths used for the access link and the bottleneck link are 10 Mb/s and 2 Mb/s respectively. The default propagation delay of the access link is 45 ms and the propagation delay of the bottleneck link is 0.01 ms. An MTU size of 400 B is used as the segment size for our simulations to compare to the original work [14, 15]. Errors are assumed to follow a uniform distribution and occur over the bottleneck link. We also use a default value of 2 as our delayed ACK count. Later, as part of our simulation scenarios, we vary the above parameters to observe the effects on protocol performance. We use the `BulkSendApplication` class for traffic generation for our performance evaluation. A single flow of traffic is used for simulations originating at the source and destined for the sink with a simulation running time of 600 seconds. Two simulation runs were performed with the standard deviations shown as error bars. The simulation parameters are summarized in Table 1.

Parameter	Values
Access link bandwidth	10 Mb/s
Bottleneck link bandwidth	2 Mb/s
Access link propagation delay	45 ms
Bottleneck link propagation delay	0.01 ms
Packet MTU size	400 B
Delayed ACK count	2 segments
Delayed ACK timeout	200 ms
Error model	Uniform error model
Error rate	0.005
Application type	Bulk send application
Simulation time	600 s

Table 1: Simulation parameters

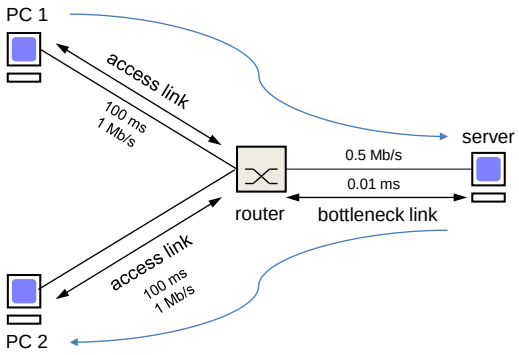


Figure 4: Congestion based simulation topology

To accurately validate TCP Westwood+, we use a different topology that concentrates on simulating ACK compression. The topology used for our simulations is shown in

Figure 4, resembling a dumbbell topology with two nodes labeled PC1 and PC2 on the left branch of a router and a server node to the right; this topology was used in the original TCP Westwood+ performance evaluation [15]. One flow is started from PC1 and destined for the server node and a second flow starting from the server destined for PC2. Thus, both flows share a common queue at the bottleneck router where ACK compression is experienced. It is also noted that the simulation parameters used for the scenario are different as the various parameters were not provided in detail in the original Westwood+ paper. We have an access link bandwidth of 1 Mb/s and a bottleneck bandwidth of 500 kb/s. In our setup, the second flow is started 2 seconds after the first flow. The various simulation parameters are included in Table 1.

4.2 Results

For our first scenario, we evaluate the performance of TCP Westwood and Westwood+ comparing with the other TCP variants in the presence of varying levels of error rates over the bottleneck link. We consider packet error rates (PER) to vary from 0.0001 to 0.05 similar to the original work [14]. The average throughput is plotted for varying PER levels for the TCP variants and is shown in Figure 5.

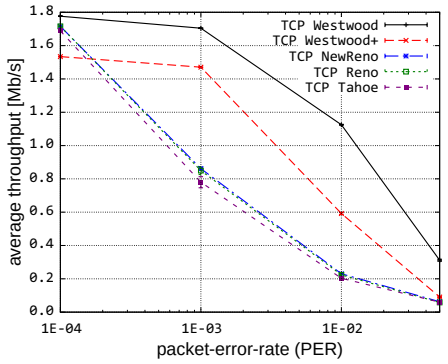


Figure 5: Throughput vs. PER of bottleneck link

From this plot, we see a huge performance improvement in TCP Westwood over the rest of the TCP variants. We also observe the similarity with the plot shown in Figure 9 in the original work [14] thus validating our implementation. The performance benefit is attributed to setting the congestion window and the slow start threshold as a function of the estimated bandwidth instead of a static conservative value as in the case of the other variants when a loss occurs. The next best performing protocol seen is TCP Westwood+. TCP Westwood+ samples the bandwidth every RTT period whereas Westwood performs the sampling every received ACK. Due to the higher sampling interval, Westwood+ takes a longer time to stabilize to the correct bandwidth when compared to Westwood. This gets worse in the presence of errors. If errors occurs before Westwood+ stabilizes, the low incorrect bandwidth estimate is used to determine the congestion window, causing a much slower sending rate. The higher the error rate, the higher the probability of bandwidth being underestimated. This is demonstrated in our plot. The performance of Westwood+ drops significantly starting from the error rate of 0.001. In summary, because of a shorter sampling interval, Westwood has a more up-to-date feedback of the network condition than Westwood+.

Another interesting observation is the similarity in performance of TCP Reno and TCP NewReno. This is because TCP NewReno improves on the performance of TCP Reno only in the case of multiple errors occurring in the same send window. The error model assumed follows a uniform distribution in our case and thus the probability of having multiple errors in the same window is low. A decrease in throughput is also observed as the error rate on the channel is increased for all considered protocols. This is intuitive as errors are increased, a larger amount of data sent and ACKs get lost triggering multiple retransmissions thus reducing the throughput of the network.

As part of our second scenario, we investigate the effects of varying the bottleneck propagation delay to our network. For this scenario, we assume a constant PER of 0.005 to occur over the bottleneck link. We vary the propagation delay of the bottleneck link from 0.01 ms to 250 ms in our simulations and compare the throughput achieved by the TCP variants. This scenario is also based on Figure 10 of the original Westwood work [14].

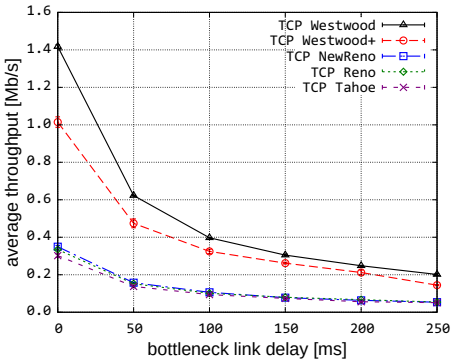


Figure 6: Throughput vs. bottleneck link delay

In Figure 6, we again observe the TCP Westwood and Westwood+ curve is comparable to the original work validating our implementation. The superior performance of Westwood and Westwood+ over the other protocols in Figure 6 is attributed again to the fact that while the other TCP variants perform conservatively, Westwood and Westwood+ attempt to fill up the pipe by using the bandwidth estimation algorithm that is independent of the level of corruption on the link. The lower performance of Westwood+

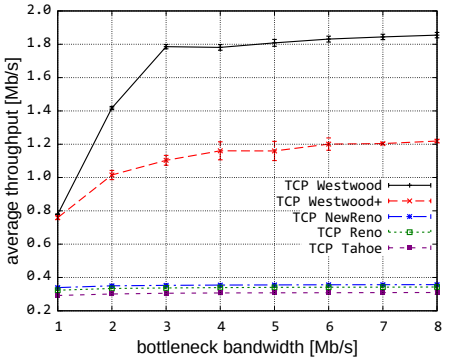
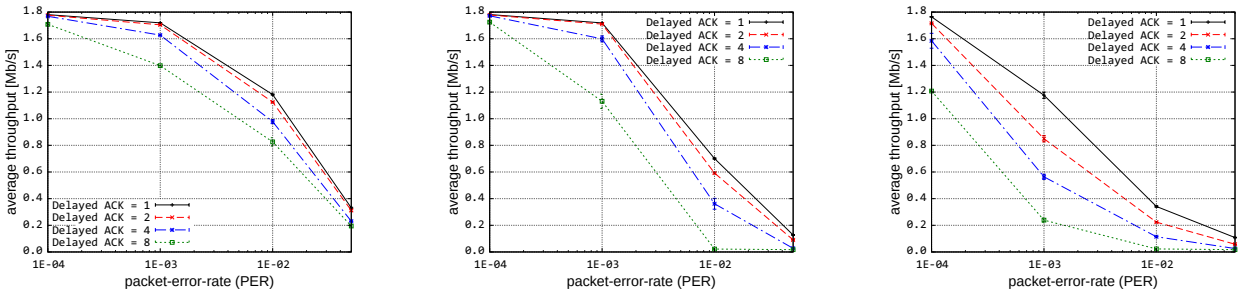


Figure 7: Throughput vs. bottleneck bandwidth



(a) Westwood (b) Westwood+ (c) Reno

Figure 8: Throughput vs. delayed ACK aggregation count

is attributed to the longer sampling interval as mentioned in the previous scenario. When comparing Reno performance to the original work, we see that for minimal propagation delays, there is a difference in the performance of Reno. This difference in performance is currently being investigated and as this concerns the performance of Reno ns-3 implementation, is beyond the scope of this paper.

Figure 7 shows the throughput achieved for different values of the bottleneck bandwidth. The bandwidth is varied from 1 Mb/s to 8 Mb/s and a constant packet error rate of 0.005 is set over the bottleneck link. The bottleneck delay used for this scenario is 0.01 ms. We see that TCP Westwood and Westwood+ outperform the other TCP variants. TCP Westwood is also observed to be comparable to the performance reported in Figure 11 of the original Westwood work [14]. The higher performance of TCP Westwood and Westwood+ can be attributed to the setting of the congestion window and the slow start threshold as a function of the estimated bandwidth following a loss. We also observe that as the bottleneck bandwidth is increased for both TCP Westwood and Westwood+, there is a performance improvement with higher throughput values noted. TCP Westwood outperforms Westwood+ significantly for the same reason as explained in the earlier scenarios. The performance of the other TCP variants are unaffected by the change in bandwidth. This is expected as the sending rate of these protocols is independent of the bandwidth of the link.

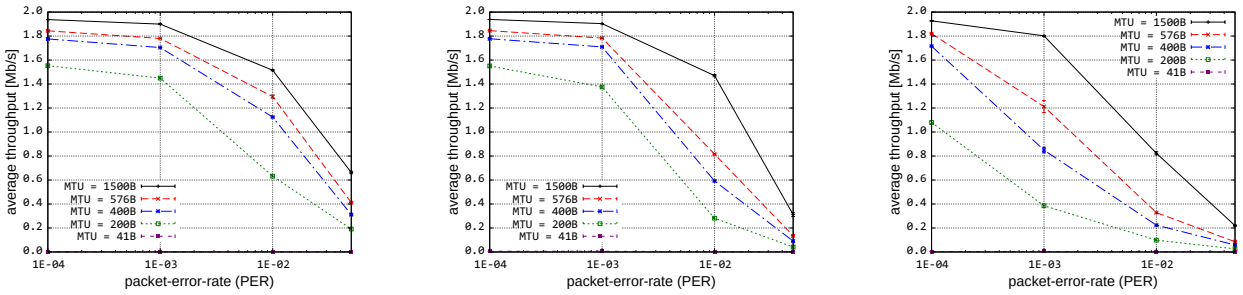
The bandwidth estimation algorithms of TCP Westwood and Westwood+ take into account the existence of delayed and cumulative ACKs and engineer the protocols to be able to handle these parameters. In order to test the robustness of the algorithms against different values of delayed ACKs, we plot the throughput of the network for different delayed ACK counts under error prone conditions using the TCP Westwood and Westwood+ and compare them with TCP Reno. TCP Reno is specifically chosen for the analysis as the Westwood family is a modification of TCP Reno.

The plot for this scenario is shown in Figure 8. The delayed ACK values are varied from 1 to 8 where a value of 1 ensures that every segment is acknowledged and a value of 8 means that the corresponding sink issues a cumulative ACK for every 8 segments received. A delayed ACK timeout of 200 ms is used for the network. We vary the PER of the bottleneck link using the same range that was employed in the first scenario.

We observe in all of the considered protocols that the curve showing the throughput for a delayed ACK count of 1 is seen to perform best while the throughput with a delayed ACK count of 8 performs worst. We observe a broader range of throughputs in TCP Reno across the various delayed ACK counts when compared to TCP Westwood. The highest difference in throughput observed for a specific error rate for Reno was close to 1 Mb/s whereas for Westwood, the highest difference was less than 0.4 Mb/s. This is because the TCP Reno algorithm is sensitive to the delayed ACKs and does not account for the cumulative ACKs received. There is also a much smaller change in throughput for different values of the delayed ACKs for TCP Westwood when compared to TCP Reno. The variations observed in TCP Westwood can be attributed to the stall times experienced at the source waiting for the ACKs to arrive in order to update the congestion window and transmit segments into the network. As the delayed ACK count increases, the sink waits longer before sending ACKs into the network thus delaying ACK reception at the source. This in turn causes a delay in the update of the sending rate thus contributing to a loss in throughput over the network. Westwood+, on the other hand, does exhibit a drop in performance especially over higher error rates of the bottleneck link. At higher rates, the incorrect bandwidth estimate is used with a higher probability and thus worsens the performance of the protocol.

Our next scenario investigates the effects of varying segment sizes on the performance of TCP Westwood and Westwood+ compared with TCP Reno. For this scenario, we compare various MTU sizes ranging from no payload to the more commonly used payload sizes of 1460 B for a 1500 B MTU. We use 41 B to represent the minimum payload case as the payload size is not allowed to be zero in the ns-3 simulator. The packet error rates are varied similar to the delayed ACK count case shown in the previous scenario. The plot for this scenario is shown in Figure 9.

We observe that the throughput achieved by an MTU size of 1500 B performs best and the MTU size of 41 B performs worst for all considered protocols. As the MTU size decreases, a decrease in performance is noted over the different error rates. Smaller segment sizes contribute to a lower rate of congestion window updates thus contributing to lower throughputs. Furthermore, smaller segment sizes require more packets to be transmitted over the network for the same amount of data. We see a much steeper decrease in throughput for Reno for the different MTU sizes



(a) Westwood (b) Westwood+ (c) Reno

Figure 9: Throughput vs. MTU sizes

when compared to TCP Westwood. Similar to the previous scenario, for any given error rate, we notice a much higher difference in throughput for TCP Reno with a maximum difference of approximately 1.4 Mb/s observed at a packet error rate of 0.001. The maximum difference in throughput observed for TCP Westwood is seen to be smaller than 1 Mb/s but observed at a much higher packet error rate of 0.01. The maximum difference for TCP Westwood+ is seen to be close to 1.2 Mb/s at an error rate of 0.01. As expected, TCP Westwood+ performs poorly at high error rates (greater than 0.001) as seen in the plot.

Our next scenario looks at the variation of the congestion window of TCP Westwood, Westwood+ and Reno in the presence of errors. The default simulation parameters with an access link of 10 Mb/s and a bottleneck link of 2 Mb/s are used in the presence of a 0.005 PER. The congestion window is traced and plotted for the entire simulation time of 600 s and the instantaneous values are sampled every 3 s over a single run. The plot is shown in Figure 10.

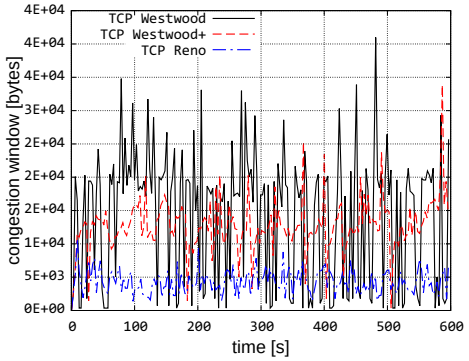


Figure 10: Congestion window comparison plot

As expected, we see that the congestion window of TCP Westwood is shown to be the highest in Figure 10. This is attributed to its bandwidth estimation process resulting in much higher throughputs as shown in the previous plots. As long as no congestion is experienced in the link, the high congestion windows or sending rates result in higher throughputs across the network. TCP Westwood+ performs second best for the given simulation parameters as expected and TCP Reno performs worst for the given scenario. It should be noted that although Westwood+ performs worse that Westwood for the above scenarios, there is no overestimation of bandwidth in congestion based scenarios. This allows Westwood+ to eliminate the aggressive behavior of Westwood thus being fairer to other TCP flows.

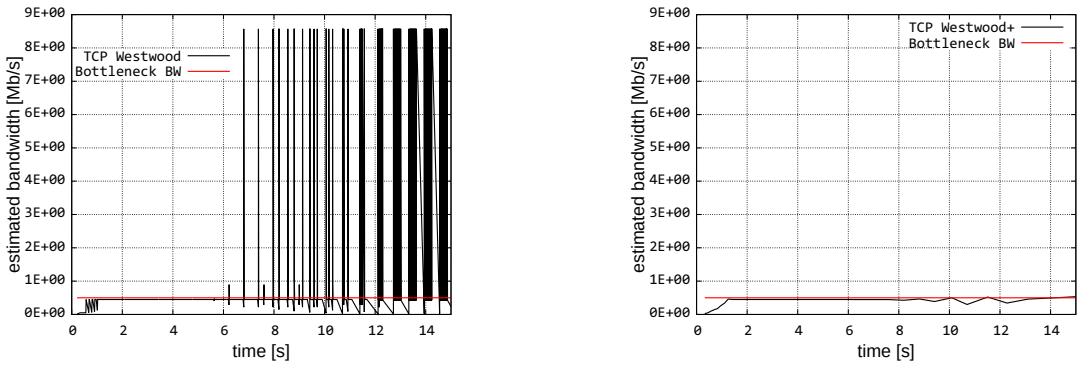
To validate TCP Westwood+, we consider a congestion related scenario that simulates ACK compression and trace the estimated bandwidth. The estimated bandwidth is traced out for a total simulation time of 20 s and is compared to the estimated bandwidth of TCP Westwood. The estimated bandwidth is sampled every microsecond of the simulation time and the plot is shown in Figure 11.

We observe in the case of Westwood that the estimated bandwidth fluctuates to a very high extent in the case of ACK compression. In a few cases, we see that the bandwidth is greatly overestimated to 8 Mb/s when the bottleneck bandwidth is only 500 kb/s. In the case of Westwood+, we observe a smoother estimation of the bandwidth around the expected value of 500 kb/s with slight fluctuations noticed between 8 and 14 s. During this fluctuation period, the maximum estimated bandwidth is only slightly higher than 500 kb/s and the minimum estimated bandwidth is approximately 300 kb/s. The advantage of Westwood+ is clearly shown in the presence of multiple flows where ACK compression is prevalent. Thus, TCP Westwood+ would be a better choice in the case of multiflow bidirectional networks in the Internet whereas Westwood would be a better choice for wireless links in the absence of congestion.

5. CONCLUSIONS

In this paper, we have presented our implementation of the TCP Westwood and Westwood+ protocols. Our implementation was validated using similar simulation scenarios as the original work. In addition to validation, we have also compared the Westwood family of protocols to other existing TCP implementations namely, TCP Tahoe, Reno and NewReno. From simulation results, we clearly saw the difference between the two protocols and the tradeoffs between TCP Westwood and Westwood+ in terms of congestion and aggressiveness. TCP Westwood could be used in over-provisioned cases in which there is no congestion and losses happen only due to corruption. TCP Westwood+ is better for congested scenarios in which ACK compression is prevalent and the usage of Westwood might further congest the network. Our simulation studies looked at the effects of parameters such as varying propagation delays, bandwidths, delayed ACK counts, and varying MTU sizes.

For future work, we plan to extend our study to include



(a) Westwood

(b) Westwood+

Figure 11: Estimated bandwidth in the presence of ACK compression

additional scenarios for our performance analysis. We will also compare to TCP SACK [3] and TCP Vegas [4], currently under development by [11]. We also plan to use real world protocols with the help of the Direct Code Execution (DCE) framework available as part of ns-3.

Acknowledgments

We would like to acknowledge the members of the ResiliNets research group for their advice, useful discussions and suggestions which helped us with this implementation. We would also like to thank Saverio Mascolo, Mario Gerla and Claudio Casetti for their help answering our questions regarding the protocols. We would also like to thank the anonymous reviewers whose feedback was not only helpful in the betterment of this paper but also provided valuable insight into our work. Finally, we would also like to thank Tom Henderson and the ns-3 development team for their timely responsiveness to guidance and issues with the ns-3 platform.

6. REFERENCES

[1] The ns-3 Network Simulator Doxygen Documentation. `http://www.nsnam.org/doxygen/group_tcp.html`, July 2012.

[2] I. Akyildiz, G. Morabito, and S. Palazzo. TCP-Peach: a new congestion control scheme for satellite IP networks. *IEEE/ACM Transactions on Networking*, 9(3):307–321, Jun 2001.

[3] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517 (Proposed Standard), Apr. 2003.

[4] 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.

[5] A. Capone, L. Fratta, and F. Martignon. Bandwidth estimation schemes for TCP over wireless networks. *IEEE Trans. on Mobile Computing*, 3(2):129–143, 2004.

[6] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), Apr. 2004.

[7] C. P. Fu and S. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE JSAC*, 21(2):216–228, 2003.

[8] J. C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *SIGCOMM '96*, pages 270–280, New York, NY, USA, 1996. ACM.

[9] V. Jacobson. Congestion avoidance and control. *SIGCOMM CCR*, 18(4):314–329, 1988.

[10] V. Jacobson. Modified TCP congestion avoidance algorithm, April 1990.

[11] e. a. James P.G. Sterbenz. Ns3-Models. `https://wiki.ittc.ku.edu/resilinets/Ns3-Models`, September 2010.

[12] R. Krishnan, J. P. G. Sterbenz, W. M. Eddy, C. Partridge, and M. Allman. Explicit transport error notification (ETEN) for error-prone wireless and satellite networks. *Computer Networks*, 46(3):343–362, 2004.

[13] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks. *ScienceDirect - Performance Evaluation*, 65:417–440, 2008.

[14] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, and R. Wang. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *MOBICOM 2001*, pages 287–297. ACM.

[15] S. Mascolo, L. Grieco, R. Ferorelli, P. Camarda, and G. Piscitelli. Performance evaluation of Westwood+ TCP congestion control. *Performance Evaluation*, 55(1-2):93–111, Jan. 2004.

[16] J. C. Mogul. Observing TCP dynamics in real networks. *SIGCOMM CCR*, 22(4):305–317, Oct. 1992.

[17] Y. Tian, K. Xu, and N. Ansari. TCP in wireless environments: problems and solutions. *IEEE Comm.*, 43(3):S27–S32, 2005.

[18] K. Xu, Y. Tian, and N. Ansari. TCP-Jersey for wireless IP communications. *IEEE JSAC*, 22(4):747–756, 2004.

[19] G. Xylomenos, G. Polyzos, P. Mahonen, and M. Saaranen. TCP performance issues over wireless links. *IEEE Comm.*, 39(4):52–58, 2001.