On the Goodput of TCP NewReno in Mobile Networks

(Invited Paper)

Sushant Sharma
Virginia Tech, Blacksburg, VA, USA

Donald Gillies
Qualcomm, San Diego, USA

Wu-chun Feng
Virginia Tech, Blacksburg, VA, USA

***Abstract*—Next-generation wireless networks such as LTE and WiMax can achieve throughputs of several Mbps with TCP. These higher throughputs, however, can easily be destroyed by frequent handoffs, which occur in urban environments due to shadowing. A primary reason for the throughput drop during handoffs is the out of order arrival of packets at the receiver. As a result, in this paper, we model the precise effect of packet-reordering on the goodput of TCP NewReno. Specifically, we develop a TCP NewReno model that captures the goodput of TCP as a function of round-trip time, average time duration between packet-reorder events, average number of packets reordered during every reorder event, and the congestion window threshold of TCP NewReno. We also developed an emulator that runs on a router to implement packet reordering events from time to time. We validate our NewReno model by comparing the goodput results obtained by transferring data between two hosts connected via the emulator to the goodput results that our model predicts.**

I. Motivation

Next-generation wireless technologies such as WiMax and LTE (long term evolution) offer very high data rates (on the order of several Mbps) to mobile users. As a result, mobile users will come to expect better peak performance from the networks than from current mobile networks. But with mobility comes a need for the base stations to perform frequent and transparent handoffs. For example, while driving for 30 minutes in the San Diego downtown area, we observed that permanent handoffs occurred every 12.21 seconds in a vehicular environment. Similarly, while walking for 10 minutes in a Qualcomm parking lot, rapid ping-pong handoffs occurred due to shadowing (i.e., signal blocking by buildings or the heads of users) every 5.43 seconds, on average.

Figure 1 shows an example of packet reordering during a handoff. In this figure, a mobile terminal that is connected to one base station is handed off to a new base station while transferring data to a remote host. After handoff, data from the old base station is routed to the new base station, before transmittal to the remote host. In this situation it is easily possible for the new base-station packets 5,6,7,8 to arrive at the remote host before packets 1,2,3,4 have arrived. To illustrate further, for a session transporting 1500 byte packets at 100 Mbps, a handoff every 5 seconds will amount to a bit-error-rate of at least $\frac{1500 \cdot 8}{5 \cdot 100\text{Mbps}} = 2.4 \times 10^{-5}$. Whereas, next generation networks must meet a packet error rate of 10^{-6} to achieve 100 Mbps throughputs [5], in the absence of handoffs. This shows that packets are disrupted by handoffs an order of magnitude more often than they are disrupted by packet losses.

Depending on the degree of reordering, the host may think that some packets are lost and ask for retransmission, resulting in a drop in the goodput of the flow. Similar reordering can occur if the host is transferring data to a mobile terminal.

Packet reordering is not limited to the scenario that we just described. It is well documented that packet reordering is not a rare event in the wired Internet [12], [13], [14], [15], and can cause severe degradation in the performance of TCP.

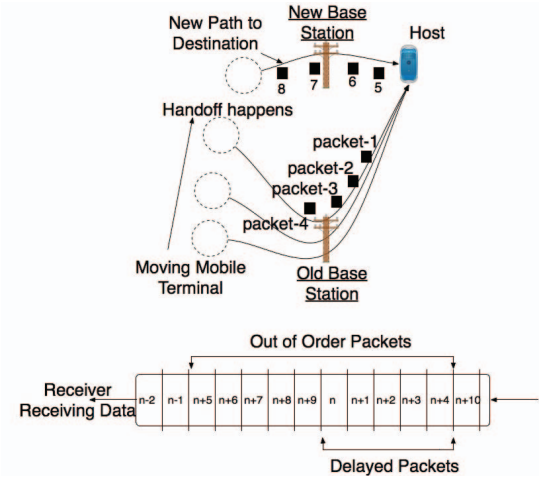


Fig. 1. Handoff description.

Though there exists many TCP throughput models [4], [5], [7], [8], [9], [10], [11], none of these models incorporate frequent packet reordering or stalling into their TCP performance models. The analytical modeling of TCP dynamics in the event of packet reordering is an important subject in wireless networks that is missing from the current literature. Our aim in this paper is to fill this gap and provide an analytical model of TCP NewReno that explicitly captures the effect of packet reordering.

In this paper we model the *goodput* of TCP NewReno, not the *throughput*. Goodput is defined as the number of unique packets delivered to an end host in a given amount of time, as opposed to the total number of packets transmitted in a given amount of time that includes retransmissions. The rationale behind modeling goodput is that the end users of a TCP flow are concerned with transferring unique packets and do not care about the number of retransmissions performed by the sender.

This paper makes the following unique contributions:

- We derive an analytical model of TCP NewReno's goodput when received packets are frequently reordered. Our model explicitly captures the effect of packet reordering on the goodput of TCP NewReno in combination with the fast recovery and fast retransmit mechanisms.
- We validate our model by providing experimental results of the data transfer between two physical machines in our lab. Both the machines were connected via a router running an air interface emulator (for 802.20 and WiMax) and performing packet reordering and packet discarding from time to time. Our model explains the significant drop in the goodput of TCP NewReno.

978-1-4244-7116-4/10/$26.00 ©2010 IEEE

TABLE I
NOTATION

$Flightsize$	Number of packets transmitted by sender but not yet acknowledged
$ssthresh$	The value of slow-start threshold
$cwnd$	Current size of sender's congestion window
$SMSS$	Sender's maximum segment size
RTT	Round trip time between the sender and the receiver
W	Limit on sender's congestion window
d	Number of packets that get delayed or arrived late at the receiver due to reordering
b	Number of received packets that a single ACK acknowledges
r	RTT-number after the packet-reorder event in which the ACKs due to the d delayed packets arrive at the sender
G_1	Goodput when delayed packets arrive after retransmitted packets, and window can be recovered
$\hat{G}_1$	Goodput when delayed packets arrive after retransmitted packets, and window can not be recovered
G_2	Goodput when delayed packets arrive before retransmitted packets, and window can be recovered
$\hat{G}_2$	Goodput when delayed packets arrive before retransmitted packets, and window can not be recovered
RP	Recovery Period: The period between two packet reorder events
A_h	Average time duration (in seconds) between two packet reorder events

II. TCP NewReno: Background

This section briefly explains the working of TCP NewReno as given in RFC 3782 [1]. We list the notation used in this paper in Table I.

When $FlightSize$ is less than $ssthresh$, TCP can increase its transmit rate exponentially; but when it is greater, transmissions increase linearly. When $FlightSize$ equals or exceeds $cwnd$, TCP must wait for acknowledgements, i.e., ACKs, which decrease $FlightSize$, before sending more packets.

Slow Start: While in slow start, the sender transmits two packets for every packet acknowledged by increasing $cwnd$ by one for every acknowledged packet. This increase continues until $cwnd$ equals or exceeds the slow-start threshold ($ssthresh$) or a packet loss is detected. After either event, the sender enters the congestion-avoidance state.

Congestion Avoidance: In this state, the value of $cwnd$ increases by $1/s$ (i.e., linearly), for every acknowledgment received where s is the size of congestion window (in packets) before the beginning of the current round trip time (RTT). This means that at the end of the current RTT, $cwnd$ will be increased by one. When three duplicate ACKs are received by the sender, indicating a packet loss (or reordering) the sender enters the fast-retransmit state.

Fast Retransmit: When the TCP enters the fast-retransmit state, the lost packet is retransmitted, and $ssthresh$ is set to $\texttt{max}(FlightSize/2, 2)$. The value of $cwnd$ is then reduced to $ssthresh + 3$, and the sender enters the fast-recovery state.

Fast Recovery: Fast recovery varies its behavior depending upon the type of ACK received. Repeated duplicate ACKs cause "hole-filling" retransmissions at the front of the TCP transmission window. New ACKs with progress in the sequence number "extend the TCP window" by causing new transmissions with higher sequence numbers. This second behavior is the "New" part of "TCP NewReno". For every duplicate ACK received by the sender, the sender increases the value of $cwnd$ by one packet and transmits a new packet if allowed by the $cwnd$ value. We assume that in fast recovery, the sender resets the retransmit timer upon receiving a partial ACK, which is known as the slow-but-steady variant of NewReno. We also assume if a sender receives a full ACK in fast recovery, it will set the $cwnd$ to $\texttt{min}(ssthresh, Flightsize+SMSS)$ and enter congestion avoidance. The sender responds to a timeout by reducing the value of $cwnd$ to 1 and entering slow start.

III. TCP NewReno: The Analytical Model

In developing our model, we consider packet reordering to be a periodic event during the transmission between sender and receiver. We define a recovery period (RP) as the time period between two successive packet reorderng events. A packet reordering event begins as soon as the receiver receives the first out-of-order packet. We divide TCP NewReno's response to packet reordering into the following two cases:

- **Case 1: Delayed packets arrive at the receiver *after* retransmitted packets.** The reordered packets get delayed significantly and arrive at the receiver after the packets that were retransmitted by the sender. This case is equivalent to when delayed packets are lost due to congestion or because the physical layer dropped them. This scenario can also occur during the handoff event for a mobile host in cellular or next-generation wireless networks. In this case, the receiver simply ignores the delayed packets upon reception.
- **Case 2: Delayed packets arrive at the receiver *before* retransmitted packets.** The acknowledgment at the sender because of delayed packets arrive before the sender is out of the fast recovery state.

For both cases, the RTT begins when the receiver detects the first out-of-order packet. Variable d denotes the number of consecutive packets that get reordered on average during every packet-reordering event. We denote A_h as the average time duration in seconds between two packet-reordering events, i.e. the average value of an RP. Since we consider that the TCP flows are window-limited [3], we denote W as the limit on the size of TCP's congestion window. The TCP receiver can consider different values for the number of packets it want to acknowledge with a single ACK. We denote b as the number of packets being acknowledged by single ACK.

A. Delayed packets arrive after retransmitted packets

Figure 2 describes the sender side behavior for this case. Since the d packets are so late, they appear to be lost, and the sender will never see the ACKs due to the reception of these packets at the receiver. As a result, during RTT1 in Fig. 2 there will be no ACKs received by the sender for first d packets. Since the sender is not receiving any ACKs, there will be no transmissions either. This period of no ACKs will be followed by three duplicate ACKs, which will trigger the sender to enter the fast retransmit state. The sender will then retransmit the packet with the sequence number requested in the duplicate ACKs, i.e. the first lost packet. The sender will also set the value of $ssthresh$ to $\max(FlightSize/2, 2)$.

Assume that before the packet loss, the sender's congestion window was W. So, the sender sets the value of $ssthresh$ to

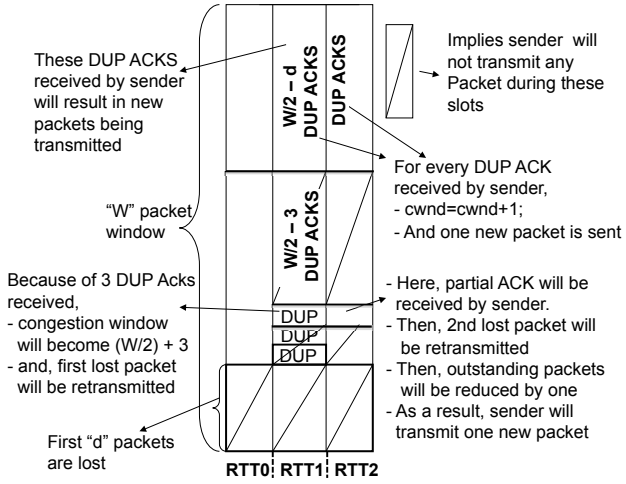


Fig. 2. Sender side of NewReno when d packets in a window W are lost.

$FlightSize/2$ or $W/2$. According to NewReno, the value of $cwnd$ is then reduced to $ssthresh + 3$, i.e. $cwnd = \left(\frac{W}{2} + 3\right)$ and enters fast recovery, and for every duplicate ACK received, $cwnd$ will increase by 1. Until the value of $cwnd$ becomes equal to W, which is the current $FlightSize$, the sender cannot send any new packets. This means that for the next $\left(\frac{W}{2} - 3\right)$ duplicate ACKs, the sender will not send any new packets. For the other $\left(\frac{W}{2} - d\right)$ duplicate ACKs that the sender will receive in RTT1, it will send one new packet per ACK. So, the sender will send $\left(\frac{W}{2} - d\right)$ new packets in RTT1.

Next, as shown in Figure 2, the sender will receive a partial ACK for the first retransmission in RTT1. Upon receiving partial duplicate ACK, TCP NewReno will retransmit the next unACKed packet. Note that it takes a full retransmit window (W) to retransmit the next unACKed packet. At this moment, the $FlightSize$ is reduced by 1, and the sender can transmit another new packet to the receiver. During RTT2, for the $\left(\frac{W}{2} - d\right)$ duplicate ACKs, the sender sends 1 new packet for every duplicate ACK received.

So, for next d RTTs, the number of new packets transmitted by the sender, or the size of $cwnd$, will increase by 1 per RTT. After d RTTs, NewReno exits fast recovery, and the new packets transmitted by the sender will increase by $1/b$ per RTT.

Depending on the duration between two packet-reordering events, a goodput model can be developed for two different scenarios. In one, A_h is large enough so that the sender can grow its congestion window back to W. In the other, A_h is so small that the sender cannot grow its congestion window back to W. Since A_h is the average duration of an RP, $\frac{A_h}{RTT}$ denotes the number of RTTs in an RP for both scenarios. Below we model the goodput of these two scenarios.

Window can be recovered. Figure 3 shows the number of new packets transmitted by the sender in $\frac{A_h}{RTT}$ RTTs after the RTT in which d packets were lost. The area under the curve represents the overall goodput when the window recovers to W. This area can be calculated by subtracting the area of `Region1` and `Region2` from the total area of Figure 3.

From Figure 3, the areas of `Region1` and `Region2`, ar_1 and ar_2, respectively, are

$$ar_1 = d\frac{W}{2} + \frac{d^2}{2}, ar_2 = b\frac{W^2}{8}$$

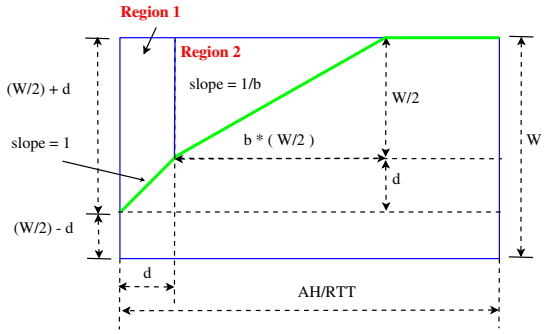


Fig. 3. New packets transmitted by the sender during an RP in which window was recovered back to W

So, the total data transferred in A_h time is given by $\left(W\frac{A_h}{RTT} - ar_1 - ar_2\right)$, i.e.,

$$W\frac{A_h}{RTT} - b\frac{W^2}{8} - d\frac{W}{2} - \frac{d^2}{2} \quad (1)$$

If we divide (1) by the total amount of time used for the transfer (i.e., A_h), the goodput G_1 is

$$G_1 = \frac{W}{RTT} - \frac{1}{A_h}\left[b\frac{W^2}{8} + d\frac{W}{2} + \frac{d^2}{2}\right] \quad (2)$$

Window cannot be recovered. Figure 4 shows the scenario when the RP is so small that the sender cannot grow its congestion window back to W. By comparing the width of `Region2` in Figure 3 and the width of `Region4` in Figure 4, the minimum number of packets that are required to be lost so that the sender can not grow its congestion window back to W must satisfy $b\frac{W}{2} > \left(\frac{A_h}{RTT} - d\right)$. Rewriting this condition we arrive at Eqn. (3):

$$\frac{A_h}{RTT} < d + b\frac{W}{2} \quad (3)$$

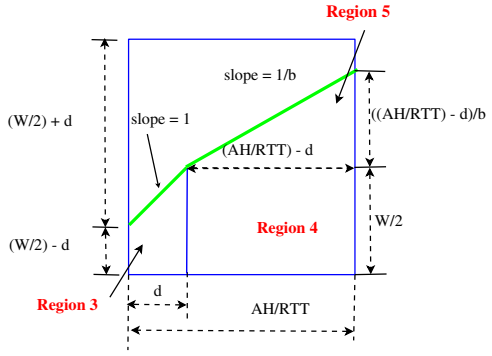


Fig. 4. New packets transmitted by the sender during an RP in which window was not recovered back to W

From Eqn. (3), $\frac{A_h}{RTT}$ is the number of RTTs in an RP, and $\left(d + b\frac{W}{2}\right)$ is the minimum number of RTTs that are required to grow the window back to W. This also tells us that the larger the number of packets that a single ACK acknowledges (i.e. b), the larger the time required by the sender to grow the congestion window back to W.

If the sender cannot recover the window back to W, the value of window at the start of the subsequent RPs decreases,

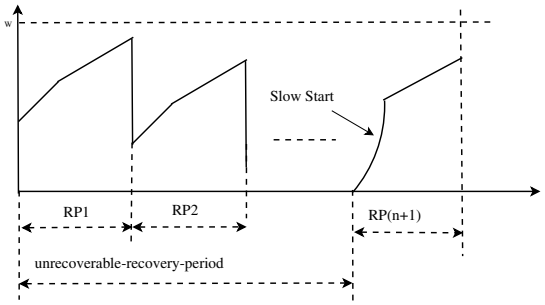


Fig. 5. Unique packets transmitted when the sender was unable to grow the congestion window back to W

causing the sender to timeout after a certain number of RPs, as shown in Figure 5. In the n^{th} RP, the sender timeouts and enters slow start. Calculating n will help us calculate the goodput during these n RPs, as shown below.

First, we use Figure 4 to derive a general expression of the window size after the n^{th} RP when d packets were lost at the start of n^{th} RP:

$$W_n = \frac{W_{n-1}}{2} + \frac{1}{b}\left(\frac{A_h}{RTT} - d\right) \tag{4}$$

where W_n is the size of congestion window after n^{th} RP, and W_{n-1} is the value of congestion window after $(n-1)^{th}$ RP.

The recurrence relation of Eqn. (4) can be solved to get:

$$W_n = \frac{W_0}{2^n} + \frac{2}{b}\left(\frac{A_h}{RTT} - d\right)\left(1 - \frac{1}{2^{n-1}}\right), n > 1, \tag{5}$$

Next, because we know the value of congestion window at the start of the first RP, i.e., W_0, and can derive the value of congestion window after first RP, i.e., W_1, using Eqn. (4), we can use these values, along with Eqn. (5), to calculate the value of n for which W_n is $d+2$:

$$n = log_2\left(\frac{W - \frac{4}{b}\left(\frac{A_h}{RTT} - d\right)}{d + 2 - \frac{2}{b}\left(\frac{A_h}{RTT} - d\right)}\right) \tag{6}$$

Second, we also use Figure 4 to derive a general expression for the amount of data transferred in n^{th} RP for the scenario where the window *cannot* be recovered back to W. From Figure 4, the total amount of data transferred is the sum of the areas of `Region 3`, `Region 4`, and `Region 5`.

$$D_n = \frac{d^2}{2} + d\left(\frac{W_{n-1}}{2} - d\right) + \left(\frac{A_h}{RTT} - d\right)^2\left[\left\{\left(\frac{W_{n-1}}{2}\right)\Big/\left(\frac{A_h}{RTT} - d\right)\right\} + \frac{1}{2b}\right], \tag{7}$$

So, the goodput obtained during the the n RPs is given by

$$\hat{G}_1 = \frac{1}{nA_h}\sum_{i=1}^{n} D_i, \tag{8}$$

where D_i is given by Eqn. (7) and n is given by Eqn. (6).

Whether or not the window recovers back to W after the $(n+1)^{th}$ RP will depend on the $ssthresh$ value at that time, which will be $d/2$. Figure 6 shows a detailed growth in the new packets transferred during $(n+1)^{th}$ RP, and the size of window reached at the end of $(n+1)^{th}$ RP, in terms of packets will be

$$\frac{d}{2} + \frac{1}{b}\left(\frac{A_h}{RTT} - log_2\frac{d}{2}\right), \tag{9}$$

For a more accurate analysis, we can use the value in (9) as the value of W_0 in our analysis of Eqns. (5), (6), (7), and (8).

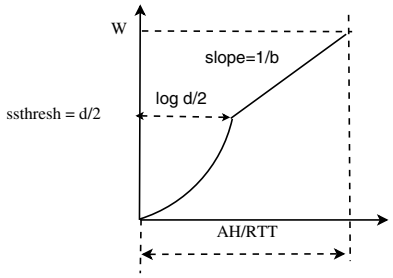


Fig. 6. New packets transferred during $(n+1)^{th}$ RP

B. Delayed packets arrive before retransmitted packets

Here the sender receives the ACKs from the delayed packets before it exits fast recovery. We first explain the behavior of TCP NewReno for two special cases: (1) ACKs due to delayed packets arrive in the first RTT after the RTT in which packets were reordered. (2) ACKs due to delayed packets do not arrive in the first RTT after the RTT in which packets were reordered, but arrive before the next packet-reorder event. After explaining the behavior of sender for these two cases, we will develop two goodput equations, one for when the congestion window can be recovered back to W and the other when the congestion window cannnot be recovered back to W.

ACKs arrive in first RTT after packet-reorder event. Figure 7 shows how the sender behaves when ACKs due to the delayed packets arrive in RTT1. During RTT1, the very first ACKs that the sender receives will be three duplicate (DUP) ACKs due to the reordering of d packets. Upon receiving the third DUP ACK, the sender retransmits the first packet that was delayed. Then, according to NewReno specification, the sender sets the value of $ssthresh$ to $W/2$ and the value of $cwnd$ to $[(W/2)+3]$. We can assume that the delayed packets at the receiver may not arrive in sequence. Thus, the sender will receive a mix of partial ACKs (due to delayed packets) and duplicate ACKs (due to non-delayed packets). For every duplicate ACK received, the sender increases the value of $cwnd$ by 1, and for every partial ACK received, the sender reduces the number of outstanding packets depending upon the packet number being ACKed. Figure 8(a) shows the number of new packets transmitted by sender in the RTTs after the RTT in which packets got reordered. The window growth in Figure 8(a) stops when the congestion window reaches its limit (i.e. W), or if another packet reorder event happens before the window can be recovered back to W.

It is possible that the sender may end up retransmitting up to $(W-3)$ old packets in RTT1. To explain, let us assume that only first packet got delayed and arrives as the fourth packet. The ACKs due to the remaining packets in this RTT will be considered as partial ACKs. Why? After receiving three DUP ACKs, the sender enters the fast-retransmit state, and these ACKs do not cover the last new packet transmitted before the

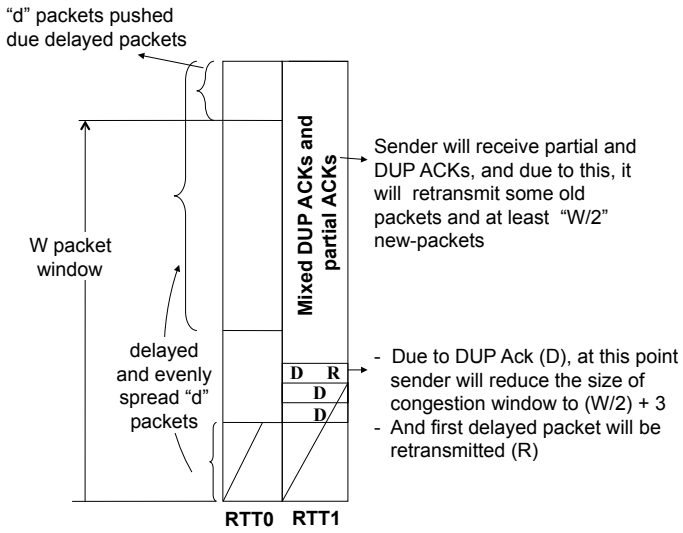


Fig. 7. Sender behavior when ACKs due to delayed packets arrive in RTT1.

sender entered fast retransmit. Hence, every partial ACK will result in retransmission of the next unACKed packet. Such retransmissions can be avoided via the SACK option in TCP (though SACK will not affect the *goodput* in this case).

ACKs arrive in $\mathbf{r^{th}}$ RTT after packet-reorder event, where $\mathbf{r \leq d}$. This scenario can be regarded as the general case for the previous scenario, where the ACKs due to delayed packets arrived in the very first RTT. Here the value of r is less than or equal to the value of d; otherwise this scenario will be same as the scenario where the delayed packets were lost because after the d^{th} RTT, the sender will be out of fast recovery.

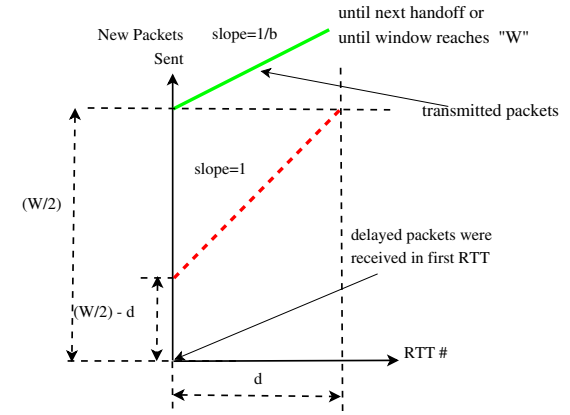
We can see that before the r^{th} RTT, similar to the case when delayed packets were lost, the number of new packets transmitted by the sender will increase by one during every subsequent RTT. This is shown in Figure 8(b). After the r^{th} RTT, the sender will come out of the fast-recovery state, the congestion window size will be set to $ssthresh = W/2 = \min(flightsize, ssthresh)$, and the congestion window size will now increase by $1/b$ packets per RTT. The size of congestion window will grow until it reaches its limit (i.e. W), or another packet reorder event happens that will prevent the congestion window from recovering back to W.

For the case when ACKs due to delayed packets arrive in r^{th} RTT, we can consider two sub-cases. One, in which the sender was able to recover the congestion window back to W. And the other sub-case, in which the sender was not able to recover its congestion window back to W.

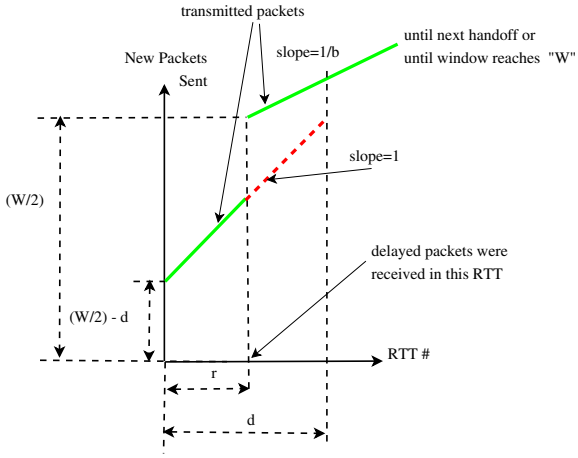
Figure 9(a) shows the growth in the window size when the sender was able to recover it congestion window back to W before the next packet-reorder event. The goodput equation for this sub-case can be given as follows

$$G_2 = \frac{W}{RTT} - \frac{1}{A_h}\left[\frac{b}{8}W^2 + r\left(\frac{W}{2} + max(0, d-r)\right) + \frac{r^2}{2}\right] \quad (10)$$

The derivation of Eqn. (10) is similar to the analysis presented for deriving Eqn. (2). Eqn. (10) is a more general form for



(a) ACKs due to delayed packets arrive in first RTT after the RTT in which packets were reordered ($r = 1$).



(b) ACKs due to delayed packets arrive in r^{th} RTT after the RTT in which packets were reordered ($r < d$).

Fig. 8. New Packets sent per RTT by NewReno sender when ACKs due to delayed packets arrive before end of fast-recovery.

Eqn. (2). That is, when r becomes greater than d (i.e., delayed packets were lost), Eqn. (10) reduces to Eqn. (2).

For the sub-case when the sender cannot recover the congestion window back to W, we perform a similar analysis to what we did in the case when delayed packets were lost and the congestion window was not recovered. Figure 9(b) shows the growth in congestion window for this sub-case. Since the window cannot be recovered, the size of congestion window becomes smaller and smaller in every subsequent RP. Finally, the size of congestion window reaches $d+2$, and upon losing d packets out of $d+2$, the sender timeouts after this RP. If we denote the RP after which the sender timeouts as the m^{th} RP, we can calculate m as we did for Eqn. (6):

$$m = log_2\left(\frac{W_0 - \frac{4}{b}\left(\frac{A_h}{RTT} - r\right)}{r + 2 - \frac{2}{b}\left(\frac{A_h}{RTT} - r\right)}\right), \quad (11)$$

where the value of W_0 is the initial size of congestion window. To calculate the goodput, W_0 can be considered as the size of congestion window after the m^{th} RTT in which the sender will time out. (Note that the value of the m in Eqn. (11) is a general

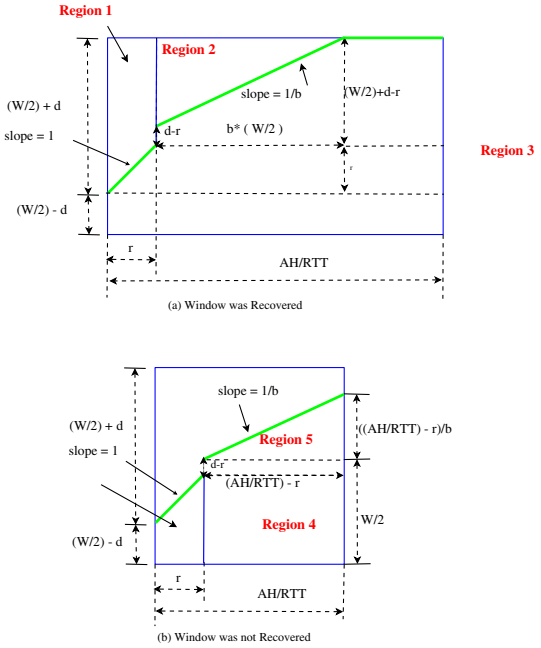


Fig. 9. ACKs due to the d delayed packets arrive in r^{th} RTT after the RTT in which packets were reordered.

form for the value of n in Eqn. (6)). The value of m becomes equal to n for $r = d$. The value of W_0 does not depend on the value of r, and can be given by Eqn. (9). Again, similar to the analysis performed for developing Eqn (7), we can develop a general equation for the total new data transferred during m^{th} RP. This can be given as

$$\hat{D}_m = \frac{r^2}{2} + r\left(\frac{\hat{W}_{m-1}}{2} - d\right) + \left(\frac{A_H}{RTT} - r\right)^2 \left[\left\{\left(\frac{\hat{W}_{m-1}}{2}\right) \Big/ \left(\frac{A_H}{RTT} - r\right)\right\} + \frac{1}{2b}\right] \quad (12)$$

where the window size after m^{th} RTT can be given by

$$\hat{W}_m = \frac{W_0}{2^m} + \frac{2}{b}\left(\frac{A_h}{RTT} - r\right)\left(1 - \frac{1}{2^{m-1}}\right), m > 1. \quad (13)$$

The goodput for this scenario, where the ACKs due to the d delayed packets arrive in r^{th} RTT, can be given by

$$\hat{G}_2 = \frac{1}{mA_h}\sum_{i=1}^{m}\hat{D}_i, \quad (14)$$

The derivation of Eqns. (12) and (13) follow the same approach as for Eqns. (5) and (8). As a result, the discussion has been omitted to save space. It is easy to see that Eqns. (13) and (14) are the general form of Eqns. (5) and (8).

We can also give a general form for Eqn. (3), which provides the condition that will prevent the sender to grow back its congestion window to W after a packet drop event, as

$$\frac{A_h}{RTT} < r + b\frac{W}{2}. \quad (15)$$

An interesting observation is that the Eqn. (15) does not contain d. This suggests that when the ACKs due to delayed packets arrive in the r^{th} RTT ($r < d$), then whether window can be recovered or not, is independent of the number of packets that get delayed, i.e. d.

We can now say that Eqns. (10) and (14) constitutes the analytical goodput model developed in this paper. Next, we will present some experimental results to validate our model.

IV. Experimental Results and Model Validation

Eqns. (10) and (14) can be used to predict goodput of a TCP NewReno flow as a function of RTT, window limit, average time duration between two packet-drop events, and the average number of packets reordered (or dropped) during every event. In this section, we validate these equations by presenting results of data transfer between two physical hosts within our lab setup.

A. Experimental Setup

The experimental setup is shown in Figure 10. Both the sender and receiver run Linux kernel 2.6.21.

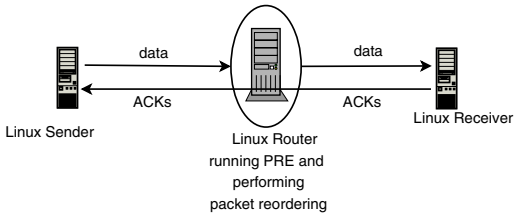


Fig. 10. Experimental setup within our lab.

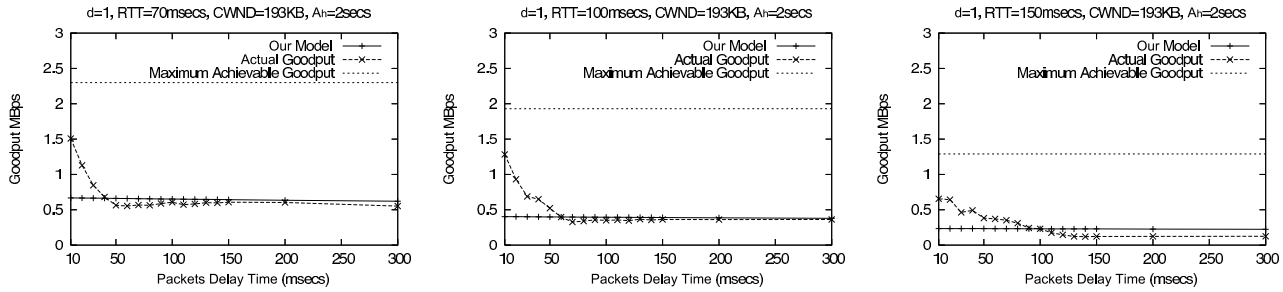
The router that connects sender to the receiver is running the *Packet Reorder Emulator* (PRE). PRE was written to explore wireless MAN air interface designs at our lab. With appropriate parameters, PRE can emulate 802.20 or WiMax. It also includes a simple handoff model. The PRE is parameterized, and the most important parameters with their default values are:

- Slot Size (911 microseconds)
- Air Interface Total Throughput (2.33 MBps)
- Backhaul Delay (20 milliseconds RTD)
- Handoff Interarrival (7.5 secs), Pushout (200 ms), and Packets-Pushed (10).

The throughput limitation of 2.33 MBps emulates the 802.20 wireless air interface bottleneck. The handoff interarrival time is the time between handoff events. Handoff is emulated by "pushing" a certain number of packets into the future, so that they are delivered out-of-order and late. The "pushout" is the amount of delay into the future, and the "packets pushed" is the number of packets that will be pushed. To help keep our experiments deterministic, handoff events were kept periodic.

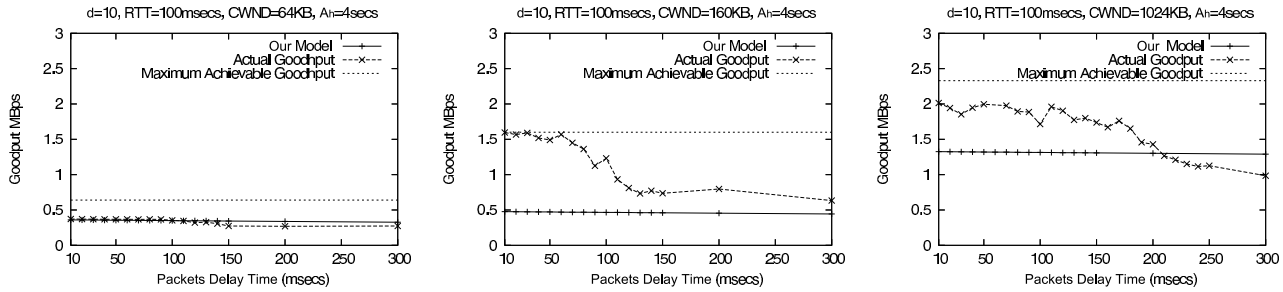
B. Measurements while Emulating Handoffs

Here we present the measurement data collected while transferring 50 MegaByte file between two hosts that were connected through the router running PRE. PRE was used to emulate different values of RTT, different durations between two packet-reorder events (A_h), and different number of packets getting reordered (d). Several different limits on window size (W) were used during the experiments by using TCP system control (sysctl) variables within the linux kernel. We collected data under different combinations of the model



(a) RTT=70 msecs (b) RTT=100 msecs (c) RTT=150 msecs

Fig. 11. Comparison of actual goodput and the goodput calculated using our model at different RTTs.



(a) 64K Window Limit (b) 160K Window Limit (c) 1024K window Limit

Fig. 12. Comparison of actual goodput and the goodput calculated using our model for window limited flows.

parameters in order to validate the models presented in (10) and (14).

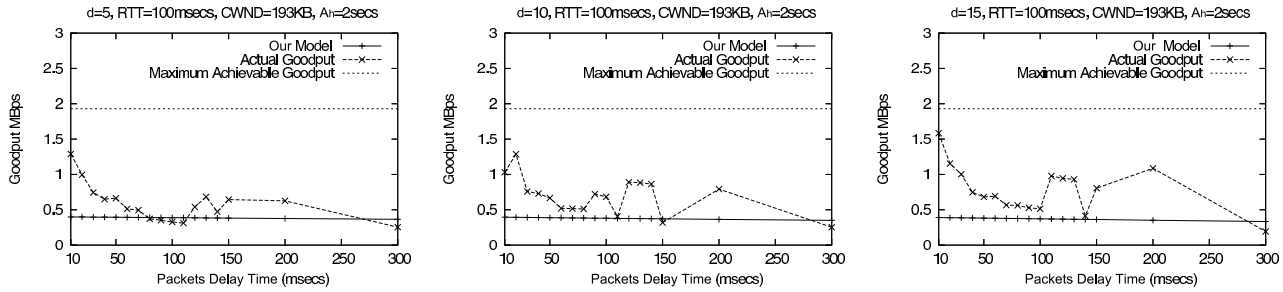
Variable RTT. Our first set of experiments were conducted while keeping the values of d, W, and A_h as constants. Value for the number of packets that got delayed due to reordering (i.e. d) was set to 1. Limit on congestion window (i.e. W) was set to 193 KiloBytes, and the value of time duration between two packet reorder events (i.e. A_h) was set as 2 seconds. We varied the values of RTT from 70 to 150 milisecs. Figures 11(a) to 11(c) presents the comparison of goodput calculated using Eqns. (10) and (14) (as applicable), and the actual goodput achieved while transferring the data between two hosts connected via PRE. X-axis in Figures 11(a) to 11(c) shows the amount of time by which d packets got delayed and y-axis show the achieved goodput. The values on X-axis are used to calculate the value of r, i.e. the RTT in which the ACKs due to delayed packets arrive at the sender.

By looking at the results presented in Figures 11(a) to 11(c), it may seem that our model is not very accurate when the delayed packets arrive the first RTT after packet-reorder event. This is because the actual goodput obtained when the packets arrive in the first RTT after the packet reorder event is higher than the goodput calculated using our goodput model. But our model is accurate, and this seemingly inaccurate behavior can be explained by looking at the implementation details of TCP NewReno in the Linux kernel. After looking at the implementation of NewReno in linux kernel, it seems that when the ACKs due to the delayed/reordered packets arrive at the sender before the end of the first RTT after which packets got reordered, the NewReno sender is overly aggressive in increasing its congestion window size. This fact is also documented by Sorolahti and Kuznetsov in [6]. They explain that the Linux fast recovery do not completely follow the behavior given in RFC 3782, and as a result when the reordered packets arrive in the first RTT after packet reorder event, the goodput results will be higher than those calculated by our model. Our result confirm the documented behavior of Linux in [6]. This behavior of NewReno implementation in Linux kernel will be present in rest of our experiments as well.

Variable Congestion Window. In our next set of experiments, we varied the value of the limit on congestion window, i.e. W, while keeping the values of d, RTT, and A_h as constants. The value of A_h was set to be 4 secs, d as 10, and RTT as 100 msecs. Figures 12(a) to 12(c) shows the results for these experiments. An interesting observation here is that increasing the congestion window size can not significantly improve the goodput of a NewReno flow if the flow is experiencing regular packet reorder events.

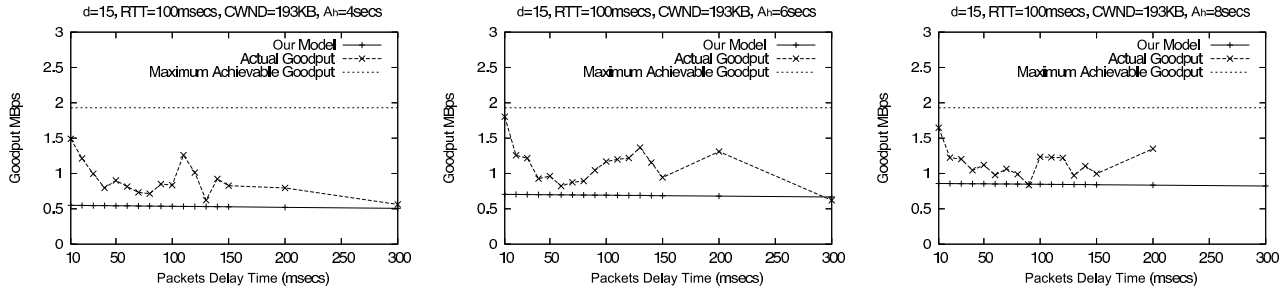
Variable Reordered Packets. For these experiments, we varied the number of packets that gets reordered, i.e. d, while keeping the values of RTT, W, and A_h as constants. The value of RTT was kept constant at 100 msecs, limit on congestion window W as 193 KB, and A_h as 2 secs. Figures 13(a) to 13(c) shows the results of our experiments for these settings. From the results in Figures 13(a) to 13(c), we can observe that if the number of packets getting reordered increase, the further drop in goodput is not significant. It means that as few as one packet is sufficient to cause a significant drop in the goodput of a flow.

Variable A_h. In these experiments, we varied the value of A_h, i.e. the time duration between two packet reorder events. We kept the values of W, RTT, and d as constants. Figures 14(a) to 14(c) show the results for this scenario. These figures show that the packet reordering has already caused the goodput to



(a) 5 Packet Reordering (b) 10 Packet Reordering (c) 15 Packet Reordering

Fig. 13. Comparison of actual goodput and the goodput calculated using our model for different number of delayed packets.



(a) 4 second time between two handoffs (b) 6 second time between two handoffs (c) 8 second time between two handoffs

Fig. 14. Comparison of actual goodput and the goodput calculated using our model for different handoff frequencies.

drop to a significantly low value and marginally increasing the time duration between two packet reorder events will not help in improving the goodput.

Flat Goodput Curves. Further, one can observe that in all the presented results, the experimental goodput curve, as well as the curve for the goodput calculated using our model remains almost flat. The goodput decrease marginally as the ACKs due to reordered packets gets delayed. This can be explained with the help of Figure 9. We can see that for RTTs from $[0, \cdots, r)$, the congestion window increase with slope 1, and for RTTs from $[r, \cdots, r+bW/2)$, the congestion window increases with slope $1/b$. This means that if the value of RP is large as compare to r (the RTT-number in which ACKs due to delayed packets arrive), r will have very small impact on the overall goodput. This is the reason why the goodput curve remains almost flat when the value of r increase in our experiments.

V. Conclusion

We presented an analytical model that captures the goodput of TCP NewReno as a function of round-trip time, average time duration between packet-reorder events, average number of packets reordered during every reorder event, and the congestion window threshold of TCP NewReno. In the process of developing our model, we identified several interesting properties of TCP NewReno.

We validated our model by presenting extensive experimental results of data transfer between two physical hosts using TCP NewReno as the transport protocol. Our model fills an important gap in the literature by explicitly capturing detailed and accurate behavior of a TCP NewReno flow that experiences packet-reorder events from time to time.

References

[1] S. Floyd, T. Henderson, and A. Gurtov. The newreno modification to TCP's fast recovery algorithm. In *RFC 3782, Standards Track*, April 2004.

[2] D. Gillies. 1xEV-DO handoff trace analysis. *Qualcomm internal study report (slide presentation)*, April 7, 2007.

[3] Q. He, C. Dovrolis, and M. Ammar. On the predictability of large transfer TCP throughput. *Proceedings of the ACM SIGCOMM '05*, pages 145–156, 2005.

[4] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM Computer Communication Review*, 27(3):67–82, July 1997.

[5] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe. Modeling TCP throughput: a simple model and its empirical validation. *Proceedings of ACM SIGCOMM*, pages 303–314, 1998.

[6] P. Sarolahti and A. Kuznetsov. Congestion control in linux TCP. *Proceedings of USENIX'02*, 2002.

[7] E. Altman, K. Avrachenkov, and C. Barakat. A stochastic model of TCP/IP with stationary random losses. *Proceedings of ACM SIGCOMM*, pages 231–242, Stockholm, Sweden, August 2000.

[8] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. *Proceedings of IEEE INFOCOM*, pages 1742–1751, Tel-Aviv, Israel, March 2000.

[9] M. Goyal, R. Guerin, and R. Rajan. Predicting TCP throughput from non-invasive network sampling. *Proceedings of IEEE INFOCOM*, Hiroshima, Japan, March 2002.

[10] C. Samios and M. Vernon. Modeling the throughput of TCP vegas. *Proceedings of ACM SIGMETRICS*, San Diego, USA, June 2003.

[11] B. Sikdar, S. Kalyanaraman, and K. Vastola. Analytic models for the latency and steady-state throughput of TCP tahoe, reno and sack. *IEEE/ACM Transactions on Networking*, 11(6):959–971, December 2003.

[12] V. Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[13] J.C.R. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking*, 7(6):789–798, 1999.

[14] D. Loguinov and H. Radha. End-to-End Internet video trafc dynamics: statistical study and snalysis. *Proceedings of IEEE INFOCOM*, New York, June 23–27, 2002.

[15] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose and D. Towsley. Measurement and classication of out-of-sequence packets in tier-1 ip backbone. *IEEE/ACM Transactions on Networking*, 15(1):54–66, 2007.