Modeling the Goodput of TCP NewReno in Cellular Environments

Sushant Sharma[†‡], Donald W. Gillies[‡], Wu-chun Feng[†]
[‡] Corporate R&D, Qualcomm
San Diego, CA
[†] Synergy Lab, Virginia Tech
Blacksburg, VA

***Abstract*— In this paper, we present an analytical model that characterizes TCP NewReno's goodput as a function of round-trip time, average time duration between handoffs, average number of packets reordered during a handoff, and the congestion window threshold. In cellular networks, the effective packet-loss probability for a flow experiencing handoffs is not exactly equal to the physical-layer packet-loss probability; it also depends on the frequency of handoffs and the number of packets that may arrive out of order at the receiver due to handoffs.**

With the emergence of technologies such as WiMax and Ultra Mobile Broadband (UMB), understanding the effect of handoffs and packet reordering on the goodput of TCP becomes very important for the designers of next generation cellular networks. Existing TCP throughput models do not capture handoff effects explicitly. As a result, these models cannot be used to understand the precise effect of handoffs and the resultant packet reordering on TCP's goodput. In this paper, we present a model of TCP NewReno goodput that captures the effect of handoffs. We validate the model by performing actual file transfers between different hosts that are connected by a router which emulates the wireless environment with handoff events and packet reordering.

I. Introduction

In next-generation cellular networks, packet-loss rates due to the physical medium have been reduced to the values as low as 1 in 10^6 [**?**]. If there is any packet loss due to the physical medium, protocol layers below TCP (such as RLP or Radio Link Protocol) repair the loss before TCP notices a problem. In addition to this, the NewReno variant of TCP can recover from multiple packet losses and rarely times out. Despite all these improvements, *window-limited* [**?**] TCP flows in cellular networks are usually unable to reach the goodput levels equivalent to $(WindowLimit/AverageRTT)$. The reason for this is handoffs. Handoffs almost always result in some degree of packet reordering, which TCP infers as effective packet loss. The real-world traces in several cities for pedestrians and vehicular users have shown that in a typical ITU traffic mix, handoffs occur roughly every 6 seconds on average [**?**]. Next-generation cellular networks aim to provide seamless Internet connectivity to highly mobile devices carried by people (e.g. passengers in a moving train or on an aeroplane). This will result in the mobile devices doing regular handoffs between base stations. These handoffs may result in packets being reordered, as shown in Figure **??**. After a handoff, the sender can have a shorter path to the destination or may have improved channel conditions for packet transmission. As a result, the new packets may arrive at the receiver earlier than the old packets sent just before handoff. That is, the receiver may sometimes receive packets out of order.

Understanding the effect of handoffs on the goodput of an individual TCP flow is very important for those who are designing or will design the next-generation cellular networks for mobile users. Existing TCP models [**?**] [**?**] do not explicitly capture the effect of handoffs on the throughput, and therefore cannot be used by next-generation cellular network designers. Generally, if a TCP flow is window-limited, any packet loss will not be beacuse of congestion but because of the physical layer dropping the packet. *However, in cellular environments, where packets may get reordered during handoffs, TCP will assume packet loss even if the flow is window-limited and the physical layer and/or routers are not dropping any packets.* Hence, there is a need to explicitly understand TCP NewReno's behavior during handoffs and include it in a model that will explain the obtained goodput correctly. "Goodput" is defined as the number of unique packets delivered to the receiver in a given amount of time. We think that an end user is concerned only with the amount of time it will take to transfer unique packets and not the total amount of data that includes retransmissions.

This paper makes the following unique contributions:

- An analytical model to predict TCP NewReno's goodput in cellular environments. The model includes the effect of handoffs and packet reordering on a TCP NewReno flow in combination with fast recovery and fast retransmit mechanisms.
- Model validation by providing extensive emulation results of the data transfer beween various hosts in our lab. Hosts were connected via a router that emulated a cellular environment with handoffs and packet reordering.
- Experimental results that show a noticeable drop in NewReno's goodput even if *one* packet gets delayed (resulting in several packets arriving out of order) during a handoff. Our model explains this significant drop in the goodput.

This is a VT Computer Science Technical Report that was submitted for review on November 16, 2007 to ACM SIGMETRICS.

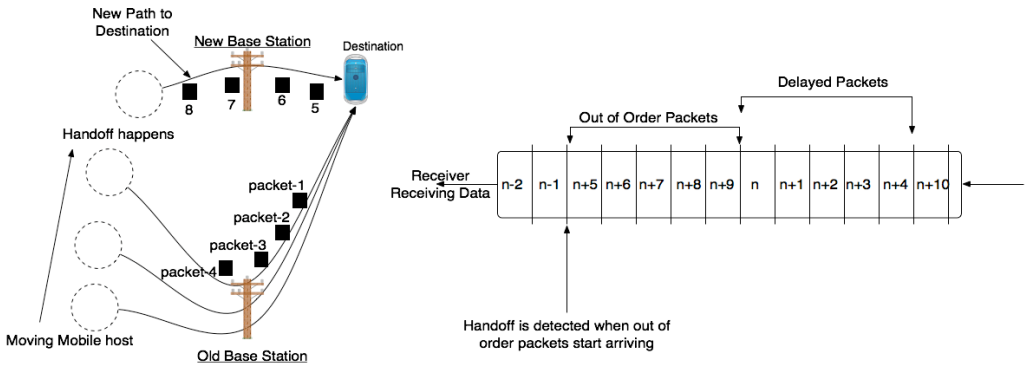


Fig. 1. **Handoff Description**

II. Background

This section explains our assumptions and description of the terms that we will be using throughout the paper. The operation of TCP NewReno is explained in RFC 3782 [**?**]. The RFC for TCP NewReno provides more than one alternative to respond to certain events. Here we explain the alternatives that must be considered for our modeling.

To being with, TCP NewReno keeps track of the number of sender packets sent but not acknowledged, using a variable called $FlightSize$. The initial state of the protocol is the slow-start state.

A. NewReno Behavior

- *Slow Start:* During slow-start state, the NewReno sender transmit two packets for every packet acknowledged by increasing its congestion window ($cwnd$) by one (i.e., exponentially) for every acknowledged packet. This increase goes on until the $cwnd$ becomes greater than or equal to Slow-Start-Threshold ($ssthresh$), or a packet loss in detected. After either event, the NewReno sender enters the congestion avoidance state state.
- *Congestion Avoidance:* During congestion avoidance, the size of $cwnd$ is increased by $1/W$ for every acknowledgment received (i.e. linearly), where W is the size of current $cwnd$ in terms of packets. When three duplicate ACKs are received, the sender enters the fast-retransmit state.
- *Fast Retransmit:* $FlightSize$ at any given time instant is the amount of data that is unacknowledged at that time. Upon entry, the lost packet (as indicated by the three duplicate ACKs) is retransmitted and $ssthresh$ is set to `max(FlightSize/2, 2*SMSS)`, where `SMSS` is the Sender's Maximum Segment Size. $cwnd$ is then reduced to $ssthresh + 3$, and the sender enters the fast-recovery state.
- *Fast Recovery:* In fast recovery, for every duplicate acknowledgment, $cwnd$ is increased by one and a new packet is transmitted, if allowed by the $cwnd$ value. We assume that in the fast-recovery state, the sender resets the retransmit timer upon receiving a partial acknowledgment. This is also known as Slow-But-Steady variant of NewReno. We also assume that while in fast-recovery, if a sender receives a full ACK, it will set the $cwnd$ to `min(ssthresh, Flightsize+SMSS)` and enters congestion avoidance. The sender responds to a timeout event by reducing the $cwnd$ to 1 and entering the slow-start state.

Further details about the operation of NewReno can be found in RFC 3782 [**?**].

III. NewReno's Behavior in Cellular Environments

We focus on modeling NewReno's goodput in cellular environments where handoffs are periodic events that result in packet reordering. To accomodate our model, we define a recovery period (RP) as the time at which transmitted packets are reorderd during a handoff to the time when packets are reordered again during the next handoff event. In other words, the RP is the period between two handoffs. We define an "old channel" as the channel between sender and receiver before the handoff happens. After the handoff happens, the new channel formed between sender and receiver is referred to as the "new channel." NewReno's behavior in the event of packet reordering can be divided into the following two cases:

- Case 1: Delayed Packets Are Lost. The packets that were sent before the handoff get delayed by a large amount of time and arrive *after* fast recovery has finished. Or, the packets that were sent before the handoff get lost due to bad channel conditions on the old channel. Both will result in the same amount of goodput drop.
- Case 2: Delayed Packets Arrive Prior To Fast-Recovery Completion: The delayed packets arrive at the receiver *before* fast recovery is finished. This will also result in goodput drop in NewReno but in a completely different manner than in the previous case.

We model each of these cases below.

A. Case 1: Delayed Packets Are Lost

Assume that when a handoff occurs, an average of the first d transmitted packets in a round-trip time (RTT) are lost

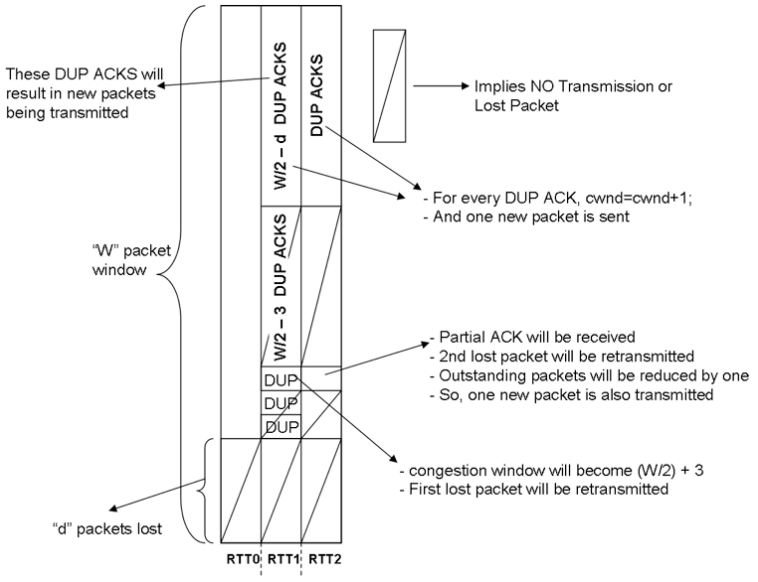


Fig. 2. **Sender side behavior of NewReno when d packets in a window of W packets are lost.**

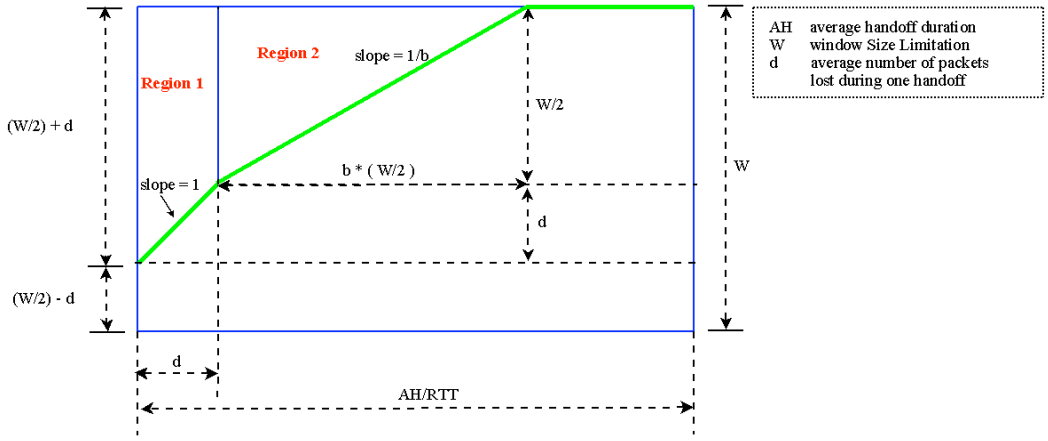


Fig. 3. **New packets sent by NewReno sender per RTT when d packets in a window of W packets are lost.**

due to bad channel conditions on the old channel. This is equivalent to the case where d packets arrive after the sender has finished its *fast-retransmit* phase. Figure **??** shows the sender-side behavior of NewReno, where W is the limit on the congestion window. This loss of d packets marks the beginning of an RP. In the next RTT, there will be no acknowledgements (ACKs) received by the sender for first d packets and then there will be three duplicate ACKs. At this point, NewReno enters fast retransmit and retransmits the sequence number requested in the duplicate ACK. It also sets $ssthresh$ to max($FlightSize/2$, 2).

Let us assume that the window is large enough so that $ssthresh = FlightSize/2 = W/2$. The $cwnd$ at this point is then reduced to $ssthresh + 3$, i.e. $cwnd = \left(\frac{W}{2} + 3\right)$. NewReno will then enter the fast-recovery state, and for every duplicate ACK received, $cwnd$ is increased by 1. Until the $cwnd$ value becomes W, which is the current $FlightSize$, the sender cannot send any new packets. This implies that for the next $\left(\frac{W}{2} - 3\right)$ duplicate ACKs, the sender will not send any new packets.

For the remaining $\left(\frac{W}{2} - d\right)$ duplicate ACKs that the sender is going to receive in that RTT, it will send *one* new packet per ACK. So, the sender will send $\left(\frac{W}{2} - d\right)$ new packets is the first RTT after handoff. In the next RTT, when it will receive a partial ACK for the retransmitted packet, it will retransmit the next unacknowledged packet. At that moment, the $FlightSize$ will be reduced by *one*, and the sender can inject another new packet into the network.

For the remaining $\left(\frac{W}{2} - d\right)$ duplicate ACKs, it will send *one* new packet per duplicate ACK. So, we can see that for

next d RTTs, the number of new packets transmitted will be incremented by *one* per RTT. After d RTTs, NewReno will come out of its fast-recovery state, and the new packets injected in the network will be incremented by $1/b$ per RTT, where b is the number of packets that every ACK is acknowledging.

The described scenario can again be divided into two cases, one in which NewReno can recover its window before the next handoff and the other in which the window cannot be recovered due to a shortf average handoff duration. The next two subsections model the goodput in both of these cases.

1) Recoverable Window: Figure **??** shows the growth of new packets sent in a single RTT after a handoff happens. We show the growth for the A_h amount of time, which is the average time duration between two handoffs. So, A_h/RTT will give us the number of RTTs between two handoffs. The resulting goodput between these two handoffs will be representative of the overall goodput. This new data transferred in A_h amount of time can be calculated by subtracting the area of *Region 1* and *Region 2* from the complete area of the rectangle. The area of *Region 1* is

$$ar1 = d\frac{W}{2} + \frac{d^2}{2} \tag{1}$$

The area of *Region 2* is

$$ar2 = b\frac{W^2}{8} \tag{2}$$

From the figure, the total area comes out to be $W\dfrac{A_h}{RTT}$. So, the total data transferred in A_h amount of time will be

$$W\frac{A_h}{RTT} - b\frac{W^2}{8} - d\frac{W}{2} - \frac{d^2}{2} \tag{3}$$

Hence, the obtained goodput will be

$$B_1(W) = \frac{W}{RTT} - \frac{1}{A_h}\left[b\frac{W^2}{8} + d\frac{W}{2} + \frac{d^2}{2}\right] \tag{4}$$

2) Non-Recoverable Window: Based on Figure **??**, NewReno will not be able to recover the congestion window if

$$\frac{A_h}{RTT} < d + b\frac{W}{2} \tag{5}$$

or by rewriting the above equation and calculating the value of d above which NewReno will not be able to recover its congestion window

$$d > \frac{A_h}{RTT} - b\frac{W}{2} \tag{6}$$

Based on this observation, we obtain the following lemma, which gives the number of RTTs required by NewReno to recover the congestion window when multiple packets from a window are lost.

Lemma 3.1: If all the d packets that are lost are not among the last three packets transmitted in an RTT, then the following statement will hold true. In an RTT, if d packets from a window of size W are lost, then it will take NewReno $\left[d + b\dfrac{W}{2}\right]$ number of RTTs to recover the window.

If the only packets that are lost are among the last *three* packets, then the receiver will not receive the triple duplicate acknowledgments in next RTT. As a result, it will not be able to reset the retransmit timer and might timeout. Because this is such an extremely rare case, we opt to leave it out of the analysis presented here. Figure **??** shows the growth in new packets sent per RTT when NewReno is unable to recover its congestion window. In this case, the starting size of the window will be different at the beginning of every subsequent RP. Figure **??** shows this behavior. We can calculate the number of RPs that are required for the congestion window to drop to the value $d+2$. We refer to the combination of all these RPs as an *unrecoverable recovery period* (URP). After a URP, the sender will timeout as it will not receive any triple duplicate ACKs and then will increase the congestion window according to $slowstart$.

Looking at Figure **??**, the equation for the new congestion window after the first RP is given by

$$W_n = \frac{W_{n-1}}{2} + \frac{1}{b}\left(\frac{A_h}{RTT} - d\right) \tag{7}$$

Solving the above recurrence relation gives us the value of W_n as

$$W_n = \frac{W_0}{2^n} + \frac{2}{b}\left(\frac{A_h}{RTT} - d\right)\left(1 - \frac{1}{2^{n-1}}\right); \forall n > 1, \tag{8}$$

where W_0 is the initial value of window before the beginning of first handoff in a given URP. W_1 can be calculated using equation **??**. And the value of n for which W_n comes out to be $p+2$ is

$$n = log_2\left(\frac{W_0 - \dfrac{4}{b}\left(\dfrac{A_h}{RTT} - d\right)}{d + 2 - \dfrac{2}{b}\left(\dfrac{A_h}{RTT} - d\right)}\right) \tag{9}$$

We can calculate the data transferred in first RP using Figure **??**. It is given by the sum of the area of $Regions\ 3, 4$ and 5 and can be calculated by the following equation for $n = 1$,

$$D_n = \frac{d^2}{2} + pC_n + \left(\frac{A_h}{RTT} - d\right)^2\left[\frac{\left(\dfrac{W_{n-1}}{2}\right)}{\left(\dfrac{A_h}{RTT} - d\right)} + \frac{1}{2b}\right], \tag{10}$$

where $C_n = \left(\dfrac{W_{n-1}}{2} - d\right)$.

Now, the goodput obtained during the n RPs after a handoff can be calculated by

$$B_2(W) = \frac{1}{nA_h}\Sigma_{i=1}^{n} D_i, \tag{11}$$

where D_i is calculated by equation **??**, n is given by equation **??** and W_i is given by equation **??**.

It can be inferred that in RP_{n+1}, NewReno may or may not recover its window. It depends on the value of A_h and the $ssthresh$ at that time. Value of $ssthresh$ at that time will be

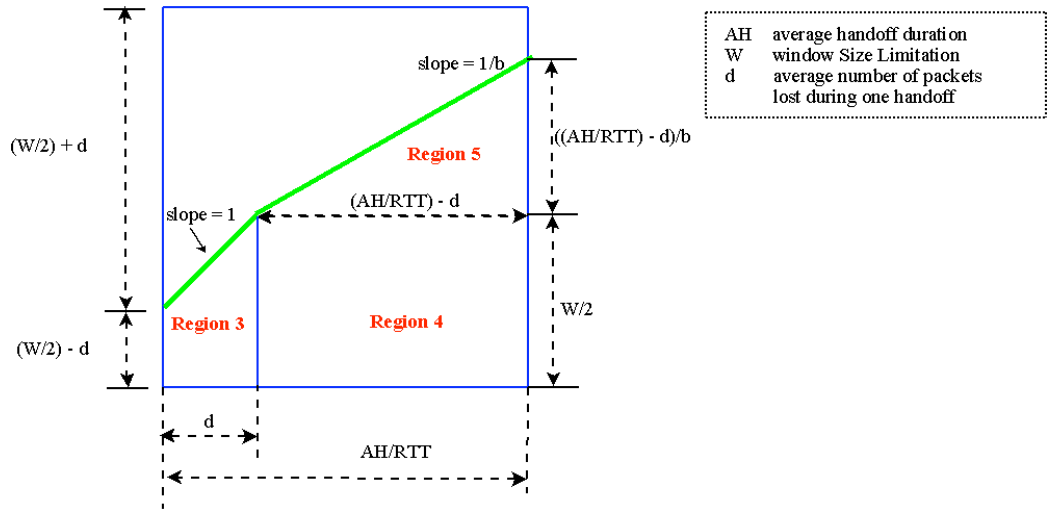


Fig. 4. **New packets sent by NewReno sender per RTT in first RP when d packets in a window of W packets are lost and NewReno is unable to recover the congestion window before next handoff.**

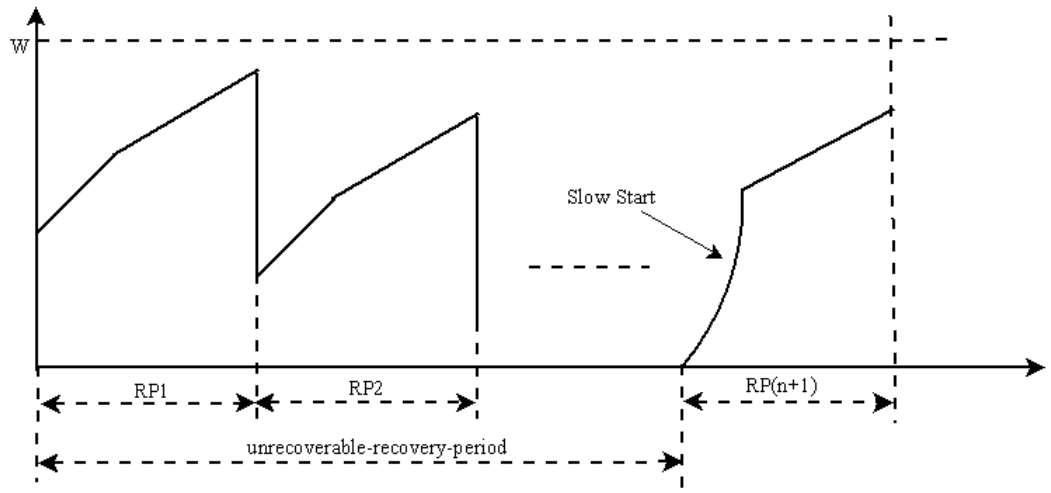


Fig. 5. **Long-term packet growth of NewReno sender when d packets in a window of W packets are lost and NewReno is unable to recover the congestion window before next handoff.**

$d/2$. And as shown in figure **??**, the size of window reached at the end of $(n+1)^{th}$ RTT, in terms of packets will be

$$\frac{d}{2} + \frac{1}{b}\left(\frac{A_h}{RTT} - log_2\frac{d}{2}\right), \tag{12}$$

For long-lived flows, we can consider (**??**) to be the value of W_0. However, for shorter-lived flows, this might be too small. During the begining of data transfer when TCP is searching for correct window size using *slow-start*, W_0 can go upto min($2^{(A_h/RTT)}$, W), where W is the limit on congestion window. To make the model more accurate, we can include the data transferred in the very first and $(n+1)^{th}$ RP also into account while calculating the goodput.

In this section, we have assumed that the first d packets are lost. It should not be difficult to see that the analysis will hold even if these packets are not the first ones in the window. We can treat the first packet lost as the beginning of window, and the same analysis will hold.

B. Case 2: Delayed Packets Arrive Prior To Fast-Recovery Completion

This section explains the NewReno behavior and presents the goodput model when the packets that get delayed after a handoff arrive before NewReno comes out of its *fast-recovery* state. Figure **??** shows the NewReno behavior in this case.

1) ACKs arrive in first RTT after handoff:: In first RTT after handoff, the sender will not receive ACKs for first p packets and then it will receive *three* duplicate ACKs. It will then retransmit the packet that it thinks is lost. It will also reduce its $ssthresh$ to $W/2$ and $cwnd$ to $[(W/2)+3]$. Now, for every duplicate ACK received, the sender will increase $cwnd$ by 1. After the ACK due to first delayed packet is received, it is

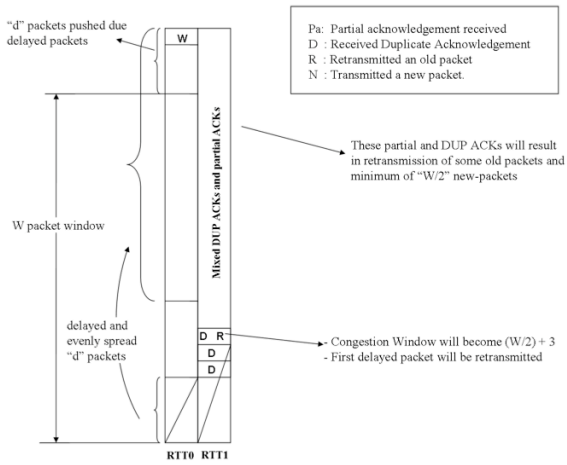
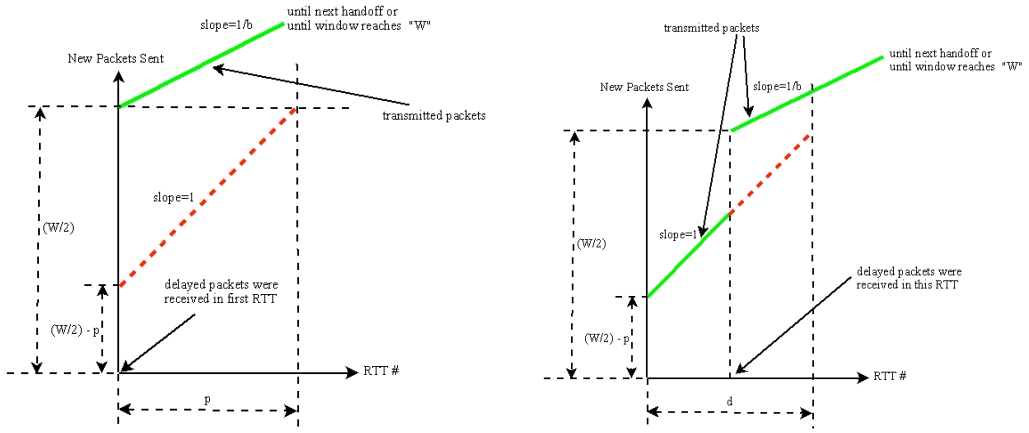


Fig. 7. **NewReno behavior when acknowledgments due to delayed packets arrive in first RTT after handoff.**



(a) **Delayed packets arrive in first RTT after handoff.**

(b) **Delayed packets arrive in rth RTT after handoff (r<d).**

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

going to receive the partial ACKs (due to delayed packets) and duplicate ACKs mixed with each other. If we assume that the partial ACKs due to these delayed packets will arrive in regular intervals, then we can say that in the remaining $(W-3)$ ACKs that the sender will receive, every $\left(\frac{W-3}{d}-1\right)$ duplicate ACKs will be followed by *one* partial acknowledgment.

Every duplicate ACK will increase the congestion window value by 1 and every partial ACK will reduce the outstanding packets by $\frac{W-3}{d}$. So, after calculations, the number of ACKs (duplicate plus partial) after which the sender can start transmitting new packets will be $\frac{W^2-9W+18}{4W-12-2d}$. Denote this by L.

So, after L ACKs, the value of the congestion window will become greater than the number of outstanding packets. Intuitively, one can see that higher the number of partial ACKs, the lesser will be the total ACKs that the sender will see before it can start sending new packets. This is because partial ACKs reduce the outstanding packets by more than one, whereas duplicate ACKs increase the congestion window by exactly one. But, as the number of packets that get delayed increase, the reduction in window size due to partial ACKs will become low. So, we can say that delaying 10 packets in a window of 30 packets will cause sender to send more new packets in the first RTT as compared to the case when only 1 packet was delayed. But, delaying 15 packets in a window of 30 packets will cause the sender to send *less* new packets in the first RTT as compared to the case when 10 packets were delayed. This result can be stated in a lemma

Lemma 3.2: If all the delayed packets due to a handoff are delivered before the end of RTT, then the following will be true for the first RTT after handoff. The number of packets transmitted by the sender when exactly one packet gets delayed will be less than the number of new packets transmitted when more than one packet gets delayed.

However, in this case, at the end of first RTT after handoff, when the sender will exit the fast-recovery state, the value of

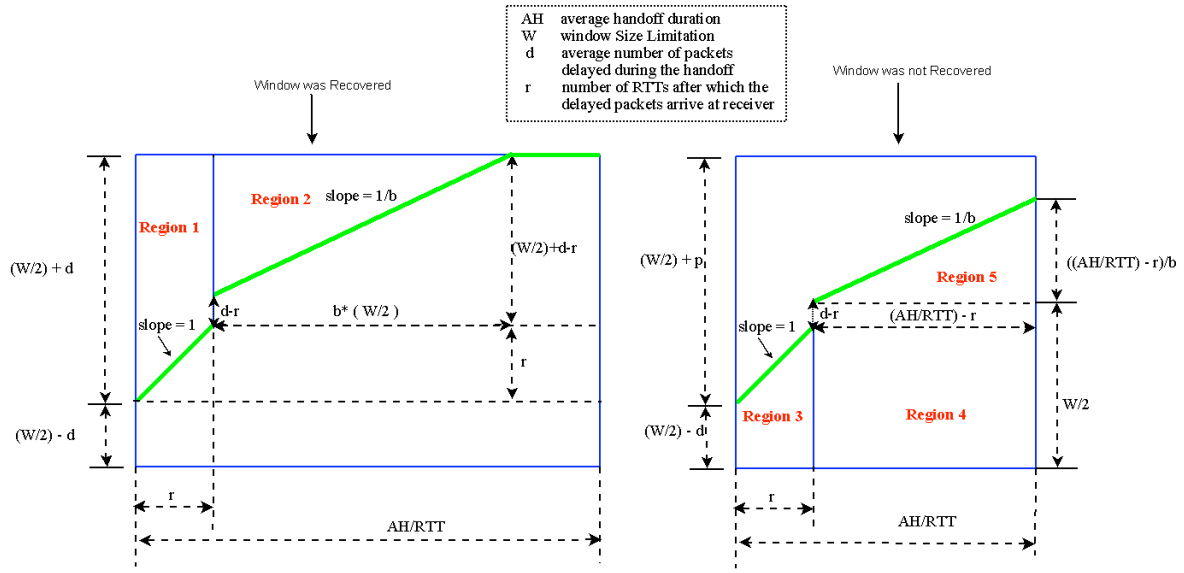


Fig. 9. **New Packets sent by NewReno in first RP after "d" packets were *delayed* in a window of "W" packets.**

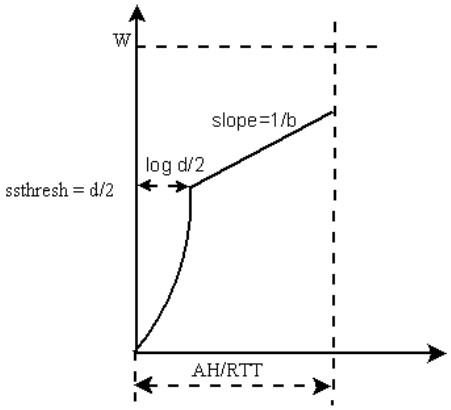


Fig. 6. **Packet growth in $(n+1)^{th}$ RP**

congestion window will become min($flightsize$, $ssthresh$) which will be $ssthresh$ (i.e. $W/2$).

Another interesting observation is that it is possible that a NewReno sender may end up retransmitting up to $(W-3)$ old packet in the first RTT after handoff. This can be explained as follows. In case only one packet got delayed and it arrives as the fourth packet. Then the ACKs due to remaining packets in this RTT will be considered as partial ACKs, because they do not cover the last packet transmitted before sender entered *fast-retransmit* state. As a result, every partial ACK will result in retransmission of next unacknowledged packet. These retransmissions can be avoided by the use of SACK option in the TCP. However, SACK will still not affect the goodput obtained in this case.

2) ACKs do not arrive in first RTT after handoff:: On the other hand, if the ACKs due to "d" delayed packets arrive in an RTT which is not the first one after the handoff, the retransmissions of old packets will not happen even in the absence of selective acknowledgement (SACK).

Until the RTT in which the ACKs due to the delayed packets arrive, the number of new packets sent will increase by *one* per RTT, according to the analysis presented in previous sections. As soon as all the ACKs due to delayed packets arrive, NewReno will come out of *fast recovery* and then the increase in the number of packets per RTT will be $1/b$. The behavior is shown in Figure **??** and Figure **??**.

We are assuming that the duplicate ACKs generated due to the arrival of delayed and retransmitted packets will not result in window reduction because these duplicate ACKs will be mixed with the new ACKs from new packets and three duplicate ACKs will not arrive in sequence. The enhancements to TCP, such as SACK, can also remove the possibility of window reduction in this scenario. If we consider that the ACKs due to the delayed packets are arriving in the r^{th} RTT after handoff, the resulting goodput can be calculated by looking at Figure **??**. The analysis is exactly same as the one in Sections **??** and **??**.

3) Summary of Goodput Modeling: The final values of goodput can be summarized as follows:

- Recoverable Window:

$$\hat{B}_1(W) = \frac{W}{RTT} - \frac{1}{A_h}\left[\frac{b}{8}W^2 + rU + \frac{r^2}{2}\right], \quad (13)$$

  where $U = \left(\frac{W}{2} + max(0, d-r)\right)$.
- Non-Recoverable Window: It is interesting to see that it is the number of RTTs by which packets are delayed determine if the window can be recovered. Whether window can be recovered or not is independent of the number of packets delayed (d). If following equation is

true window can not be recovered.

$$r > \frac{A_h}{RTT} - b\frac{W}{2} \tag{14}$$

Value of window after n^{th} RTT can be given as

$$\hat{W}_n = \frac{\hat{W}_{n-1}}{2} + \frac{1}{b}\left(\frac{A_h}{RTT} - r\right) \tag{15}$$

or

$$\hat{W}_n = \frac{\hat{W}_0}{2^n} + \frac{2}{b}\left(\frac{A_h}{RTT} - r\right)\left(1 - \frac{1}{2^{n-1}}\right); \forall n > 1, \tag{16}$$

where $0 \leq r \leq d$ and $\hat{W}_1$ can be calculated using equation **??**.

The value of $\hat{n}$ for which $\hat{W}_n$ comes out to be $d+2$ is

$$\hat{n} = log_2\left(\frac{\hat{W}_0 - \frac{4}{b}\left(\frac{A_h}{RTT} - r\right)}{r + 2 - \frac{2}{b}\left(\frac{A_h}{RTT} - r\right)}\right) \tag{17}$$

Here we are again assuming that if d out of $(d+2)$ packets are delayed, sender will not receive 3 duplicate ACKs so as to reset its retransmit timer and as a result, will time out.

Total data transferred in n^{th} RP will be

$$\hat{D}_n = \frac{r^2}{2} + r\hat{C}_n + \hat{Q}\left(\hat{W}_{n-1} + \frac{\hat{Q}}{b}\right) \tag{18}$$

where $\hat{C}_n = \left(\frac{\hat{W}_{n-1}}{2} - d\right)$, $\hat{Q} = \frac{1}{2}\left(\frac{A_H}{RTT} - r\right)$

The total goodput when ACKs for delayed packets come before end of fast-recovery and the window can not be recovered will be

$$\hat{B}_2(W) = \frac{1}{\hat{n}A_h}\Sigma_{i=1}^{\hat{n}}\hat{D}_i, \tag{19}$$

$\hat{W}_0$ will be same as W_0. For long lasting flows, it can be given as

$$\hat{W}_0 = \frac{d}{2} + \frac{1}{b}\left(\frac{A_h}{RTT} - log_2\frac{d}{2}\right) \tag{20}$$

and for shorter flows

$$\hat{W}_0 = min(2^{(A_h/RTT)}, W), \tag{21}$$

where W is the limit on congestion window.

4) Important Observations: **1.** Equation **??** can also be used as a general purpose model to predict goodput of a TCP NewReno flow on any network where goodput is congestion limited. For long lasting flows, the equation is independent of W which was assumed as the window threshold that can not be reached. So, equation **??** can be used to predict NewReno goodput on a network where we can calculate the average duration between two "*drop-events*" and average number of packets lost in one event. r will be equal to d in this case.

2. With SACK, the sender can come out of fast recovery in the 3^{rd} RTT after the handoff. This is because, it will be able to detect the hole and retransmit the packets with missing sequence numbers in 2^{nd} RTT, thereby receiving the complete ACK in 3^{rd} RTT. Hence, r will be equal to 3 in case of SACK.

3. If d is greater than $W/2$, i.e. if a flow is loosing/reordering more than half of its window during every loss event, then there will be $\left(d - \frac{W}{2}\right)$ RTTs following the handoff (or loss event) during which sender will not be sending any *new* data. After these $\left(d - \frac{W}{2}\right)$ RTTs the number of *new* packets sent per RTT will grow with slope 1 for $r - \left(d - \frac{W}{2}\right)$ RTTs and then with slope $1/b$ for remaining $[(A_h/RTT) - r]$ RTTs.

IV. Data Transfer Results and Model Validation

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

A. Experimental Setup

The experimental setup within our lab is shown in figure **??**. We have one sender and one receiver, both running linux kernel 2.6.21.

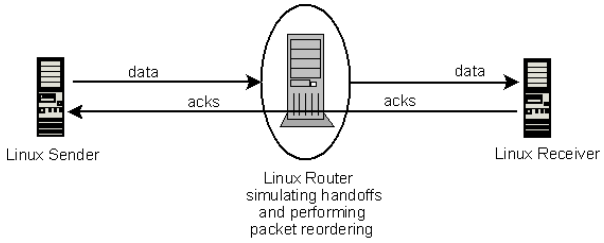


Fig. 10. **Experimental setup within our lab.**

Inbetween these two machines is a Linux router running our *handoff-emulator (H-E)*. *H-E* was written by us so that we can emulate handoffs and packet-reordering. *H-E* can also be used to limit the bandwidth between the sender and receiver.

B. Measurements while Emulating Handoffs

This subsection presents the measurement data collected while transferring 50 MB files between two hosts that were connected through the *H-E*. We used *H-E* to emulate diffferent values of RTT, different inter handoff durations (A_h) and different number of packets getting reordered (d). Several different limits on window size (W) were obtained at the end-hosts by using TCP system control (sysctl) variables within the linux kernel. We collected data under different combinations of above values in order to validate the models presented in (**??**) and (**??**).

Figures **??** to **??** presents the comparision of goodput calculated using (**??**) or (**??**), as applicable, and the actual goodput achieved when only 1 packet gets delayed by a certain amount of time. x-axis shows the amount of time by which the packet get delayed and y-axis show the achieved goodput. Window limit was kept constant at 193 KB in all these three cases and RTT was increased from 70 to 170 msecs. Time duration between two handoff events was also kept constant at 2secs for all these runs. It is important to note here the difference between the actual goodput and the goodput calculated using our model is quite large when the delayed packet was received at the receiver before the RTT finishes. This answer to this lies in the implementation details of TCP in the Linux kernel. It is documented in [**?**] that the Linux fast recovery do not completely follow the behavior given in RFC 3782. As a result, when the reordered packets arrive before the end of the RTT, Linux TCP implementation is overly agressive in increasing its congestion window size. In order to confirm that, we did the following. After the actual data transfer was complete, we counted the total triple duplicate ACKs that the sender received and the total packets that it had sent and calculated the packet drop probability. When this packet drop probability was used in the model presented in [**?**], the throughput given by that model confirmed the Linux TCP's agressive behavior. Figure **??** shows that when reordered packets arrive before the end of RTT, similar to our model, throughput calculated using the model in [**?**] is much lower than the actual values of goodput obtained.

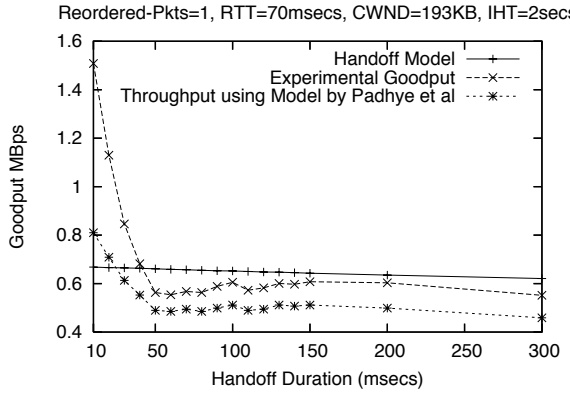


Fig. 11. **Figure confirming the aggressive increase in the Linux TCP window when reordered packets arrive before the end of RTT.**

It should be noted here that the maximum achievable throughput in any of the experiments can not go beyond 2.33Mbps because our handoff-emulator is restricting the throughput to that value. So, even if we have a window limit of 1024KB and a RTT of 100msecs between sender and receiver, the maximum achievable goodput will still be 2.33Mbps and not 10.24Mbps.

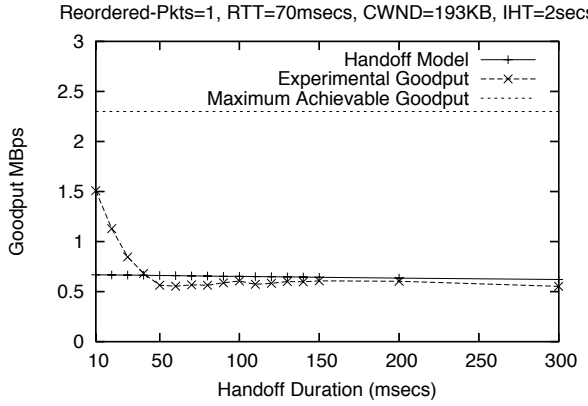
Figures **??** to **??** shows the comparison when RTT, Window limit and time durations between two handoffs was kept constant at 100msecs, 193Kbps and 2seconds respectively in all the cases but the number of packets that get delayed were increased from 5 to 15. These results show that delaying of one packet is sufficient to drop the goodput significantly and the number of subsequent packets that get delayed do not drop the goodput to a significantly much lower value.

Figures **??** to **??** shows the results when the RTT, the number of delayed packets and the time duration between two handoffs (IHT) was kept constant at 100msecs, 10 and 4seconds respectively, but the Window limit was increased from 64KB to 1024KB. These results show that the increase in window can result in inproving the goodput but this increase is very minor if a flow is experiencing regular handoffs that are resulting in packets being reordered.

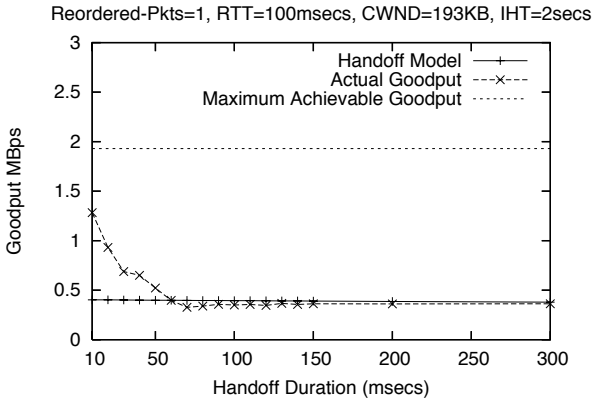
Again, looking at figures **??** to **??**, it is evident that increasing in the time duration between handoffs is not affecting the goodput as much. In these experiments, we have kept everything constant but changed the time duration between two handoff events. It shows that packet reordering has already caused the goodput to drop to a significantly low value and increasing the time duration between two handoffs will not improve the goodput much.

V. Conclusion

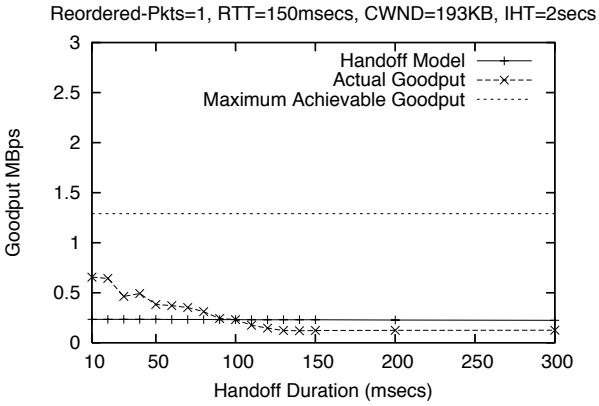
We have presented an analytical model to determine TCP NewReno's goodput in cellular networks, where user mobility can result in a flow experiencing regular handoffs between base stations. The model explains the significant drop in the goodput of TCP even when the network and intermediate routers are not dropping any packets. The model can be used to study the direct impact of packet reordering on the TCP's goodput. This model can have significant impact on the way in which network designers design next-generation cellular networks aimed at highly mobile users.



(a) RTT=70 msecs

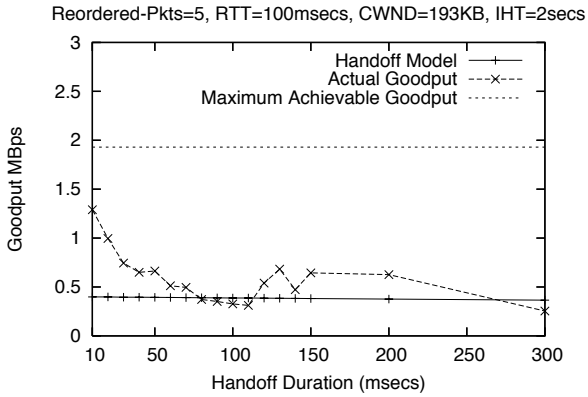


(b) RTT=100 msecs

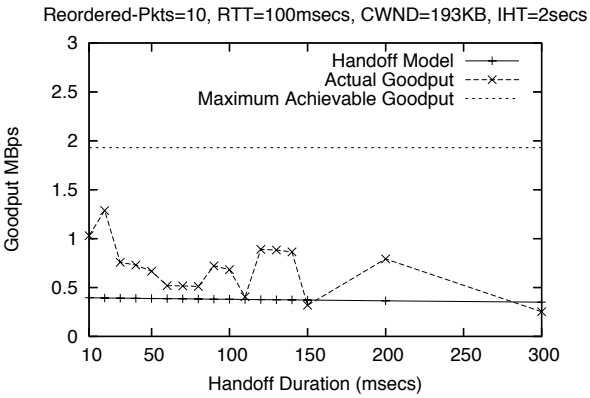


(c) RTT=150 msecs

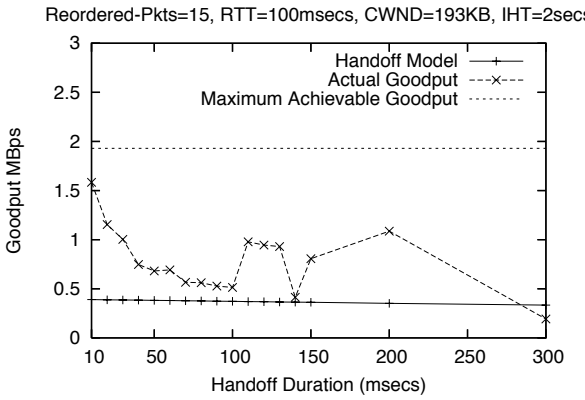
Fig. 12. Comparison of actual goodput and the goodput calculated using our model at different RTTs.



(a) 5 Packet Reordering

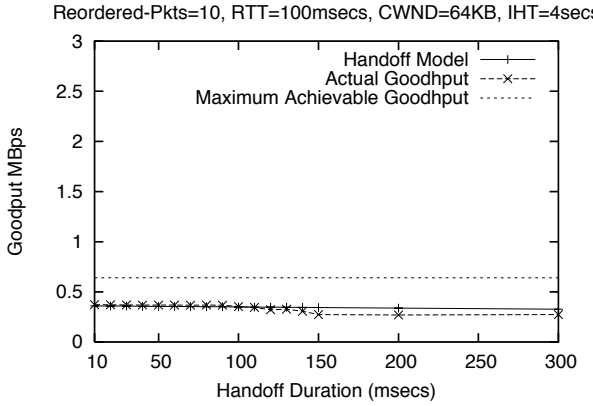


(b) 10 Packet Reordering

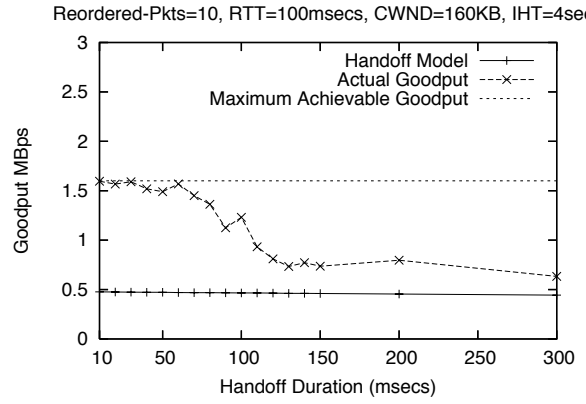


(c) 15 Packet Reordering

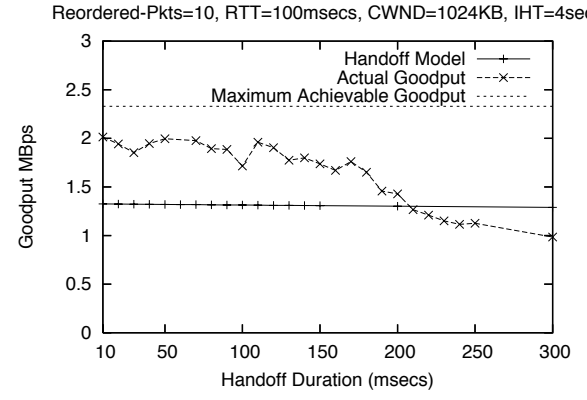
Fig. 13. Comparison of actual goodput and the goodput calculated using our model for different number of delayed packets.



(a) 64K Window Limit

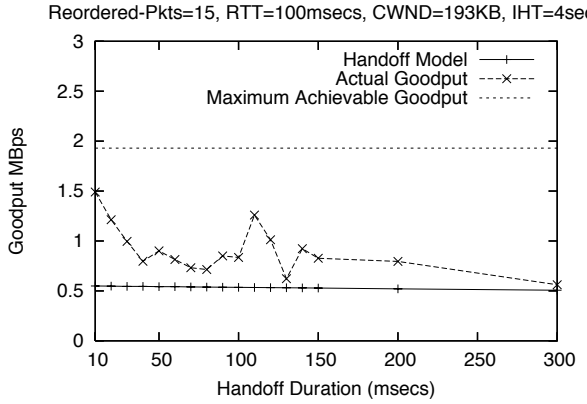


(b) 160K Window Limit

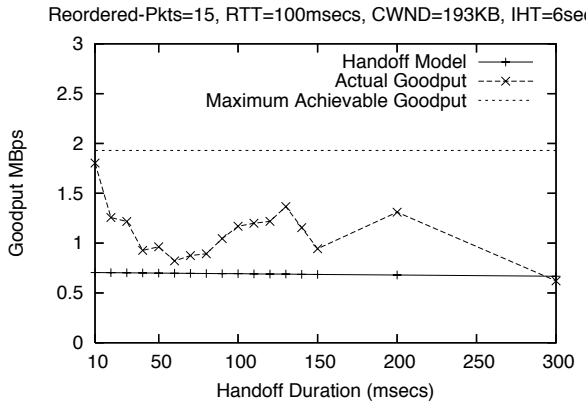


(c) 1024K window Limit

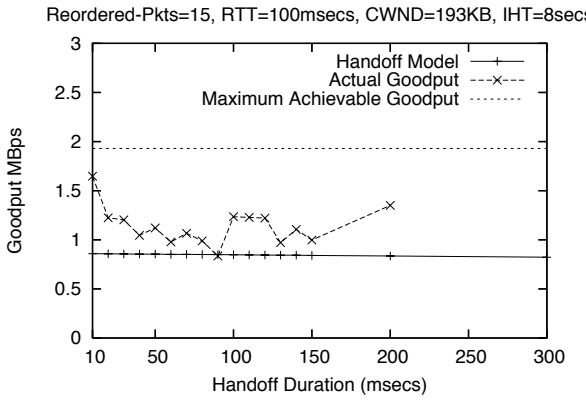
Fig. 14. Comparison of actual goodput and the goodput calculated using our model for window limited flows.



(a) 4 second time between two handoffs



(b) 6 second time between two handoffs



(c) 8 second time between two handoffs

Fig. 15. Comparison of actual goodput and the goodput calculated using our model for different handoff frequencies.