Dynamics of Continuous, Discrete and Impulsive Systems
Series **B:** *Applications & Algorithms* **13 (2006)** 77-94

TCP Performance with Segment-in-Flight Estimation Algorithm over Wireless Links

Jianping Pan, Jon W. Mark, and Xuemin Shen

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Abstract. TCP performs reasonably well over the Internet where packet losses are mainly due to network congestion. However, TCP suffers significant throughput degradation over hybrid wireless/IP networks where packet losses are also due to transmission errors in wireless segments and during mobile host handovers. In this paper, the micro-scale behavior and packet-level performance of four popular TCP variants over wireless links are assessed, and a heuristic *Segment-in-Flight Estimation algorithm* for TCP senders is proposed and evaluated. Extensive simulations confirm that the enhanced TCP is more robust against packet losses, can achieve better end-to-end performance, and still keeps the fairness and compatibility with ordinary TCP variants.
Keywords. Internet, TCP/IP, Protocol performance, Wireless communications, Network simulation.

1 Introduction

The integration of the Internet and wireless networks is expected to offer multimedia services to mobile and fixed users from anywhere at anytime in the near future. There are many different interworking strategies to connect voice-oriented cellular systems and other wireless segments to IP-based data networks [1]. Moreover, the next generation cellular systems are also IP-centric. However, an essential issue here is how to support various existing Internet applications, *e.g.*, web surfing and packetized audio/video, running properly and efficiently over hybrid wireless/wired networks with prescribed end-to-end QoS provisioning.

The performance degradation of TCP over wireless links has received much attention recently. Various approaches in different protocol layers to mitigate this deficiency have been proposed, and they are compared and surveyed in [3, 4]. Data link layer approaches try to transparently enhance the quality of wireless links by Forward Error Correction (FEC), local acknowledgment, and selective retransmissions. However, the end-to-end TCP control logic might be interfered by the complicated lower layer control mechanisms, which may result duplicate retransmissions in multiple layers or rapid changes in packet transit time [6]. Transport layer approaches, which might adopt implicit snooping or explicit notification, or split connection with or without end-to-end semantics, try to help TCP endpoints not to mistakenly react to packet losses due to transmission errors as those due to network congestion. It should be mentioned that the interoperability between these approaches and the original protocols need further investigation.

The strategy adopted in this paper is different from those aforementioned and other approaches discussed in [4] where brand new transport protocols are proposed.

This paper investigates the performance and behaviors of existing algorithms embedded in popular TCP variants over generalized wireless links, and proposes a heuristic algorithm for TCP senders to improve the end-to-end performance. This strategy is taken here in recognition that today there exists a large number of TCP/IP installations and TCP/IP-enabled applications over the global Internet. Therefore, it is much easier for an enhanced TCP sender which is compatible with and fair to ordinary TCP variants to be adopted and deployed incrementally.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of four popular TCP variants and their embedded algorithms of interest, as well as an analysis comparing the recovery cost of two loss detection schemes used in these algorithms. Section 3 investigates the performance degradation of these TCP variants over wireless links, and the deficiency is explained by some unexpected TCP endpoint behaviors. An enhancement to TCP senders by incorporating a heuristic Segment-in-Flight Estimation algorithm to recapture the capacity wasted by ordinary TCP variants is proposed and evaluated in Section 4. The description of the algorithm, simulation comparisons, and fairness validation of the enhanced TCP are also presented. Concluding remarks are given in Section 6.

2 TCP Congestion Control Algorithms

This section first outlines the principle of congestion control in which building blocks for various TCP algorithms and variants are discussed. These algorithms are designed to combat network congestion, but also can be triggered by transmission errors. Two practical schemes to detect and recover packet losses are then compared analytically in terms of achievable throughput.

2.1 Congestion Control Principles

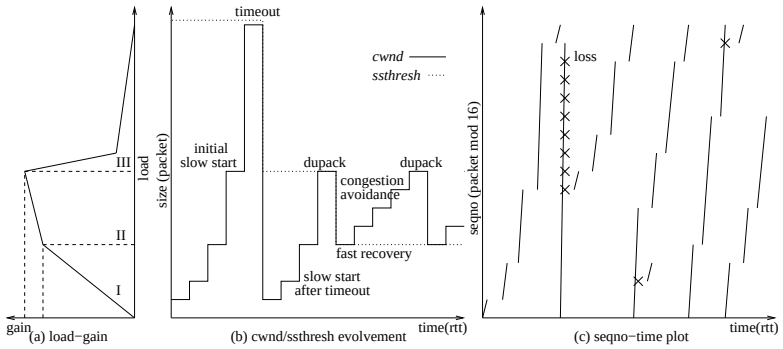

Figure 1: TCP Congestion Algorithms

In adaptive dynamic control, TCP senders, which assume the underlying network and receivers are passive and routers only drop packets when overloaded, are designed to react to packet losses due to network congestion. Figure 1(a) shows the relation between *offered load* and *performance gain*, where a larger throughput with a shorter delay gives a higher gain. When a network is slightly loaded in region I, an increase in load sees an obvious increase in gain. When a network is almost fully loaded or in its equilibrium state in region II, the same load increase only has a slight increase in gain. An overloaded network in region III then unavoidably encounters congestion collapse with increased packet losses, extremely low throughput, and unacceptable long delay.

The illustrated congestion collapse was actually observed in the 1980's, and it has motivated the design and incorporation of TCP congestion control algorithms since then. Ideally, it is expected that a network should be operated near the adjacent areas of region I and II, where load and gain are both optimized. However, due to the unpredictable dynamics in network resources and user requirements, most operating networks usually oscillate between regions I and II, and sometimes even in region III. TCP senders treat acknowledgment (*ack*) returned from receivers on a packet reception as an input signal. Since a packet and its *ack* travel a round between sender and receiver, a received *ack* indicates a packet has already left the network, so a new packet can be sent. The spacing between consecutive *ack*s also help the TCP sender, as a *producer*, to cope with the rate that the bottleneck link, as a *consumer*, can afford. This packet conservation law is also known as "*ack* self-clocking".

The absence of *ack* for a particular packet is interpreted as an indication of packet loss. TCP congestion control assumes that packet losses are caused by network congestion. According to Fig. 1(a), TCP senders consider that the system is approaching or already in region III, so drastic actions have to be taken to bring it back to region I or II. These actions are known as TCP congestion control algorithms which will be overviewed in the following subsection.

2.2 Algorithms and Variants

TCP has evolved into many incremental variants in the last two decades, mostly driven by the design and incorporation of new congestion control algorithms. Currently, **Reno** and **NewReno** are two mainstream TCP variants, while **Tahoe** is an old but still existing variant. TCP **SACK** is an emerging variant and now becomes more and more widely supported. Besides these four popular TCP variants, there are other experimental algorithms proposed in the literature.

TCP **Tahoe** features the **Slow-Start**, **Congestion Avoidance**, and **Fast Retransmit** phases [7]. When a TCP connection is established, the sender initializes its congestion window (*cwnd*) to one Maximum Segment Size (MSS), and sets the Slow-Start threshold (*ssthresh*) to the advertised receiver window (*rwnd*). The receiver normally returns *ack* accumulatively every other packet with the delayed *ack* (*delack*) policy. If *cwnd* is below *ssthresh*, the Slow-Start algorithm increases *cwnd* exponentially by one MSS for every new *ack*; otherwise the Congestion Avoidance algorithm increases *cwnd* linearly by $\frac{\text{MSS}*\text{MSS}}{cwnd}$ for every new *ack*, as Fig. 1(b) shows.

The sender samples round-trip time (*rtt*) once per sender window (*swnd*, the minimum of *cwnd* and *rwnd*) and calculates the smoothed *rtt* (*srtt*) and its deviation (*rttv*). A timer associated with a packet in a *swnd* is armed with an *rtt* timeout (*rto*) value derived from *srtt* and *rttv*, when this packet is sent. The reception of an *ack* for this packet cancels the timer, and a new timer for another packet will be set up. If a timeout occurs, the oldest unacknowledged packet is resent, while *ssthresh* is throttled to half of the current *cwnd*, and *cwnd* is reset to its initial value. When a sender gets three duplicate *ack*s (*dupack*s), it assumes that the oldest unacknowledged packet is lost, and *fast-retransmit*s that packet, while *ssthresh* and *cwnd* are reassigned as a timeout occurs in Tahoe.

Based on the algorithms in Tahoe, TCP **Reno** further introduces a **Fast Recovery** algorithm that adjusts *cwnd* more optimistically. When invoking Fast Retransmit on the third *dupack* (triple-*dupack*), Reno sets both *ssthresh* and *cwnd* to half of its current *cwnd*. Therefore, Congestion Avoidance is followed after a triple-*dupack* in Reno instead of Slow Start in Tahoe. Fig. 1(b) shows the evolution of *cwnd* and *ssthresh* for Reno. Also, as the *ack self-clocking* principle implies, a *dupack* indicates that a packet has left the network. Thus, during Fast Recovery, TCP can temporarily **Inflate** *Cwnd* when senders receive *dupack*s. When a new *ack* (*newack*) comes, TCP exits from Fast Recovery, and deflates *cwnd* conservatively.

However, neither Tahoe nor Reno in most cases has the ability to recover multiple packet losses in one *swnd* without relying on a timeout, as shown in Fig. 1(c), since there are insufficient *dupack*s to identify the next lost packet. As analyzed in the next subsection, timeout costs much more than *dupack* to recover a lost packet, and it also reduces *cwnd* to the initial value, which severely degrades the achievable throughput in the next few *rtt*s. TCP **NewReno** [13] improves Reno with an additional **Partial Acknowledgment** (*pack*) algorithm. When TCP enters the Fast Recovery phase, it records the highest sequence number (*hiseq*) it has ever sent. If a *newack* arrives but does not cover *hiseq*, TCP evaluates it as a *pack* and assumes that there are more packets lost in the same *swnd*. NewReno conservatively retransmits the oldest unacknowledged packet on a *pack* for every *rtt*.

TCP **SACK** [12] enables TCP receivers to return a **Selective Acknowledgment** option which also acknowledges received but nonconsecutive packets, since the ordinary *ack* only acknowledges the latest consecutive packets. Due to the header length constraint, TCP receivers can return up to three pairs of sequence numbers in the SACK option. The first pair specifies the consecutive data block which contains the packet that triggers this *ack*, and the other two just repeat the first two pairs in the previous SACK-ed *ack*. This redundant design reduces the risk if the previous *ack* is lost. With the sending history it maintains, an SACK-capable TCP sender then has the capability to determine exactly which packets are received and which are indeed lost. SACK is applicable to all previous TCP variants and is able to retransmit all known lost packets in the next *rtt* if *cwnd* permits. An SACK-capable sender can skip the SACKed packets during retransmission, unless a timeout occurs.

Commercial TCP implementations normally conform to one of these four variants, with a set of built-in congestion control algorithms varying from the original Slow-Start to the most promising Selective Acknowledgment. No matter how

these algorithms react, any packet losses, signaled by timeout or triple-*dupack*, are assumed to be an indication of network congestion. Extensive reactions such as Additive Increase and Multiplicative Decrease (AIMD) to regulate *cwnd* and exponential backoff for *rto* are taken to avoid congestion collapse. These algorithms work reasonably well when the Internet is scaled from a closed research platform to a global information infrastructure as far as the assumption on packet losses remains valid in most situations. However, with the popularization of emerging link layer technologies that have different characteristics, *e.g.*, Geostationary-Earth-Orbit or Lower-Earth-Orbit satellites, varietal Digital Subscriber Lines and Cable Modems, and Radio-Frequency or Infrared wireless links, the consistency and efficiency of these algorithms deserve a revisit. Moreover, large bandwidth-delay product [8], asymmetric [10], and unidirectional links also have impacts on TCP performance.

2.3 Timeout vs. Triple-*dupack*

Besides timeout and triple-*dupack* based loss detection schemes, Explicit Congestion Notification (ECN) is recently proposed to notify TCP senders in a proactive manner even before network congestion and packet losses actually occur. However, ECN requires the support from Active Queue Management (AQM) capable routers, which has not been widely deployed yet. More importantly, ECN alone is not a fully reliable scheme since the notification can also get lost during severe congestion. Therefore, triple-*dupack* and timeout are still the second and final guards for TCP senders to detect and recover packet losses.

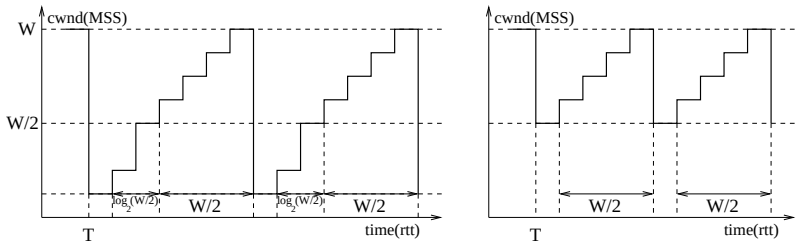

Figure 2: Timeout v.s. Triple-*dupack*

Here we consider a simplified scenario where only timeout or triple-*dupack* occurs. Suppose at time T when $cwnd = W * \text{MSS}$, a packet is lost. During $[T, T + rtt]$, this packet is retransmitted, and $cwnd$ is either throttled to one MSS if triggered by timeout or to $\frac{W*\text{MSS}}{2}$ if by triple-*dupack*. In both cases, $ssthresh = \frac{W*\text{MSS}}{2}$. Subsequently, $cwnd$ evolves differently in these two cases. It takes approximately $\log_2 \frac{W}{2} * rtt$ for $cwnd$ to reach $ssthresh$ in the timeout case, when $cwnd$ is increased exponentially by a factor of 2 if the receiver returns an *ack* for every received packet [1]. Then in the Congestion Avoidance phase it takes approximately

[1]Or a factor of 1.5 if *delack* is enabled at TCP receivers. Here we consider a factor of 2 to give the

$\frac{W*rtt}{2}$ for $cwnd$ to reach $W *$ MSS, when another packet loss occurs. The number of packets pumped into the network in the Slow Start phase is

$$2 + 2^2 + ... + 2^{\log_2 \frac{W}{2}} = \frac{2(1 - 2^{\log_2 \frac{W}{2}})}{1 - 2} = 2(\frac{W}{2} - 1) = W - 2;$$

in the Congestion Avoidance phase, the number is

$$\frac{W}{2}\frac{W}{2} + \frac{W}{2}\frac{W}{2}\frac{1}{2} = \frac{3W^2}{8}.$$

Therefore, the average throughput for these two phases is

$$\text{Th}_{\text{timeout}} = \frac{W - 2 + \frac{3W^2}{8}}{\log_2 \frac{W}{2} + \frac{W}{2}} \frac{\text{MSS}}{rtt}.$$

For the triple-*dupack* case, it always takes $cwnd$ about $\frac{W*rtt}{2}$ to encounter another packet loss, and the number of packets pumped into network during this period is $\frac{3W^2}{8}$. Therefore, the average throughput for this case is

$$\text{Th}_{dupack} = \frac{3W}{4} \frac{\text{MSS}}{rtt}.$$

To compare $\text{Th}_{timeout}$ and Th_{dupack}, define

$$\text{Th}_{\text{diff}} = \text{Th}_{\text{timeout}} - \text{Th}_{dupack} = (\frac{W - 2 + \frac{3W^2}{8}}{\log_2 W - 1 + \frac{W}{2}} - \frac{3W}{4}) \frac{\text{MSS}}{rtt},$$

or

$$\text{Th}_{\text{diff}} = \frac{\frac{7 - 3\log_2 W}{4} W - 2}{\log_2 W - 1 + \frac{W}{2}} \frac{\text{MSS}}{rtt}.$$

Since

$$\log_2 W - 1 + \frac{W}{2} > 0$$

when $W \geq 2$ and

$$\begin{cases} \frac{7 - 3\log_2 W}{4} W - 2 = 0, & W = 2 \\ \frac{7 - 3\log_2 W}{4} W - 2 < 0, & W > 2 \end{cases},$$

we have $\text{Th}_{\text{timeout}} < \text{Th}_{dupack}$ when $W > 2$, which is the case for practical networking scenarios. When $W = 2$, actually there is no difference between Slow Start and Congestion Avoidance in terms of $cwnd$ evolution.

This analysis confirms that triple-*dupack* has much less impact than timeout on achievable throughput after a packet loss occurs. Also, triple-*dupack* takes much less time to detect packet losses than timeout, since the TCP timeout algorithm is designed to be very conservative and many implementation choices such as coarse

timeout-based scheme a quicker recovery.

timer granularity make it even more conservative. Ideally, if all packet losses were detected and recovered by triple-*dupack*, the performance would be improved substantially. In practice TCP has to rely on both triple-*dupack* and timeout since sometimes there is no *dupack* or insufficient *dupack*s for the lost packet. The above analysis gives the upper and lower throughput bounds. Section 3 reveals that TCP suffers many unnecessary timeouts over wireless links. This recognition motivates the proposal of a heuristic algorithm in Section 4, which eliminates some of the unnecessary timeouts.

3 TCP Performance over Wireless Links

In this section, the performance of TCP variants over wireless links is studied by software simulations. The simulation setup and the simulation parameters are described in the next subsection.

3.1 Simulation Setup

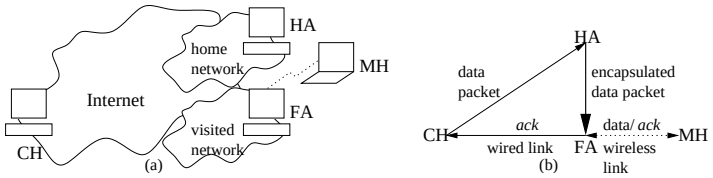

Figure 3: Simulation Configuration

The simulation model is based on the IETF Mobile IP (MIP) approach [2]. In Fig. 3(a), Mobile Host (MH) is a node which travels around the Internet and wants to maintain its ongoing communications with Correspondent Host (CH). MH has a *home address* for identification purpose. When away from its *home network*, MH is additionally assigned a *care-of address* which reflects its current point of attachment. Home Agent (HA) is a node in MH's home network and is informed of MH's current location. HA then intercepts packets sent to MH's home address and tunnels them to MH's care-of address. Foreign Agent (FA) is a node in MH's visited network which assists the control and packet exchange between MH and HA.

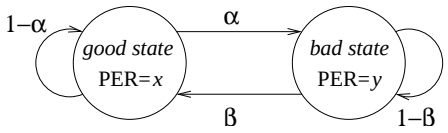

Figure 4: Wireless Links Model

The wireless lossy link between FA and MH is approximated by a two-state (namely *good* and *bad* states) generic model [14, 15] shown in Fig. 4. Each state has an independent submodel of its error behavior, *i.e.*, a function of air-link quality, user mobility, and packet length. If α and β are the *good-to-bad* and *bad-to-good* state transition probabilities, the steady residence probabilities for a link in the good and bad state are $\frac{\beta}{\alpha+\beta}$ and $\frac{\alpha}{\beta+\alpha}$. Let x and y be the Packet Error Rates (PER) in the good and bad states. Then $z = \frac{x\beta+y\alpha}{\alpha+\beta}$ is the *overall average error rate*.

Two types of applications are considered in the simulation. One is an FTP-like bulk data transfer that is conveyed by TCP. The other one is UDP-transported Constant Bit Rate (CBR) voice stream that has neither error nor flow control in the transport layer. As shown in Fig. 3(b), both data flows are unidirectional from CH to MH. The available bandwidth and propagation delay for wired links among any two nodes of CH, HA and FA are 1 $Mbps$ and 5 ms, and those for wireless links between FA and MH are 128 $Kbps$ and 25 ms. Although wireless links can have extra radio control protocols, only the characteristics observed by upper layers are of interest here.

According to the BSD reference implementation, the timer granularity is set about 500 ms for TCP senders in the *ns2* simulator [16] to sample *rtt* and calculate *rto*, and is about 200 ms for TCP receivers on *ack* delaying. TCP MSS and UDP payload are both 512 $bytes$, and TCP *rwnd* is about 8 $Kbytes$, which is assigned on purpose in the simulation to avoid any network congestion. The TCP and UDP *source* and *sink* agents in *ns2* have been modified accordingly with additional variables. and their values are captured after the simulation for offline analysis. FTP source is persistent, *i.e.*, it always has new data to send, and only keeps idle if it is limited by the TCP flow and congestion regulation. The PCM coded voice stream is also persistent at a constant data rate of 64 $Kbps$.

3.2 Observed End-to-End Performance

The observed end-to-end TCP performance over wireless lossy links, in terms of throughput degradation, is presented in the following. The simulation duration is 100 $seconds$, and different initial randomization seeds are used to eliminate the simulation dynamics and the system warming-up effects.

3.2.1 Packet Loss vs. Throughout Degradation

Fig. 5 illustrates the throughput degradation of FTP and CBR applications as a function of the overall packet loss rate z over wireless links. In this simulation, $\alpha = \beta = 0.3$ for the state locality, and z ranges from 0.01 to 0.13 in a step of 0.005 while x is always 0.01. Due to the combined TCP error and congestion control, the throughput of FTP traffic, no matter transported by which TCP variants, degrades dramatically in a negative-exponential manner, even when there are only few packet losses. TCP has to wait for a triple-*dupack* or timeout to detect packet losses, which causes the persistent sender to enter an idle state temporarily. While retransmitting the lost packet, TCP also shrinks *cwnd*, which decreases the amount

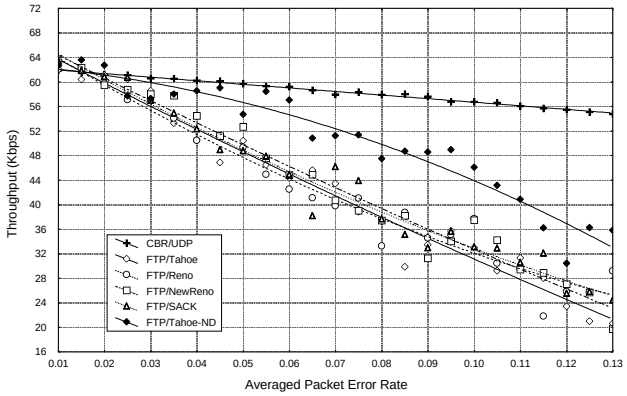

Figure 5: Packet vs. Throughput Loss

of in-flight packets. Both sender being idle and the shrinking of the *cwnd* reduce the achievable application throughput. In contrast, the throughput degradation for the UDP-transported CBR is proportional to z. Moreover, TCP may easily lose the desired *ack self-clocking*, especially with the default *delack*. Figure 5 shows that without *delack* (ND), TCP can better maintain *ack self-clocking* and get a relatively higher throughput.

3.2.2 Clustered vs. Random Error

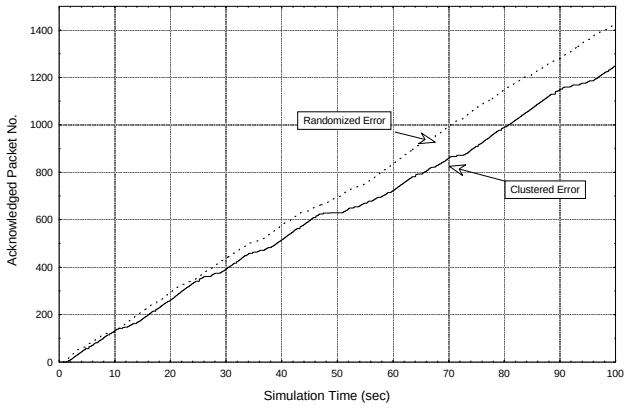

Figure 6: Clustered vs. Random Error - Tahoe

The simulation further indicates that the stronger a lossy link exhibits clustered error

behaviors, the more degradation TCP suffers. The results shown in Fig. 6 pertain to the cases where $x = y = 0.03$ for a randomized error link and $x = 0$ and $y = 0.06$ for a clustered error link, *i.e.*, $z = 0.03$ in both cases. The number of received FTP packets over the randomized error link is 1443 when the simulation ends, and that over the clustered link is only 1262, or about 12.5% less. The results in Fig. 6 show that TCP performs poorly in a clustered error environment. This phenomenon reveals that the TCP congestion algorithms are designed and optimized to combat randomized losses due to network congestion, and these algorithms might misbehave when the packet losses are in burst due to transmission errors over wireless links.

3.3 Explored Endpoint Behaviors

The TCP performance degradation observed above is explained by some unexpected TCP endpoint behaviors discussed below.

3.3.1 Multi-Loss in One Window

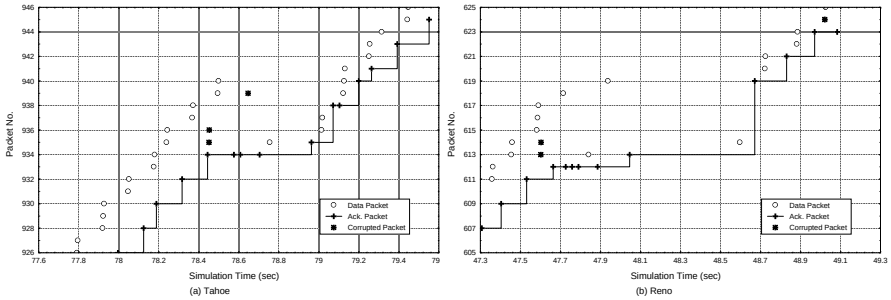

Figure 7: Multiple Packet Losses

When lossy links exhibit clustered error behaviors, it is quite common that more than one packet can get lost in the same *swnd*, *e.g.*, packets $\#613$ and $\#614$ in Fig. 7(b) and packets $\#935$ and $\#936$ in Fig. 7(a) with $x = 0.01$ and $y = 0.09$. Both Tahoe and Reno are triggered by a triple-*dupack* to fast-retransmit the first unacknowledged packet. However, there are usually insufficient *dupack*s to trigger further recoveries for other lost packets. After this Fast Retransmit, Reno halves its *cwnd* with *cwnd* inflating by sending packet $\#619$, and normally has to wait for a timeout to recover the second lost packet in the same *swnd*. On the other hand, Tahoe re-initializes its *cwnd* and follows Slow-Start immediately after the first Fast Retransmit. If multiple packet losses are consecutive, Tahoe usually outperforms Reno in error recovery. Therefore, in the following discussions, Tahoe is chosen as a representative TCP variant since it is more robust than Reno over wireless links. NewReno and SACK are designed to recover multiple losses in one *swnd*, and Fig. 5 does show that in most cases SACK outperforms NewReno since it has the

capability to recover multiple packet losses in one *rtt*, but both SACK and NewReno are challenged by the next unexpected behavior.

3.3.2 Tiny Congestion Window

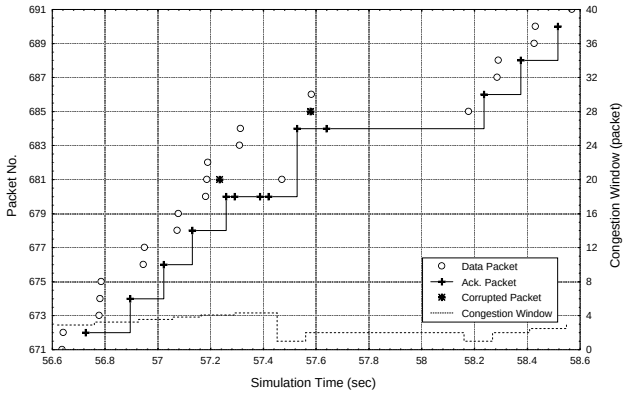

Figure 8: Tiny Congestion Window

Since congestion regulations are frequently enforced when TCP flows over lossy links, *cwnd* remains very small over relatively long intervals. As shown in Fig. 8 where $x = 0.01$ and $y = 0.09$, *cwnd* is about two MSS around the runtime $57.6sec$ and two full-sized packets are pumped into the network. The second one goes through successfully, while the first one is lost. In this situation, Fast Retransmit fates to fail since it is impossible for a triple-*dupack* to return. However, the sender should well anticipate that the first packet is lost. Most BSD-derived TCP implementations require at least three *dupack*s to trigger Fast Retransmit and Recovery procedures, as what happened around the runtime $57.48sec$. This requirement eliminates the side effect of packet reordering which also causes *dupack*. However, Fig. 8 indicates that in some situations this constraint is actually undesirable.

3.3.3 Resent Packet Loss

The third serious problem is that the resent packet may get lost again when TCP flows through lossy links that exhibit very clustered error behaviors. In this case, TCP has to recover lost packets by the exponential back-off timeouts. In Fig. 9 where $x = 0.01$ and $y = 0.15$, packet $\#618$ is resent three times, triggered by consecutive timeouts and interleaved exponentially by $0.7sec$, $1.4sec$ and $2.8sec$. Neither standard nor experimental TCP has the capability to detect and identify the loss of resent packets. Once it occurs, the sender remains unnecessarily idle for a comparatively long time, *e.g.*, a total of $4.9sec$ in Fig. 9. AIMD-based *cwnd* regulation and exponential timer backoff are designed to combat severe and persistent

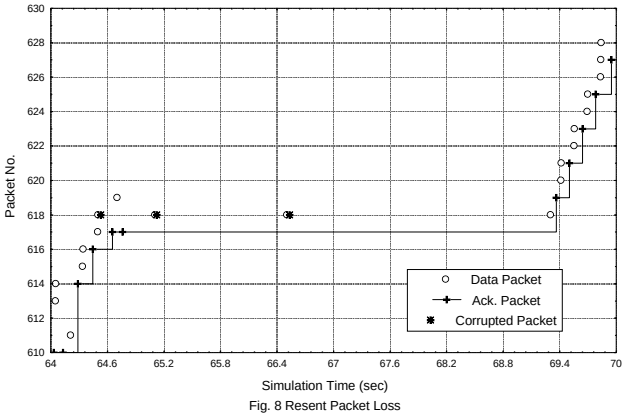

Figure 9: Resent Packet Loss

network congestion, and are too conservative for wireless lossy links.

4 Proposal for an Enhanced TCP Sender

In this section, the sources that trigger the end-to-end TCP retransmissions, *i.e.*, timeout and triple-*dupack*, are first quantitatively decomposed. An algorithm that enhances the TCP robustness and avoids some unnecessary timeouts is then proposed and evaluated in many detailed aspects.

4.1 TCP Retransmission Decomposition

Fig. 10 show the effect of decomposing the sources that trigger the end-to-end TCP retransmissions. *x-dupack* represents how many *dupack*s have been received when a timeout occurs. *failed-dupack* means when a timeout occurs TCP has already received *x dupack*s and there are a total of *x*+1 packets outstanding. *succeeded-dupack* denotes a Fast Retransmit event triggered by a triple-*dupack*. When the overall error rate is less than 3.5%, packet losses are more likely detected and recovered by triple-*dupack*. When the error rate goes higher, timeout becomes the dominant event for retransmission, which is less efficient than triple-*dupack* according to the analysis in Section 2.3. Meanwhile, the number of *failed-dupack* increases rapidly. This fact suggests that there is a large number of timeouts that can actually be avoided, so that the TCP performance can be improved considerably.

A tiny *cwnd* is not the only cause that prevents the efficient triple-*dupack* from being effective. TCP senders might also be kept unnecessarily idle if a receiver only advertises a *very small window*, or the connection is *session-limited*, *i.e.*, the application does not always have sufficient data to fully utilize the available sender

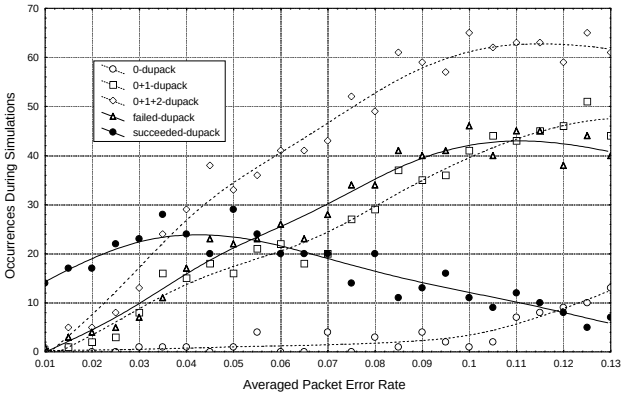

Figure 10: TCP Retransmission Decomposition

window. An enhanced TCP sender is proposed in the following to deal with all these situations and to improve the TCP performance.

4.2 Segment-in-Flight Estimation Algorithm

If a TCP sender has accumulated a certain number of *dupack*s and has the confidence that the next unacknowledged packet is lost, it can invoke Fast Retransmit immediately. The number of *dupack*s should be accumulated is a function of the number of currently unacknowledged packets through the **Segment-In-Flight** Estimation (*sifest*) algorithm proposed in this paper.

A new variable *sifest_*, which is an estimation of TCP sender on the amount of in-flight segments and is zero initially, is proposed to be added at the sender for every active TCP session. When a new data packet is sent, *sifest_* is increased by one. When the TCP sender gets a *newack*, *sifest_* is decreased by the number of acknowledged segments, but it is always kept non-negative. If a timeout occurs or the sender has been idle for more than one *rtt*, *sifest_* is zero again since the sender is going to re-probe the network. If the sender retransmits a packet triggered by schemes other than a timeout, *sifest_* remains the same, since it is assumed that the previous copy of this packet has already left the network. Since there is no need to modify TCP receivers or intermediate systems, the *sifest*-enhanced TCP sender can be deployed incrementally and interact with ordinary TCP receivers compatibly.

When a TCP sender already has $(sifest\_ - 1)$ *dupack*s, or the number of *dupack*s has exceeded the fixed threshold, and if the most unacknowledged packet has not been resent in the last *rtt*, this packet is resent immediately according to Fast Retransmit. More new data packets can be pumped into the network, if allowed by *cwnd* or the *cwnd inflating* algorithm, to trigger more *ack*s to help TCP endpoints regain *ack self-clocking*. Integrating the *sifest* algorithm with ordinary TCP variants is quite straightforward and conflict-free. The enhanced TCP sender might

have less tolerance on re-ordered packets, and the estimation error may accumulate for a while. However, when *sifest* is effective, since there are only a few in-flight segments and the estimation is re-initialized periodically, the probability of *dupack* occurrence due to packet recording is negligible according to our performance study.

Integration with Large Initial Window (LIW) [9] is expected to further enhance *sifest*'s capability to avoid unnecessary idle periods at TCP senders. When a TCP sender performs Slow-Start after a timeout or Fast Retransmit, let TCP probe the available data pipe with a two-segment initial *cwnd*. If only the first packet gets lost, *sifest* has the capability to fast-retransmit the lost packet without waiting for a timeout. If only the second packet gets lost, the TCP sender can pump more packets into the network once the *ack* of the first one returns, and still has a chance to recover the lost packet without a timeout. When both packets are lost, the TCP sender has to wait for a timeout. This is the same situation when TCP probes with a one-segment initial *cwnd* and the packet gets lost. However, the chance for both packets getting lost in the enhanced scheme is much smaller.

4.3 Performance Evaluation

The performance of the *sifest*-enhanced TCP is evaluated and compared with ordinary TCP variants.

4.3.1 Performance with SIF

The TCP modules in *ns2* are modified accordingly to incorporate the proposed *sifest* algorithm, and the LIW option. These add-on modules are switchable for any instanced TCP agents through the *TCL* script, and can be changed dynamically during the simulation.

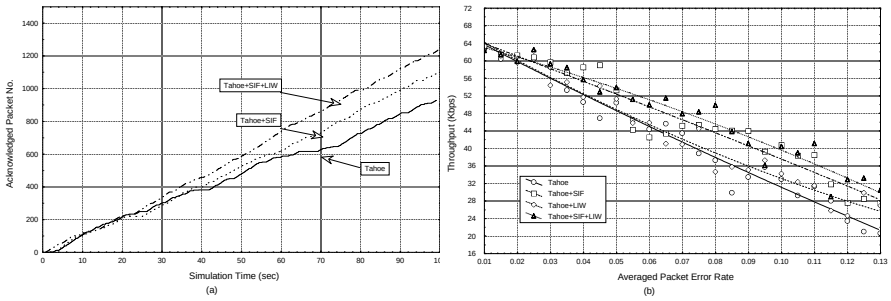

Figure 11: End-to-End Performance Comparisons

As shown in Fig. 11(a) where $x = 0.01$ and $y = 0.15$, the *sifest* algorithm improves the TCP performance in terms of a higher throughput and lower idle periods. When the simulation ends, the number of acknowledged packets is 932 for Tahoe, and 1111 for the *sifest*-enhanced Tahoe, which is 19.2% higher. The *ack* curve of Tahoe has many flat segments (*e.g.*, during the runtimes 37 to 42 sec and 64 to 69

sec), which indicate that during these periods the TCP sender is kept idle to wait for timeouts. If the link quality is not too bad, the *sifest*-enhanced Tahoe can successfully avoid most noticeable sender timeouts, keeping the *cwnd* comparatively large, and consequently yielding a higher throughput. Integrating with LIW, the *sifest*-enhanced Tahoe performs even better. The number of acknowledged packets for the hybrid SIF/LIW Tahoe is 1245, which is 33.6% higher than the plain Tahoe and 12.1% higher than the Tahoe enhanced with the *sifest* algorithm alone.

The performance gains due to the *sifest* algorithm for other TCP variants are quite similar. For example, when $x = 0.01$ and $y = 0.15$, 1059 or 831 packets are received for Reno with or without the *sifest* algorithm, respectively. Furthermore, *sifest* also improves the TCP performance over randomized lossy links. When $x = y = 0.08$, 1116 or 970 packets are received when the simulation ends for NewReno with or without the *sifest* algorithm, respectively.

Fig. 11(b) gives the throughput comparison as a function of z, and has the same simulation settings as Fig. 5. When the error rate is comparatively low (*e.g.*, $z \leq 0.03$), the difference between Tahoe and the *sifest*-enhanced Tahoe is quite trivial, since it is still in the region that the original algorithms function properly. When $z \geq 0.07$, the enhancement due to the *sifest* algorithm becomes quite obvious, since many timeouts and unnecessary idle periods are avoided, as depicted in Fig. 12(a) when comparing to Fig. 10. Even when the *sifest*-enhanced Tahoe receives more packets, it has fewer timeouts. Fig. 12(b) gives a better comparison by normalizing the number of timeout and Fast Retransmit events every 1000 acknowledged packets. It is obvious that about an half of the timeouts have been replaced by the more efficient Fast Retransmit with the *sifest* algorithm. This explains the performance improvement observed in Fig. 11.

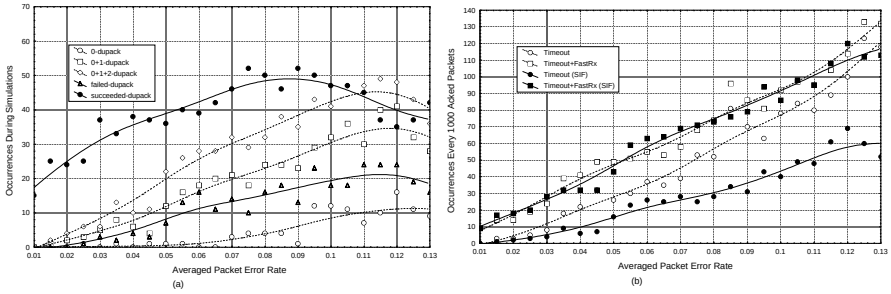

Figure 12: Retransmission Decomposition Comparison

4.3.2 Fairness with Ordinary TCP

The performance gain due to the *sifest* algorithm is further demonstrated from the margin that the ordinary TCP has wasted, not from the fair portion that other ordinary TCP sessions should share. To illustrate this, simulation parameters are slightly modified. Now the bandwidth for wireless links is increased to 192 $Kbps$, and an

additional FTP session conveyed by the ordinary TCP from CH to MH is introduced. Increasing the available bandwidth for wireless links is to ensure that these TCP connections are still congestion-free, with other parameters remain the same.

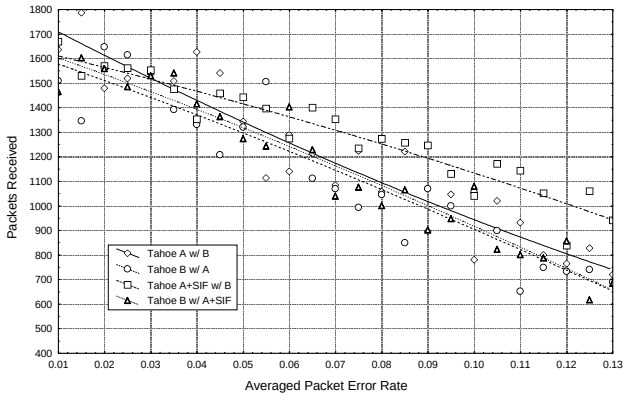

Figure 13: Fairness Validation

First, two FTP sessions conveyed by Tahoe A and Tahoe B, both are ordinary TCP, are simulated. Due to the phase effect between any two competing TCP connections, Fig. 13 shows that there is a slight performance difference between Tahoe A and Tahoe B, but the difference is almost independent of z. Then the same simulation is repeated with an ordinary Tahoe (A) and a *sifest*-enhanced Tahoe (B). Fig. 13 illustrates the *sifest*-enhanced Tahoe A outperforms the ordinary Tahoe B significantly when z goes higher. However, the ordinary Tahoe B that is competing with the *sifest*-enhanced Tahoe A has almost the same performance as that when it runs with an ordinary Tahoe. This fact confirms that the *sifest* algorithm does not take any fair shares that other competing TCP connections should have, which is the design objective of this algorithm. Similar simulation evaluations indicate that the *sifest*-enhanced Reno, NewReno and SACK also have the fairness and compatibility properties with ordinary TCP variants.

5 Related Work

After the discovery of the TCP performance degradation over wireless and other lossy links [6, 11], there are many empirical measurement studies in the literature that confirmed similar observations with different network settings [4, 5]. In contrast, besides the investigation of TCP performance over Mobile IP for end-to-end TCP connections in hybrid wireless/wired networks, this paper further features an in-depth exploration of internal TCP end-point behaviors that are responsible for the performance degradation. This approach allows us to exactly identify the problems and then develop a *sifest* algorithm that improves the TCP robustness when

the number of in-flight segments is small. Moreover, the performance improvement also applies when the receiver only has small buffer (*e.g.*, a PDA) or the application is session limited (*e.g.*, multi-request-reply transactions).

Many brand new schemes in different protocol layers have been proposed in the literature to improve the TCP performance in these challenged networks (refer to [4, 3, 5] and the references therein for more discussions and comparisons). However, due to the compatibility and interoperability issues, most new schemes failed to be developed and deployed in an incremental manner over the Internet, since they need to introduce dramatic changes at intermediate systems or receivers. The approach adopted in this paper takes advantage of the existing TCP variants and only adds a new variable per connection at the TCP sender. For most Internet applications, the centralized servers are the TCP sender for bulk data transfer. This fact enables us only to upgrade the TCP implementation in these servers, which are much less in magnitude than individual clients, to improve the performance perceived by all wireless and mobile users. Also the competition among service providers can speed up this server-based deployment incrementally.

Recently, Limited Transmit was proposed by Allman *et al.* [17] to allow TCP senders to keep probing the network with new packets that have not been transmitted yet when receiving *dupack*s. This scheme can somehow alleviate the occurrence of timeout by triggering more *dupack*s to reach the fixed threshold of 3. However, this scheme still suffers when the receiver window is small or the application is session limited. More importantly, without an estimation of in-flight segments, the TCP sender still does not have the capability to retransmit the lost packets in a timely manner. Given the fact that TCP (except for TCP SACK) is an accumulatively acknowledged protocol, the armed timer for old packets inevitably runs the risk of unnecessary timeout when waiting for the extra *dupack*s triggered by new data packets. In contrast to the reactive approach employed by Limited Transmit which mandates additional control delay, the *sifest* algorithm always keeps a proactive and up-to-date estimation of in-flight segments and retransmits the lost packet immediately when it has accumulated enough confidence.

6 Conclusions

This paper presents an analysis and evaluation of the end-to-end TCP performance and the responsible endpoint behaviors over wireless links. An enhanced TCP sender with the Segment-in-Flight Estimation algorithm is proposed to improve the TCP performance over wireless links, in terms of higher application throughput and less sender idle time. It is demonstrated that, by trading off the deficient timeout by the much more efficient *dupack* on loss detection and recovery, TCP performance over wireless links can be improved considerably, while the fairness, compatibility and scalability with ordinary TCP are still well maintained.

Acknowledgments

This work has been supported by the National Science and Engineering Research Council (NSERC) of Canada under grants No. RGPIN7779 and No. RGPIN203560.

References

[1] A. Seneviratne, and B. Sarikaya, "Cellular networks and mobile Internet," *Computer Networks*, 21(1998):1244-1255.

[2] C. E. Perkins ed., "IP mobility support," *IETF RFC 2002*, 1996.

[3] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz, "Comparison of mechanisms for improving TCP performance over wireless links," *IEEE Trans. on Networking*, 5(6):756-769, 1997.

[4] J. W. Mark, X. Shen, Y. Zeng, and J. Pan, "TCP/IP in wireless/Internet interworking," *Proc. IEEE 3Gwireless*, 2000.

[5] K. Pentikousis, "TCP in wired-cum-wireless environments," *IEEE Communications Surveys & Tutorials*, 3(4), 2000.

[6] A. DeSimone, M. Chuah, and O. Yue, "Throughput performance of transport-layer protocols over wireless LANs," *GlobeCom'93 Conf. Record*, 542-549, 1993.

[7] M. Allman, V. Paxson, and W. Stevens, "TCP congestion control," *IETF RFC 2581*, 1999.

[8] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," *IETF RFC 1323*, 1992.

[9] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's initial window," *IETF RFC 2414*, 1998.

[10] H. Balakrishnan, and V. N. Padmanabhan, "The effects of asymmetry on TCP performance," *Proc. ACM MobiCom'97*, 1997.

[11] T. Lakshman and U. Madhow, "The performance of TCP/IP for networks with high delay-bandwidth products and random loss," *IEEE/ACM Transactions on Networking*, 5(3):336-350, 1997.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," *IETF RFC 2018*, 1996.

[13] S. Floyd, and T. Henderson, "The NewReno modification to TCP's fast recovery algorithm," *IETF RFC 2582*, 1999.

[14] A. Chockalingam, M. Zorzi, and R. Rao, "Performance of TCP on wireless fading links with memory," *Proc ICC'98*, 595-600, 1998.

[15] M. Zorzi, and R. Rao, "Effect of correlated errors on the performance of TCP," *IEEE Comm. Letters*, 1(5):127-129, 1997.

[16] S. Bajaj, L. Breslau, and D. Estrin, "Improving Simulation for Network Research," *TR-99-702*, Univ. of Southern California, 1999.

[17] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's loss recovery using Limited Transmit," *IETF RFC 3042*, 2001.

Received March 2004; revised July 2004.
emailto:journal@monotone.uwaterloo.ca
`http://monotone.uwaterloo.ca/~journal/`