Enhancing TCP Performance to Persistent Packet Reordering

Ka-Cheong Leung and Changming Ma

Abstract: **In this paper, we propose a simple algorithm to adaptively adjust the value of *dupthresh*, the duplicate acknowledgement threshold that triggers the transmission control protocol (TCP) fast retransmission algorithm, to improve the TCP performance in a network environment with persistent packet reordering. Our algorithm uses an exponentially weighted moving average (EWMA) and the mean deviation of the lengths of the reordering events reported by a TCP receiver with the duplicate selective acknowledgement (DSACK) extension to estimate the value of *dupthresh*. We also apply an adaptive upper bound on *dupthresh* to avoid the retransmission timeout events. In addition, our algorithm includes a mechanism to exponentially reduce *dupthresh* when the retransmission timer expires. With these mechanisms, our algorithm is capable of converging to and staying at a near-optimal interval of *dupthresh*. The simulation results show that our algorithm improves the protocol performance significantly with minimal overheads, achieving a greater throughput and fewer false fast retransmissions.**

Index Terms: **Computer communications, congestion control, dispersity routing, high-speed networks, multipath routing, transmission control protocol (TCP).**

I. INTRODUCTION

Recent studies have suggested that packet reordering is not a pathological behaviour in the Internet [1]. Yet, the impact of packet reordering on protocol performance is significant, especially for transmission control protocol (TCP), the most commonly used transport protocol in the Internet.

Packet reordering can be caused by misconfigured or malfunctioning network components that leads to frequent route fluttering or router pauses. The inherent parallelism in modern packet switches also brings about packet reordering during normal operation. Besides, multipath routing [2]–[4] is an effective traffic engineering technique to improve network throughput and reduce network load fluctuation. It has shown [5], [6] that not only does multipath routing balance the load significantly better than single-path routing over wired networks, but also it provides better performance in congestion and capacity over mobile ad hoc networks. In packet-switched networks such as the Internet, the smallest data switching unit is a packet. A packet flow can be split over multiple paths between a source and a destination in a multipath network. When packets travel on paths with different round-trip times (RTTs), they may arrive out-of-order at the destination.

The standard TCP source agent does not receive any explicit information about the current congestion status from the underlying protocols. It probes the available network bandwidth by increasing the congestion window size until a packet (or segment) loss occurs, at when it shrinks the window size. The TCP fast retransmission algorithm, running in parallel with the timeout mechanism, exploits the fact that the TCP receiver always acknowledges the last segment successfully received in the correct order. The reception of duplicate acknowledgements (ACKs) can be an indication to the sender for the occurrences of either packet reordering or packet loss. The ability to disambiguate these two cases can improve the protocol performance considerably. If the network paths reorder packets persistently and packet reorderings are interpreted as packet losses, the fast retransmission algorithm is activated frequently to resend packets which have not been lost, wasting network bandwidth and keeping window size unnecessarily small. Besides, persistent spurious retransmission can exacerbate network congestion, lead to classical congestion collapse, and reduce the TCP connection throughput [7].

A. Our Contributions

Although it is hard, both economically and theoretically, to eliminate packet reordering, recent research has been conducted to improve the reordering robustness of TCP. In this paper, we survey some of these proposals and devise a simple algorithm to adaptively adjust the value of *dupthresh* to improve the TCP performance in a network environment with persistent reordering.

Our algorithm uses an exponentially weighted moving average (EWMA) and the mean deviation of the lengths of the reordering events reported by a TCP receiver with the duplicate selective acknowledgement (DSACK) extension to estimate the value of *dupthresh*. We also apply an adaptive upper bound on *dupthresh* to prevent *dupthresh* too high to trigger a retransmission timeout. In addition, our algorithm includes a mechanism to exponentially reduce *dupthresh* when the retransmission timer expires. Our algorithm is engineered to avoid a certain portion of the false fast retransmissions so as to strike a balance between the avoidance of spurious retransmissions due to packet reordering and timely retransmissions of lost packets.

B. Organization of the Paper

This paper is organized as follows. Section II gives a survey of the related work. Section III presents our algorithm to adaptively adjust the value of *dupthresh*, the duplicate acknowledgement threshold that triggers the TCP fast retransmis-

Manuscript received August 9, 2004; approved for publication by Song Chong, Division III Editor, May 25, 2005.

K.-C. Leung is with the Department of Electrical and Electronic Engineering, the University of Hong Kong, Pokfulam Road, Hong Kong, China, email: kcleung@ieee.org.

C. Ma is with the Department of Computer Science, Texas Tech University, Lubbock, TX 79409-3104, USA, email: ma@cs.ttu.edu.

sion algorithm, to improve the TCP performance in a network environment with persistent packet reordering. Section IV examines our simulation results and discusses the effectiveness of our proposed algorithm for improving the reordering robustness of TCP. Section V concludes and discusses some possible extensions to our work.

II. RELATED WORK

The DSACK extension [8] to the TCP SACK option [9] has been proposed to make TCP more robust to packet reordering. The information of spurious retransmission inferred from DSACK is helpful in adjusting the sender behaviour to improve the TCP performance. Originally, the TCP fast retransmission algorithm is triggered when three duplicate acknowledgements are received [10], [11]. Some approaches that adaptively modify *dupthresh* have been developed to make TCP more robust in the presence of various levels of packet reordering.

Several techniques have been proposed in [12] to adjust *dupthresh* by

1. constantly increasing *dupthresh*;
2. increasing *dupthresh* based on the average length of a reordering event and the current value of *dupthresh*;
3. using a duplicate ACK threshold and a delay timer; and
4. using a running average of the reordering length as the estimator of *dupthresh*.

Hereafter, we will refer to the above algorithms as INC, AVG, DEL, and EWMA, respectively.

Their simulation results showed that, when compared with the original fixed *dupthresh* method, the proposed techniques improved throughput with different extents. The algorithms also reduced the numbers of unnecessary retransmissions. However, their algorithms have several shortcomings. *dupthresh* is not very sensitive to the dynamic behaviour of the reordering events. It slowly converges to a satisfying value. The upper bound of *dupthresh* is set to 0.9 *cwnd*, where *cwnd* is the size of the congestion window (in segments). A retransmission timeout may occur when multiple packets are reordered or lost within the same congestion window. When *cwnd* is small, the false fast retransmissions can also happen. When a retransmission timeout occurs, *dupthresh* is reset to three, losing all historical information that should be useful in adjusting *dupthresh* after the reset.

RR-TCP [13] extended the sender to detect and recover from the false fast retransmissions using the DSACK extension, and avoided future false fast retransmissions proactively by adaptively changing *dupthresh*. Their simulation results showed that RR-TCP could significantly improve the TCP performance over reordering networks. It employed a reordering histogram to store the reordering information. This information can be used to adjust *dupthresh* indirectly via the false fast retransmit avoidance (FA) ratio, the percentile value in the cumulative reordering length distribution.

A timer-based approach to avoid false fast retransmission has been proposed in [14]. It employed a timer, of which the threshold is a function of RTT, to trigger fast retransmission. In fact, the DEL algorithm [12] could be viewed as an extension to this approach. TCP-PR [15] also utilized timers. It did not rely on duplicate ACKs. However, it was computationally expensive to estimate the maximum possible round-trip time (as the value of the retransmission timeout) since a series of exponentiation computations had to be performed on every ACK arrival. The Eifel algorithm [16] enhanced the TCP error recovery mechanism. It detected false timeouts and false fast retransmissions and revoked their penalties. However, the algorithm did not proactively avoid the false fast retransmissions. Yet, we focus on how to avoid the false fast retransmissions by adjusting *dupthresh* with minimal overheads in this paper.

An integrated sender-side and receiver-side solution to improve the TCP performance over multiple paths has been proposed in [17]. On the sender side, *dupthresh* was set to increase logarithmically on the number of paths used. On the receiver side, delayed ACKs were generated for out-of-order packet arrivals. However, the level of packet reordering depends on the differences in path delays and how the packets belonging to a single flow are distributed to these paths, but there exists no direct correlation between the extent of packet reordering and the number of paths used [18]. This approach also requires modifications to both TCP senders and receivers to achieve the desired performance improvement.

RR-TCP [13] requires excessive computational and storage overheads, whereas the techniques proposed in [12] may not adapt well to the real network conditions. Particularly, their methods are not sensitive enough to accommodate to changes in the network environment promptly. In this paper, we propose an algorithm which possesses the similar performance speedup as in RR-TCP with much less overheads to improve the TCP robustness in case of significant packet reordering.

III. OUR ALGORITHM

A. Detecting False Fast Retransmit

As in [12] and [13], we use the DSACK extension in TCP to detect the occurrences of the false fast retransmissions. A TCP sender is able to learn whether a retransmission is necessary, using the DSACK option and the historical segment retransmission information stored in the sender's scoreboard. If the sender later receives both the ACKs of the original packet and the spuriously retransmitted packet, it can detect a false fast retransmission and potentially recover from its adverse impact on the TCP performance by undoing the reduction of the congestion window size. The DSACK specification itself does not stipulate the sender's actions upon receiving the DSACK information. However, the information is helpful for improving protocol performance by accomplishing the following two objectives.

1. Recovering from the unnecessary window size backoffs during fast retransmit.
2. Avoiding any future false fast retransmissions by adjusting *dupthresh*.

To satisfy the first objective, the unnecessary window reduction is rolled back to the most recent value prior to the false fast retransmission. Similar to the approach used in [12], the sender uses slow start to increase the size of the congestion window to its prior value so as to avoid injecting traffic bursts to the network. The second objective is achieved by using the following techniques to adjust the value of *dupthresh*.

B. Adjusting the Value of dupthresh

The key idea of our algorithm is to evaluate an exponentially weighted moving average (EWMA), *avg*, of the lengths of the detected reordering events. To render *dupthresh* consistent with the dispersion of the reordering lengths and make it more conservative (but not overly conservative) in discriminating packet reordering and packet loss, we let *dupthresh* be the sum of *avg* and a fraction of *mdev*, where *mdev* is the mean deviation of the reordering length samples. In our implementation, the mean deviation is used instead of the standard deviation of the reordering length for computational simplicity. The use of EWMA on the reordering length has been proposed in [12] as an alternative to adjust *dupthresh*. Our method, however, differs from theirs as it adds a fraction of *mdev* into *dupthresh*. Theoretically, by including this additional term, it is possible for *dupthresh* to avoid a certain portion of the false fast retransmissions due to packet reordering. To a certain extent, it shares the same design philosophy as the FA ratio proposed in [13], but it incurs less overheads. As inferred from our simulation results, this seemingly minor modification (together with some other enhancements described in the following subsections) can improve the protocol performance substantially. Our procedure is described in (1)–(4).

Let r be the $(k+1)$-th sample of the reordering length, i.e., the length of the $(k+1)$-th reordering event detected by the TCP sender, and α be a pre-defined smoothing constant (typically, $\alpha \in [0.3, 0.4]$ is used in our experiments). The EWMA value, *avg*, is calculated as follows.

$$avg(k+1) = \alpha \cdot r + (1-\alpha) \cdot avg(k) \tag{1}$$

where $avg(k)$ is the original EWMA value and $avg(k+1)$ is the new value updated with r.

The mean deviation of the reordering length samples, with a small weight to the most recent instance (our simulation experiments show that $\beta \in [0.2, 0.4]$ achieves the best result), is used to estimate the value of *mdev*.

$$aerr(k+1) = |r - avg(k)|, \tag{2}$$
$$mdev(k+1) = \beta \cdot aerr(k+1) + (1-\beta) \cdot mdev(k) \tag{3}$$

where *aerr* and *mdev* are the absolute error and the mean deviation of the reordering length, respectively.

When a new reordering event is detected, the new values of *avg* and *mdev* are recomputed using (1), (2), and (3). Then, the value of *dupthresh* is computed as

$$dupthresh = \lfloor avg + \lambda \cdot mdev \rfloor \tag{4}$$

where λ is a pre-set parameter (typically, $\lambda \in [0.2, 0.4]$).

C. Avoiding Timeouts with Upper Bound

Merely increasing *dupthresh* can possibly trigger a retransmission timeout when a very large *dupthresh* prevents the sender from timely retransmitting a lost packet. To overcome this problem, we apply an upper bound $dupthresh_{ub}$ upon our *dupthresh* to reduce the possibility of a timeout event.

Let RTO be the value of the retransmission timeout, RTT be the estimated value of the round-trip time, and $T(m)$ be the total time elapsed between when the sender transmits a packet (which is lost in the network) and when the sender receives the ACK of the retransmitted packet (which is sent after m duplicate ACKs have been received). If we can guarantee that $T(m)$ is less than RTO, we have a good chance of avoiding a timeout event. By applying the TCP self-clocking effect [19],

$$T(m) = 2 \cdot RTT + m \cdot T_{int} \tag{5}$$

where T_{int} is the average inter-packet time.

The average inter-packet time can be evaluated from the congestion window size *cwnd* and RTT. If the transmission paths used by a TCP connection are viewed as a queueing system, again by self-clocking, the arrival rate is $\frac{1}{T_{int}}$, the residence time (the time spent by a packet and its ACK in the system) is RTT, and the number of items in the system is *cwnd*. According to Little's theorem [20],

$$cwnd = \frac{RTT}{T_{int}}. \tag{6}$$

From (5) and (6),

$$T(m) = 2 \cdot RTT + m \cdot \frac{RTT}{cwnd}. \tag{7}$$

To prevent the TCP sender from timeout, $T(m)$ should be less than RTO. To introduce a safety margin to counteract the estimation errors of RTT and T_{int}, we let $T(m)$ be a portion of RTO. Assume γ is a constant which is less than 1,

$$T(m) \leq \gamma \cdot RTO. \tag{8}$$

By substituting (7) into (8),

$$2 \cdot RTT + m \cdot \frac{RTT}{cwnd} \leq \gamma \cdot RTO. \tag{9}$$

In order to avoid a retransmission timeout, the value of m is given by

$$m \leq \left(\gamma \cdot \frac{RTO}{RTT} - 2\right) \cdot cwnd. \tag{10}$$

An upper bound of *dupthresh* is thus given by

$$dupthresh_{ubf} = \left\lfloor \left(\gamma \cdot \frac{RTO}{RTT} - 2\right) \cdot cwnd \right\rfloor. \tag{11}$$

In this way, we can theoretically prevent *dupthresh* from becoming too high to trigger a retransmission timeout. In practice, however, the accuracy of $dupthresh_{ubf}$ depends upon the estimators of RTT and RTO. The occurrences of timeout events can be reduced by $dupthresh_{ubf}$, but it cannot be avoided entirely. When a timeout event occurs, we use the value of *dupthresh* at that time to serve as the second upper bound. This value is called $dupthresh_{tmo}$ and acts as an auxiliary constraint to the upper bound of *dupthresh*

$$dupthresh_{ub} = \min(dupthresh_{ubf}, dupthresh_{tmo}). \tag{12}$$

That is, the ultimate upper bound of *dupthresh* is the minimum of the value given in (11) and the most recent value of *dupthresh* which leads to a timeout event.

```
Algorithm AVG-DEV
Begin
    after a reordering event of length N is detected
        update avg using (1);
        update mdev using (3);
        update dupthresh using (4);
        recompute dupthresh upper bound using (11) and (12);
        if ( dupthresh >= dupthresh upper bound
            dupthresh = dupthresh upper bound;
    after a retransmit timeout event
        save the current value of dupthresh;
        recompute avg using (13);
        recompute mdev using (14);
        update dupthresh using (4), (11), and (12);
        if ( dupthresh >= dupthresh upper bound
            dupthresh = dupthresh upper bound;
    after receiving k-th duplicate ACK
        if ( k > dupthresh
            retransmit the segment following
            the one that is ACKed in duplicate;
End
```

Fig. 1. Pseudo-code of the combined algorithm.

D. Decreasing dupthresh

In the previous subsections, we have discussed how to adjust *dupthresh* when detecting a false fast retransmission. We still need a strategy to decrease *dupthresh* when RTO expires. The algorithm in [12] simply resets *dupthresh* to 3 upon a timeout. The algorithm proposed in [13] reduces *dupthresh* based on the ratio of the cost of a timeout to that of a false fast retransmission. The first strategy is too crude to be used in a real network since it loses information about the current value of *dupthresh* after *dupthresh* is reset. We also believe that the second approach is too complex and involves too much overheads as a histogram storing the reordering length distribution has to be maintained and manipulated. Based on our simulation results and those given in [12] and [13], we decide to multiply *avg* and *mdev* with two positive constants C_1 and C_2, respectively. These two constants are both less than unity. The procedure leads to multiplicative decreases of *avg* and *mdev* in order to achieve a fast convergence to the targeted *dupthresh*. It can be viewed as adding some dumping to our *dupthresh*-adjusting mechanism. As shown in [21], a linear system can be stabilized by adding exponential dumping components. When a retransmission timeout occurs,

$$avg = C_1 \cdot avg, \tag{13}$$

$$mdev = C_2 \cdot mdev. \tag{14}$$

dupthresh is then computed using (4).

E. The Combined Algorithm

Our combined algorithm, referred to hereafter as AVG-DEV, is shown in Fig. 1.

IV. PERFORMANCE EVALUATION

In this section, we present our simulation results, compare our proposed algorithm with those described in [12] and [13], and discuss the fairness issue of our algorithm. Section IV-A considers and compares the performance of the algorithms under study over a single-path network topology. Section IV-B investigates the performance of these algorithms over a multipath network topology. Section IV-C compares the space and computational overheads of our method to that of RR-TCP. A discussion of fairness issue of our algorithm is given in Section IV-D.

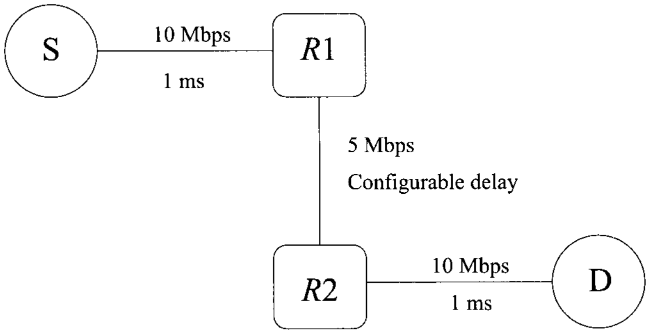


Fig. 2. Single-path network topology.

A. Single-Path Network

Our simulations are carried out in *ns-2*. The first topology, a single-path network topology, is shown in Fig. 2. It involves two end-systems (S and D) and two routers ($R1$ and $R2$). A single TCP flow from S to D lasting for 1000 seconds is simulated. The sender S uses the $sack1$ TCP and the receiver D is capable of generating the DSACK information.

The path between $R1$ and $R2$ models the underlying network path connecting $R1$ and $R2$. The path usually consists of multiple hops. A hop-count average of 16.2 was reported in [22] for a path in the Internet. The central limit theorem [23] suggests that the end-to-end delay over a multi-hop path, which is the sum of a large number of independent hop-delays, is approximately normally distributed. To simulate packet reordering (such as those caused by route fluttering), we periodically change the $R1$-$R2$ path delay according to a normal distribution. The time interval between two successive changes in delay, denoted as the delay update interval, imitates various extents of the reordering events. In our simulation, we update the delay every 50 ms or 100 ms. The former will introduce reordering events more frequently. The mean and standard deviation of the delay distribution simulate the reordering length distribution itself. The mean and standard deviation of the delay are $200f$ ms and $\frac{200f}{3}$ ms, respectively, where f is the "packet delay factor." The factor f ranges from 1.0 to 3.8 in our simulations. A larger packet delay factor results in reordering events with longer reordering lengths. We are going to demonstrate the impact of the delay distribution to packet reordering when we show the simulation results with different delay update intervals and delay distributions.

The setting of the simulation parameters are summarized in Table 1. Before collecting our experimental results, we have conducted a small subset of experiments to determine how the parameters (α, β, λ, γ, C_1, and C_2) are chosen. For each experiment, we have examined a large number of combinations of these parameters. The selected parameter configuration repre-

Table 1. Setting of the simulation parameters.

Parameter	Value
Mean of $R1$-$R2$ path delay	$200f$ ms, $f \in [1,4)$
Standard deviation of path delay	$\frac{200f}{3}$ ms, $f \in [1,4)$
Interval between two successive delay changes	50 ms, 100 ms
Maximum *cwnd*	100 packets
Minimum RTO	1 s
α in (1)	0.3
β in (3)	0.3
λ in (4)	0.3
γ in (8)	0.7
C_1 in (13)	0.5
C_2 in (14)	0.25

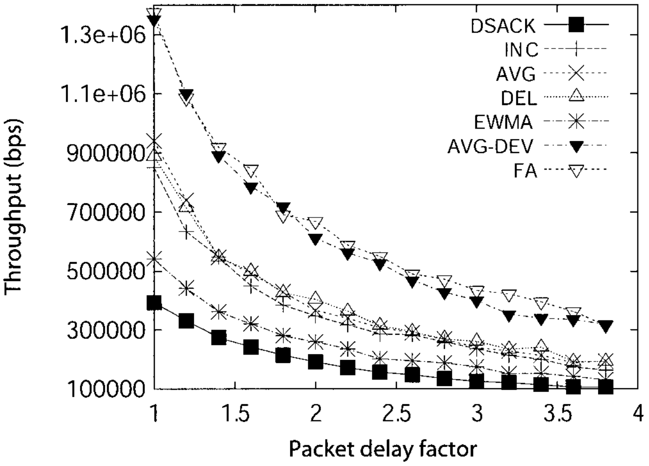


Fig. 3. Throughput against packet delay factor. Path delay is changed every 50 ms. No packet loss.

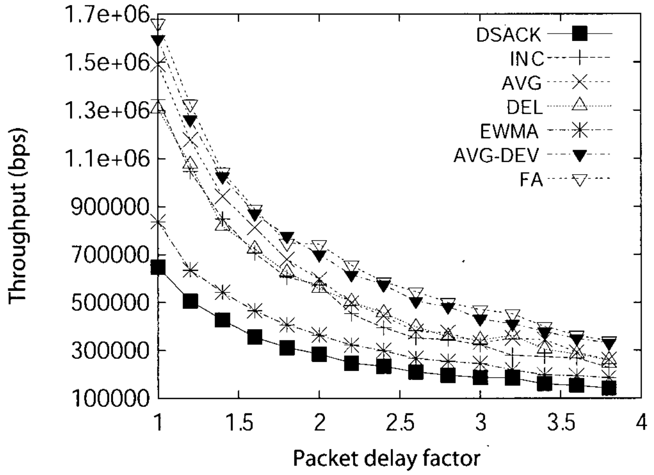


Fig. 4. Throughput against packet delay factor. Path delay is changed every 100 ms. No packet loss.

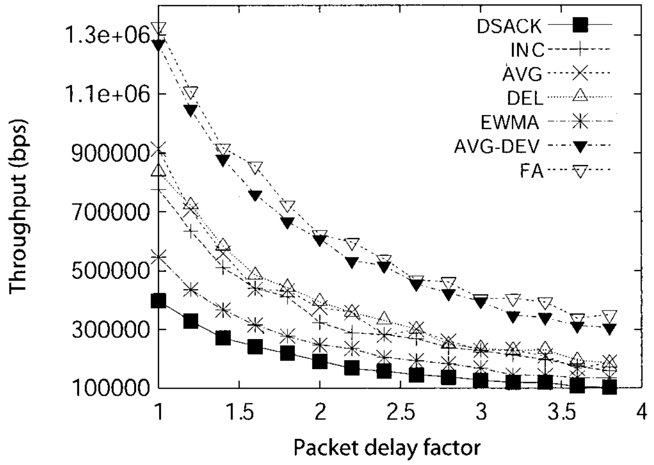


Fig. 5. Throughput against packet delay factor. Path delay is changed every 50 ms. Packet loss rate is 0.2%.

sents the best performance of our proposed algorithm throughout these preliminary experiments. Indeed, the performance of our algorithm is rather insensitive to α, β, and λ so long as they are taken within their operating regions as specified in Section III-B. We plan to devise an adaptive algorithm to dynamically estimate the values of these parameters as part of our future study.

In all of the following figures, DSACK is the method using the DSACK TCP with a fixed *dupthresh* of 3. INC, AVG, DEL, and EWMA are those techniques proposed in [12]. AVG-DEV is our proposed algorithm. FA is an algorithm of RR-TCP [13], DSACK-FA-MEAN with an enhanced RTT sampling (ES), with a fixed FA ratio. The parameters are configured the same way as described in [13].

The simulation results, in terms of the average connection throughput over 15 runs, are shown in Figs. 3–5. We change the path delay every 50 ms in Figs. 3 and 5, while the path delay is altered every 100 ms in Fig. 4. This means that the reordering events occur more frequently in Figs. 3 and 5 than those in Fig. 4. In addition, we introduce packet loss with the loss rate of 0.2% in Fig. 5. The packet delay factor f is within the interval of $[1,4)$ for all results shown in these three figures. A larger value of f yields a larger mean and standard deviation of the path delay. This results in more dispersed reordering lengths, that corresponds to the network scenarios with more severe reordering events.

As exhibited in Fig. 3, our algorithm improves the connection throughput by around 40%–80% compared to DEL, INC, and AVG. When it is compared to EWMA, it achieves the throughput improvement by 120%–150%. This shows that our algo-

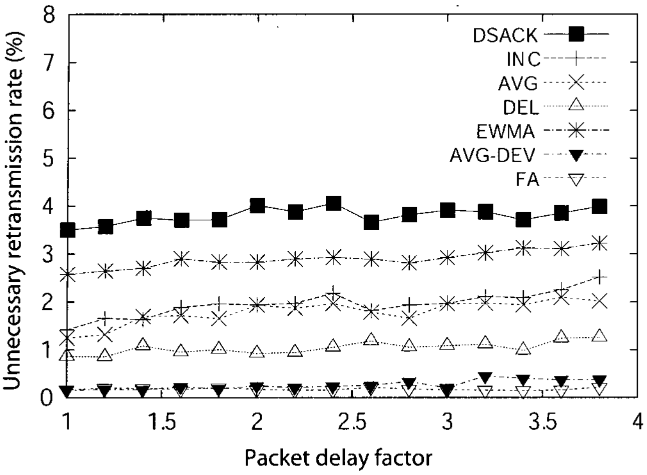


Fig. 6. Unnecessary retransmission rate against packet delay factor. Path delay is changed every 50 ms. No packet loss.

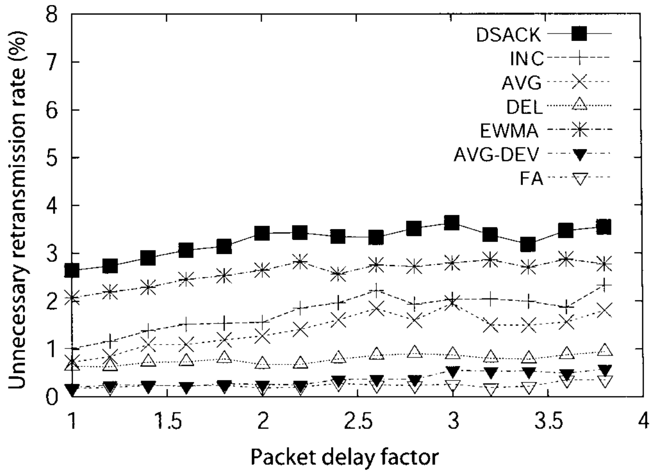


Fig. 7. Unnecessary retransmission rate against packet delay factor. Path delay is changed every 100 ms. No packet loss.

rithm adapts very well when the reordering events occur more frequently. The introduction of the upper bound to *dupthresh* effectively prevents it from triggering timeout events. When compared to FA, AVG-DEV possesses a very similar performance improvement.

When the reordering events happen less frequently, our algorithm achieves less throughput improvement (about 15%–35% compared to DEL, 7%–30% compared to AVG, and 75%–100% compared to EWMA). This is reasonable, since our algorithm adaptively adjusts *dupthresh* when a reordering event occurs. A fewer occurrences of ordering events means smaller performance differences among all the methods examined in our experiments. This is demonstrated in Fig. 4. The simulation results in [12] did not evaluate the impact of packet loss to their algorithms. We introduce random packet loss with a loss rate of 0.2% in our experiments.

As shown in Fig. 5, the performance improvement contributed by our algorithm is greater than those proposed in [12] (improved by 50%–70% compared to DEL and 125%–155% compared to EWMA). This indicates that our algorithm is robust in a lossy network environment. Again, in Figs. 4 and 5, AVG-DEV achieves the performance improvement that is very close to that of FA.

Figs. 6–8 show the comparisons of various algorithms based on the unnecessary retransmission rate, which is defined as the ratio of the number of unnecessary fast retransmissions to the total number of packets transmitted. When the path delay is changed every 50 ms, as exhibited in Fig. 6, our algorithm effectively reduces the unnecessary retransmission rate to 15%–40% of that of DEL and 6%–15% of that of EWMA.

Fig. 7 shows the performance of various algorithms running in an environment with fewer reordering events. Our algorithm still outperforms others by reducing the unnecessary retransmission rate, though its performance superiority diminishes. By introducing the packet loss with a loss rate of 0.2%, the connection throughput is dropped significantly, but the unnecessary retransmission rate is more or less the same. Our algorithm achieves a drastic reduction in the false fast retransmissions as shown in Fig. 8. This is attributed to the ability of our proposed algorithm to correctly identify a larger portion of the reordering events. This means that most of the reordering events will not trigger the false fast retransmissions.

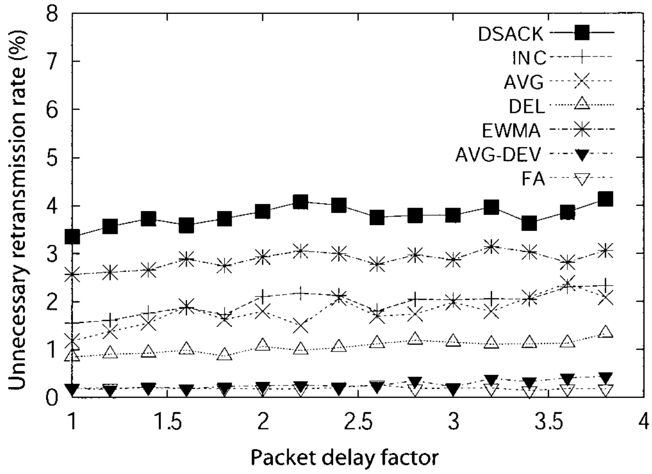


Fig. 8. Unnecessary retransmission rate against packet delay factor. Path delay is changed every 50 ms. Packet loss rate is 0.2%.

B. Multipath Network

Multipath routing has recently been found to be an effective traffic engineering technique to improve network throughput and reduce network load fluctuation [2]–[4]. However, packets belonging on the same TCP flow and travelling on different paths may arrive out-of-order at the destination. This results in triggering fast retransmissions frequently and unnecessarily because of the inability for TCP to distinguish between packet reordering and packet loss. Thus, some network bandwidth is wasted while the connection throughput is kept to be small. To make packet-based multipath routing to become a practical traffic engineering technique for core networks, the problem of performance degradation due to out-of-order packet arrivals has to be alleviated. In this subsection, we demonstrate the ability of our proposed algorithm to resolve the aforementioned problem by comparing its performance to other existing approaches in a

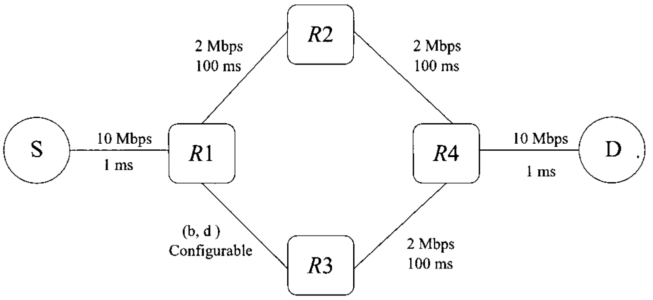


Fig. 9. Multipath network topology.

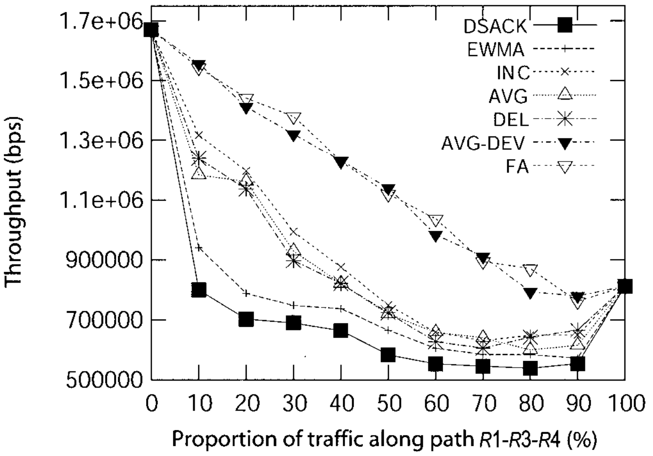


Fig. 10. Throughput against proportion of traffic through path $R1$-$R3$-$R4$. No packet loss over link S-$R1$.

multipath network environment.

Instead of using the default round-robin forwarding algorithm in *ns-2*, we have implemented the weighted round-robin traffic splitting algorithm. It allows any feasible load distributions along different paths. A multipath network topology, as shown in Fig. 9, consists of two end-systems (S and D) and four routers ($R1$–$R4$). A TCP flow from S to D lasting for 1000 seconds is simulated.

In our simulation results reported here, the configurable bandwidth and delay of $R1$-$R3$ link are 1 Mbps and 250 ms, respectively. The TCP traffic proportion between two paths ($R1$-$R2$-$R4$ and $R1$-$R3$-$R4$) is adjustable. We change the proportion of the flow travelling along $R1$-$R2$-$R4$ from 0% to 100%. Meanwhile, the $R1$-$R3$-$R4$ proportion decreases from 100% to 0%.

When the proportion of the flow travelling along $R1$-$R2$-$R4$ is 0% or 100%, the TCP flow travels on a single path. All packets are thus arrived at the destination in the same order as they are sent. Since the techniques under study merely differ in how packet reordering is handled, they therefore have the same connection throughput.

By changing the proportion of the flow to be routed on these two paths, various levels of packet reordering can be simulated. The methods perform differently in these reordering scenarios as shown in Figs. 10 and 11. Again, the results reported here are the averages over 15 runs. The throughput performance of an "ideal" algorithm to deal with out-of-order packet arrivals would show a straight line joining the two points corresponding to the connection throughput when all packets travel on either one of these two paths. This can be used to calibrate the quality of all techniques under consideration.

Table 2. Comparisons of the unnecessary retransmission rates.

Method	No packet loss	With 0.3% packet loss
DSACK	3.078	3.423
INC	1.146	1.569
AVG	0.935	1.535
DEL	0.993	1.327
EWMA	2.212	2.382
AVE-DEV	0.348	0.427
FA	0.257	0.392

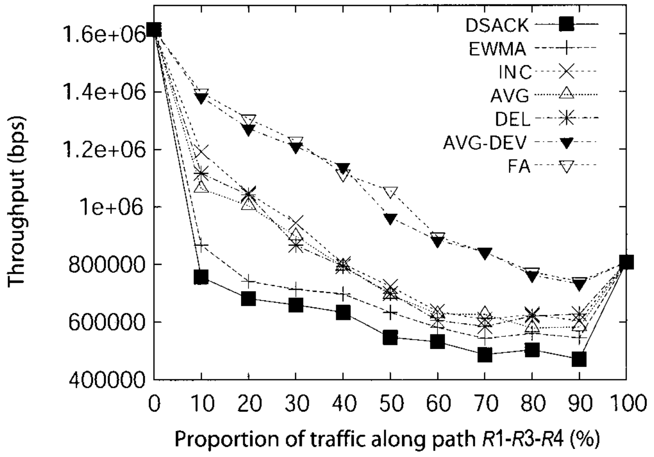


Fig. 11. Throughput against proportion of traffic through path $R1$-$R3$-$R4$. Packet loss rate over link S-$R1$ is 0.3%.

Fig. 10 suggests that our algorithm improves the throughput by 35%–45% compared to DEL, INC, and AVG on the average. When it is compared to EWMA, it improves the throughput by 65%. Fig. 11 depicts the throughput curves when a loss rate of 0.3% over Link S-$R1$ is introduced. Again, our algorithm outperforms EWMA, DEL, INC, and AVG.

Table 2 compares the unnecessary retransmission rates among various algorithms. It shows that our algorithm is very effective in reducing the number of the unnecessary fast retransmissions in a multipath network environment.

C. Overhead Comparison: Our Method and RR-TCP

As shown in the foregoing subsections, the performance of our method is comparable to that of RR-TCP. In fact, the connection throughput of AVG-DEV is very close to that of FA. Its unnecessary retransmission rate is almost identical to that of FA. However, FA needs to maintain a histogram of the lengths of the reordering events. The histogram records up to 1000 reordering events. Each record consists of a four-byte timestamp and a four-byte pointer. Thus, the histogram requires up to 8000 bytes of memory space. It is scanned and updated for every detected reordered packet. On the contrary, what our algorithm requires is a few (less than 20) counters. The values stored in these counters are updated using a set of simple arithmetic formulae as described in Section III. These counters require no complicated data structures. They are used to store integers and floating-point numbers only. In terms of the

Table 3. Storage space and execution time comparisons.

Method	Storage space	Time spent on *recv*
AVE-DEV	< 200 bytes	0.52 seconds
FA	≥ 8000 bytes	1.58 seconds

ns-2 runtime performance, we have measured the times spent on *TcpSack1Agent::recv* function, which is called when an ACK is received. On the average over 15 runs, FA spends 1.58 seconds while AVG-DEV spends 0.52 seconds.

Table 3 summarizes the storage and computational overheads for FA and AVG-DEV. It shows that our algorithm has negligible overheads when compared to FA. Our algorithm meets the design objective by achieving the performance improvement that is comparable to the best known algorithm so far, without paying substantial storage and computational costs.

D. Discussion on Fairness Issue

When compared with TCP Reno [10], [19] (the most popular TCP variant), there are three major changes. First, our proposed algorithm is installed with an adaptive algorithm to dynamically determine an appropriate value of *dupthresh* based on the current network condition, whereas TCP Reno is always associated *dupthresh* with a fixed value (three by default). Second, when a spurious fast retransmission is detected, our algorithm applies the approach used in [12] that a sender uses slow start to increase the size of the congestion window to the value just before a fast retransmission has falsely occurred. TCP Reno has no mechanism to detect any occurrences of spurious retransmission. Third, our algorithm uses a variant of the limited transmit algorithm described in [12] to allow the sender to transmit a new segment upon the receipt of the first two duplicate ACKs and every two duplicate ACKs received afterwards, while TCP Reno does not send any new segments until an ACK for a new segment arrives.

When the sender receives a number of duplicate ACKs, our algorithm does allow the sender to transmit a new segment upon the receipt of the first two duplicate ACKs and every two duplicate ACKs received afterwards. This not only maintains ACK-clocking but also the transmission rate of the sender has been halved as an indication of network congestion. When an ACK for a new segment is received before a fast retransmission is triggered, the longest burst that can be injected into the network at once cannot be longer than half of *dupthresh*. Since an effective *dupthresh* is always less than the size of the congestion window, the sender does not increase the consumption of the network bandwidth. Moreover, unlike TCP Reno, the limited transmit extension does help to reduce burst lengths in order to alleviate some adverse effects due to the potential burst injection when packet reordering exists.

When a packet drop occurs, our algorithm initiates a fast retransmission and halves the size of the congestion window when *dupthresh* duplicate ACKs have received. Though the sender may delay retransmitting the lost segment, about half of *dupthresh* new segments have been sent since the latest train of duplicate ACKs has occurred. Thus, the effective transmission rate of the sender has already been halved while ACK-clocking still maintains.

Combining, our proposed algorithm does permit the sender to interpret any segment loss as an indication of network congestion and reduce the size of the congestion window by at least in half. Since our algorithm adapts the same congestion avoidance mechanism as TCP Reno, the sender increases the size of the congestion window by at most one segment per round-trip time. Hence, a TCP connection established by using our proposed algorithm is a conformant TCP connection [7]. When our algorithm is globally deployed, its flow would not increase its throughput with aggressive manners, break fair sharing with other conformant TCP flows, and cause congestion collapse from undelivered packets [7].

V. CONCLUSIONS

We have proposed a simple method to improve the robustness of TCP on the network paths with persistent packet reordering. The value of *dupthresh* is adaptively adjusted with the EWMA and the mean deviation of the reordering lengths. We have also developed a mechanism which exerts a reasonable upper bound on *dupthresh* to avoid *dupthresh* too high to trigger a retransmission timeout. In addition, our algorithm includes a mechanism to exponentially reduce *dupthresh* when the retransmission timer expires. Our algorithm is engineered to avoid a certain portion of the false fast retransmissions so as to strike a balance between the avoidance of spurious retransmissions due to packet reordering and timely retransmissions of lost packets.

Compared to the previous work, our proposed algorithm is simple, implementation-friendly, and effective. It achieves a significant performance improvement, without the need of adding timers or maintaining complex data structures for storing the reordering information. It meets the design objective by achieving the performance improvement that is comparable to the best known algorithm so far, without paying substantial storage and computational costs.

The simulation results show that our method.

1. Significantly improves the protocol throughput significantly, as compared to methods proposed in [12];
2. substantially reduces the number of unnecessary retransmissions; and
3. achieves the performance comparable to those in [13] with less overheads.

There are several possible extensions to our work, some of which are listed below.

1. Revise the estimators for RTO and RTT to improve the stability of the *dupthresh* estimator;
2. devise an adaptive mechanism to dynamically estimate the values of all pre-defined constants used in our algorithm; and
3. implement and examine the performance of our proposed algorithm on the experimental testbeds.

ACKNOWLEDGEMENT

We would like to thank Ethan Blanton and Ming Zhang for releasing their *ns-2* codes to our studies. We are grateful to Nianen Chen for conducting some of the simulation experiments and engaging in fruitful discussions on our algorithm. Last, but not least, we would like to express our gratitude to

Victor O. K. Li and the anonymous reviewers for their valuable comments and suggestions which assisted us in improving the quality of this paper.

REFERENCES

[1] J. Bennett, C. Partridge, and N. Shectman, "Packet reordering is not pathological network behavior," *IEEE/ACM Trans. Networking*, vol. 7, no. 6, pp. 789–798, Dec. 1999.

[2] K.-C. Leung and V. O. K. Li, "Generalized load sharing for packet-switching networks I: Theory and packet-based algorithm," *IEEE Trans. Parallel Dist. Syst.*, to be published.

[3] K.-C. Leung and V. O. K. Li, "Generalized load sharing for packet-switching networks II: Flow-based algorithms," *IEEE Trans. Parallel Dist. Syst.*, to be published.

[4] N. F. Maxemchuk, "Dispersity routing in high-speed networks," *Computer Networks ISDN Syst.*, vol. 25, no. 6, pp. 645–661, Jan. 1993.

[5] S.-J. Lee and M. Gerla, "AODV-BR: Backup routing in ad hoc networks," in *Proc. IEEE WCNC 2000*, Chicago, IL, USA, vol. 3, 26–29 Sept. 2000, pp. 1311–1316.

[6] P. P. Pham and S. Perreau, "Performance analysis of reactive shortest path and multi-path routing mechanism with load nalance," in *Proc. IEEE INFOCOM 2003*, San Francisco, CA, USA, vol. 1, 30 Mar.–3 Apr. 2003, pp. 251–259.

[7] S. Floyd and K. Fall, "Promoting the use of end-to-end congestion control in the Internet," *IEEE/ACM Trans. Networking*, vol. 7, no. 4, pp. 458–472, Aug. 1999.

[8] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An extension to the selective acknowledgement (SACK) option for TCP," *RFC 2883*, Network Working Group, Internet Engineering Task Force, July 2000.

[9] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options," *RFC 2018*, Network Working Group, Internet Engineering Task Force, Oct. 1996.

[10] M. Allman, V. Paxson, and W. Stevens, "TCP congestion control," *RFC 2581*, Network Working Group, Internet Engineering Task Force, Apr. 1999.

[11] V. Jacobson, "Modified TCP congestion avoidance algorithm," *end2end-interest mailing list*, 30 Apr. 1990.

[12] E. Blanton and M. Allman, "On making TCP more robust to packet reordering," *Computer Commun. Rev.*, vol. 32, no. 1, pp. 20–30, Jan. 2002.

[13] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: A reordering-robust TCP with DSACK," in *Proc. IEEE ICNP 2003*, Atlanta, GA, USA, 4–7 Nov. 2003, pp. 95–106.

[14] V. Paxson, "End-to-end Internet packet dynamics," *IEEE/ACM Trans. Networking*, vol. 7, no. 3, pp. 277–292, June 1999.

[15] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka, "TCP-PR: TCP for persistent packet reordering," in *Proc. IEEE ICDCS 2003*, Providence, RI, USA, 19–22 May 2003, pp. 222–231.

[16] R. Ludwig and R. H. Katz, "The Eifel algorithm: Making TCP robust against spurious retransmissions," *Computer Commun. Rev.*, vol. 30, no. 1, pp. 30–36, Jan. 2000.

[17] Y. Lee, I. Park, and Y. Choi, "Improving TCP performance in multipath packet forwarding networks," *J. Commun. Networks*, vol. 4, no. 2, pp. 148–157, June 2002.

[18] K.-C. Leung and V. O. K. Li, "Flow assignment and packet scheduling for multipath routing," *J. Commun. Networks*, vol. 5, no. 3, pp. 230–239, Sept. 2003.

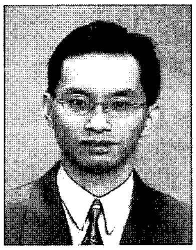
[19] V. Jacobson, "Congestion avoidance and control," *Computer Commun. Rev.*, vol. 18, no. 4, pp. 314–329, Aug. 1988.

[20] L. Kleinrock, *Queueing Systems (Volume I: Theory)*, John Wiley & Sons, 1975.

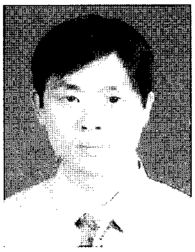
[21] D. G. Luenberger, *Introduction to Dynamic Systems: Theory, Models, and Applications*. John Wiley & Sons, New York, 1979.

[22] W. Theilmann and K. Rothermel, "Dynamic distance maps of the Internet," in *Proc. IEEE INFOCOM 2000*, vol. 1, Tel Aviv, Israel, 26-30 Mar. 2000, pp. 275–284.

[23] R. V. Hogg and E. A. Tanis, *Probability and Statistical Inference*, 5th ed, Prentice Hall, 1996.



Ka-Cheong Leung was born in Hong Kong in 1972. He received the B.Eng. degree in Computer Science from the Hong Kong University of Science and Technology, Hong Kong, in 1994, the M.Sc. degree in Electrical Engineering (Computer Networks) and the Ph.D. degree in Computer Engineering from the University of Southern California, Los Angeles, California, USA, in 1997 and 2000, respectively. He worked as Senior Research Engineer at Nokia Research Center, Nokia Inc., Irving, Texas, USA from 2001 to 2002. He was Assistant Professor at the Department of Computer Science at Texas Tech University, Lubbock, Texas, USA between 2002 and 2005. Since June 2005, he has been with the University of Hong Kong, Hong Kong, China, where he is Visiting Assistant Professor at the Department of Electrical and Electronic Engineering. His research interests include routing, congestion control, and quality of service guarantees in high-speed communication networks, content distribution, high-performance computing, and parallel applications. He is listed in the 60th (2006) Edition of Marquis Who's Who in America.



Changming Ma received the M.S. degree in mechanical engineering (CAD/CG) from Zhejiang University, Hangzhou, China, in 1997. Presently, he is a Ph.D. candidate in Computer Science at Texas Tech University, Lubbock, Texas, USA. His current research interests include parallel and distributed systems, high level language design, and automatic programming.