

TCP Karak: A New TCP AIMD Algorithm Based on Duplicated Acknowledgements for MANET

Wesam A. Almobaideen, Njoud O. Al-maitah

Computer Science Department, King Abdulla II School for Information Technology, The University of Jordan, Amman, Jordan
Email: almobaideen@inf.ju.edu.jo

Received 16 August 2014; revised 31 August 2014; accepted 10 September 2014

Copyright © 2014 by authors and Scientific Research Publishing Inc.

This work is licensed under the Creative Commons Attribution International License (CC BY).

<http://creativecommons.org/licenses/by/4.0/>



Open Access

Abstract

Transmission Control Protocol (TCP) performance over MANET is an area of extensive research. Congestion control mechanisms are major components of TCP which affect its performance. The improvement of these mechanisms represents a big challenge especially over wireless environments. Additive Increase Multiplicative Decrease (AIMD) mechanisms control the amount of increment and decrement of the transmission rate as a response to changes in the level of contention on routers buffer space and links bandwidth. The role of an AIMD mechanism in transmitting the proper amount of data is not easy, especially over MANET. This is because MANET has a very dynamic topology and high bit error rate wireless links that cause packet loss. Such a loss could be misinterpreted as severe congestion by the transmitting TCP node. This leads to unnecessary sharp reduction in the transmission rate which could degrade TCP throughput. This paper introduces a new AIMD algorithm that takes the number of already received duplicated ACK, when a timeout takes place, into account in deciding the amount of multiplicative decrease. Specifically, it decides the point from which Slow-start mechanism should begin its recovery of the congestion window size. The new AIMD algorithm has been developed as a new TCP variant which we call TCP Karak. The aim of TCP Karak is to be more adaptive to mobile wireless networks conditions by being able to distinguish between loss due to severe congestion and that due to link breakages or bit errors. Several simulated experiments have been conducted to evaluate TCP Karak and compare its performance with TCP NewReno. Results have shown that TCP Karak is able to achieve higher throughput and goodput than TCP NewReno under various mobility speeds, traffic loads, and bit error rates.

Keywords

TCP Congestion Control, Additive Increase Multiplicative Decrease, Mobile Ad Hoc Networks, Duplicated Acknowledgement

1. Introduction

Transmission Control Protocol (TCP) is one of the most important transport layer protocols over the transport layer. TCP service model includes reliable transmission, flow control, and congestion control. Packet or segment loss has been adopted in TCP to indicate congestion in the network core and for triggering retransmission of lost data packets in order to achieve guaranteed delivery. Congestion control is done by adapting the transmission rate, represented by a congestion window size (cwnd), to the available network capacity. TCP sets a retransmission timeout period (RTO) within which it expects to receive an acknowledgment (ACK) of a transmitted data segment [1] [2].

TCP interprets the absence of ACK, for an RTO period, as a loss of that data segment. Accordingly, the expiration of the RTO causes packet retransmission, duplicates the value of RTO, and decreases the cwnd to one packet. The decrement is referred to as Multiplicative Decrease (MD) since it sharply lowers the value of cwnd to one packet as a precaution of further packet dropping in the network core. TCP sender then uses slow start algorithm to increase the cwnd size to what is called the slow start threshold (ssthresh) which equals half its current cwnd value [1] [2].

Slow start is a binary exponential growth of the cwnd. Additive Increase (AI) is then used to carefully recover the size of cwnd by adding one extra segment size to its current value should the whole previous cwnd size gets acknowledged [1]. Out of order delivery of packets causes duplicated ACKs for each packet that reaches the destination before its predecessor. Since the estimation of an accurate RTO is by no means an easy task, duplicated ACKs could also be used as a sign of packet loss which is known as fast retransmit mechanism [3].

Receiving one or two duplicated ACKs could be interpreted as a delayed packet delivery instead of a lost one. TCP sender waits for three duplicative ACKs in order to invoke fast retransmit mechanism. This provides a kind of assurance that the packet is really lost and should be retransmitted immediately without the need to wait until RTO expiration. Retransmitting a lost packet via fast retransmit mechanism is adopted by many TCP variant, such as TCP NewReno, and is followed by invoking fast recovery mechanism [4]. Fast recovery sets the cwnd to ssthresh, waits the ACK for all segments sent after the lost segment and before receiving the first duplicated ACK, and then starts the AI, also known as congestion avoidance, phase [1] [4].

Mobile Ad-hoc Network (MANET) is characterized by its very dynamic topology due to the mobility of its wireless nodes. Such a very flexible and infrastructure-less network serves a wide range of applications such as rescue missions, military operations, and in any other situation where infrastructure establishment is either very expensive or quite impossible [5] [6].

Wireless networks, such as MANET and wireless sensor networks (WSN), are also characterized by high bit error rate (BER) channels [8]. TCP faces some challenges over these networks since it is not inherently designed to adapt for such environments properties. The performance of TCP degrades with high mobility and bit error rate since it reduces the transmission rate when a segment is lost due to link breakages or bit errors and cannot distinguish between these two cases and severely congested routers buffers [7] [8].

In this paper we proposed and evaluate a modified version of TCP NewReno which we called TCP Karak. TCP Karak relies on the reception of a number of duplicated ACKs that are not enough to trigger fast retransmit, to improve the AIMD mechanism when an RTO expires. TCP Karak reduces the level of decrement when a timeout occurs after the reception of one duplicated ACK and reduces it to a greater extent in case of a timeout after two duplicated ACKs. This is done in order to allow TCP Karak to distinguish between packet loss due to severe congestion and that results due to link breakage or bit error.

The rest of this paper is organized as follows. Some of the related work is presented in Section 2. Section 3 illustrates the details TCP Karak AIMD algorithm. Section 4 presents the discussion of results of evaluating TCP Karak in comparison with TCP NewReno. Finally, Section 5 concludes the paper.

2. Related Works

Many previous researches have mentioned the need to modify TCP AIMD to make it more properly adaptive to the characteristic of new paradigms [9]-[12]. These paradigms could represent new environment such as MANET, WSN, and Content-Centric Network (CCN).

In [9] an end-to-end congestion control algorithm has been proposed. The main idea is to use adaptive setting for the additive increase/multiplicative decrease (AIMD) congestion control scheme. This adaptivity means that parameters can change dynamically as the current network conditions change. Results have shown that the pro-

posed algorithm reach almost optimal utilization in case of a steady state network and responds quickly to links bandwidth changes. The algorithm is not TCP-compatible and needs to be studied over more complex network topologies.

A transport mechanism that is adapted to video flows was presented in [10]. It is called Q-AIMD for video quality AIMD. Q-AIMD enables fairness in video quality when transmitting multiple video flows and improves the overall video quality for all flows, especially when the transmitted videos provide various types of content with different spatial resolutions. Q-AIMD lessens congestion events by decreasing the video quality, and accordingly the bit rate, whenever congestion occurs. Q-AIMD has been evaluated with different video contents and spatial resolutions and has shown an improved overall video quality compared to other throughput-based congestion control mechanism.

In [11] different factors such as the way CCN routers are deployed, the popularity of contents, or the capacity of links have been taken into consideration in observing the performance of AIMD when used over a CCN network. Results advocate the need to design a proper congestion control to avoid less popular contents from becoming hardly accessible in tomorrow's Internet.

A fuzzy inference system that is based on factors such as the expected throughput and actual throughput has been presented in [12] to dynamically adjust TCP congestion window size over MANET. Extensive simulation experiments have shown multiple concurrent flows significantly affect TCP performance over MANET and that the proposed scheme achieves improvement in performance compared to other TCP variants.

In [13] a framework for AIMD in TCP has been developed in order to analyze problems of adapting TCP AIMD algorithms over wireless networks. The framework presents a systematic analysis of existing AIMD-based TCP variants, classifies them into two main streams, and develops a generic expression that covers the rate adaptation processes of both approaches. It further identifies a new approach in enhancing the performance of TCP and assists in the design of new TCP variants. A tax-rebate approach was proposed as an approximation of the compensation scheme, and used to enhance the AIMD-based TCP variants to offer unified solutions for effective congestion control, sequencing control, and error control. A set of simulation experiments have been conducted to examine their performance under various network scenarios. In most scenarios, significant performance gains have been achieved.

TCP Karak does not rely on any explicit feedback from the network core; it requires only the sender side adaptation. TCP Karak also works on top of any underlying routing protocol such as AODV [14].

3. TCP Karak Algorithm Description

The main idea of TCP Karak is to utilize the information provided by the reception of duplicated acknowledgement events to make a decision on how much aggressive the multiplicative decrease should be. This is to define the value of the congestion window from where Slow-start should begin its recovery of the cwnd over again after a timeout event. An occurrence of a timeout event without receiving any duplicated ACK beforehand indicates a very severe congestion in the network core. In this case, an aggressive multiplicative decrease such as the one followed by TCP Tahoe, which sets the cwnd to one segment, could be used [1].

Receiving duplicated acknowledgements is considered as a sign of light congestion. We think that the number of received duplicated acknowledgments provides valuable information about the level of contention in the network, since the more duplicated ACKs received, the less the level of congestion is assumed to be along the path to the destination. If the source node has received one duplicated ACK before the expiration of the RTO, then a less aggressive multiplicative decrease value could be adopted to lower the current congestion window. Should the source node receive two duplicated ACKs before the RTO expiration, then this justifies a lesser value of a multiplicative decrease which allows for faster recovery of the cwnd value.

Receiving three duplicated ACKs before the expiration of the RTO activates the well-known fast retransmit procedure and moves the congestion window to half its current value to follow the fast recovery mechanism steps before activating the, afterward, additive increase procedure. See **Table 1** for a pseudo-code description of TCP Karak.

One can argue that the expiration of RTO is, in the first place, an indication of severe congestion and, accordingly, it does not matter how many duplicated ACKs have been received before. We argue that this is not generally true and specially over MANET due to more than one reason. Firstly, the expiration of RTO could happen when a packet is lost while using a small congestion window or at the end of a large congestion window size. In both cases there will not be enough transmitted packets in the pipe to trigger a fast retransmit event. Secondly,

Table 1. TCP Karak pseudo-code.

```

Initially:
    cwnd = 1
    ssthresh = infinite
    Dup_ACK = 0
New ACK arrives:
    If (cwnd < ssthresh)
        /*Slow start*/
        cwnd = cwnd+1
    Else
        /*Congestion avoidance*/
        cwnd = cwnd+1/cwnd
Duplicated ACK arrives:
    Dup_ACK =Dup_ACK+1
    If (Dup_ACK ==3)
        /*Fast retransmit*/
        cwnd =cwnd /2;
Timeout:
    /* Multiplicative decrease */
    /* check dup_Ack */
    If (Dup_ACK==1)
        cwnd=cwnd/8
    Else if (Dup_ACK ==2)
        cwnd = cwnd /4
    Else
        /*Dup_Ack is zero*/
        cwnd = 1

```

and due to the very dynamic topology of MANET which causes frequent link breakages and due to its environment which is characterized with high bit error rate, multiple packets could be dropped preventing some data packet from reaching the destination. It could also prevent ACK packets from getting back to the source node. In both cases, this results in triggering a timeout event after the expiration of the RTO.

In such cases the right action to take is to recover the loss by quick transmission of the dropped packets as well as other queued packets even though we are unable to invoke fast retransmit. TCP Karak tries to achieve this goal by taking into consideration the number of duplicated ACK received before the expiration of the RTO in order to control the level of decrement in the congestion window.

Figure 1 and **Figure 2** shows how TCP Karak differentiates between packet loss due to link breakage or bit error and that due to severe congestion. TCP assumes a packet loss results from severe congestion when a timeout event occurs without a reception of any duplicated ACK. Then, it aggressively reacts by multiplicative decrease that returns the sender cwnd to one segment as the starting point of Slow-start algorithm. In the other cases, where a time out event occurs after receiving one or two duplicated ACKs, then TCP Karak reacts less aggressively by reducing the current congestion window to one eighth or one quarter its current size, respectively. In both cases, and since Slow-start is a binary exponential algorithm, the value of cwnd is reduced to the first power of two value higher than the one eighth or one quarter points. Afterward, TCP Karak continues by Slow-start to reach the point of additive increase.

TCP Karak does not depend on any explicit notification mechanism, from the core of the network and needs some modification on the source node. TCP Karak utilizes fast retransmit when three duplicated ACKs are received before the expiration of an RTO.

4. Results Evaluation and Discussion

Several simulated experiments have been conducted to evaluation the performance of TCP Karak and to compare it with TCP NewReno. Performance evaluation has been performed by measuring the throughput and goodput of both protocols. While the throughput gives an important indication of a protocol performance since it measures the amount of data that can be transmitted in a time unit, goodput is more accurate measurement since it excludes the retransmission of erroneous and lost packets and counts only the correctly delivered amount of data.

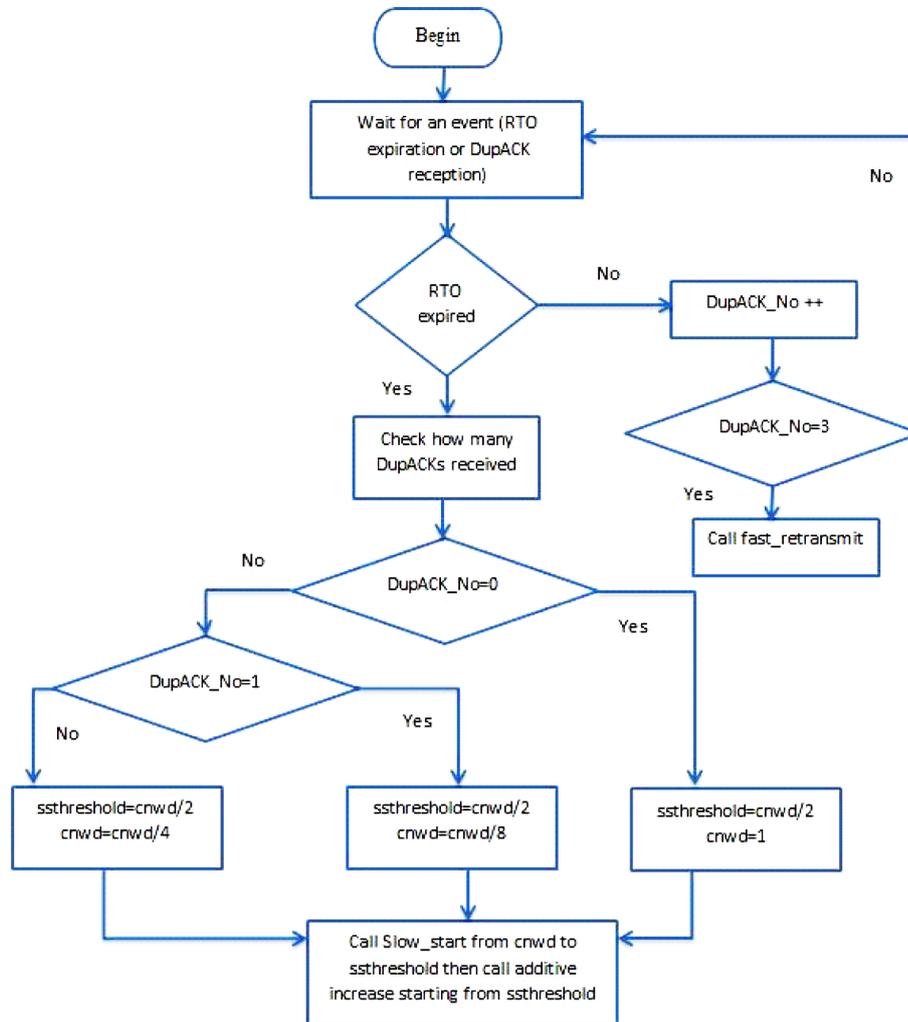


Figure 1. TCP Karak flow chart illustration.

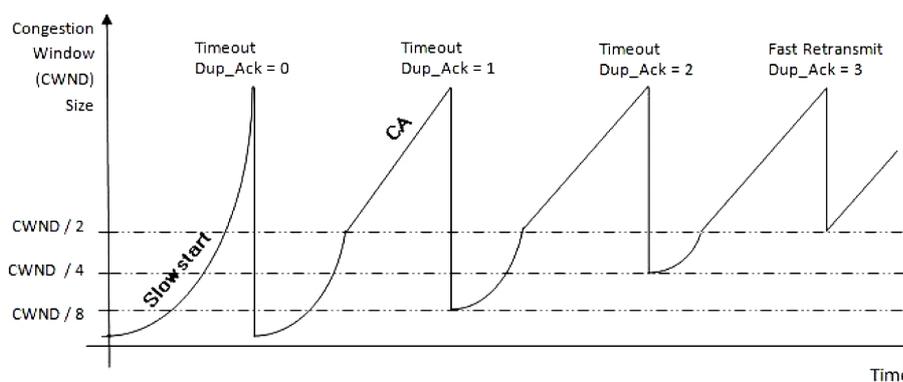


Figure 2. TCP Karak illustration.

Glomosim 2.02 simulation package was used [15]. Different scenarios have been generated assuming 50 nodes in square terrain of 1200 m². The number of source nodes was 25 out of the 50 total nodes. Simulation time was 10 minute. Each simulated scenario has been repeated 20 times with different simulation seeds in order to get confidence in the resulted average. **Table 2** summarizes the simulation parameters used and the adopted values of such parameters.

Table 2. Summary of simulation parameters and their values.

Parameter	Value
TERRAIN-DIMENSIONS	1200 m × 1200 m
NUMBER-OF-NODES	50
NODE-PLACEMENT	RANDOM
MOBILITY-WP-PAUSE	30 S
SIMULATION-TIME	10 M
MOBILITY	RANDOM-WAYPOINT
Transmission Range	250 m
ROUTING-PROTOCOL	AODV
Error Bit Rate (EBR) with varying mobility speed	10^{-7}
Average mobility speed with varying BER	15 m/s

4.1. Results with Various Mobility Speeds

In the first set of simulated experiments, **Table 3** and **Table 4**, nodes were able to move randomly with a speed that varies in different scenarios from 5, 10, 15, 20, 25, and 30 mps with a 30 seconds pause time. The comparison between TCP Karak and TCP NewReno is shown in these two tables which contain figures that measure the throughput and goodput of both protocols as the mobility speed increases and as the number of packets sent by each TCP source increases from 1000, and up to 5000 data packets. This is done in order to evaluate the effect of increased traffic load over the simulated ad hoc network.

Table 3 shows a set of figures that illustrate the evaluation of the throughput of both protocols. The throughput of both protocols decreases as the mobility speed increases due to the effect of raising mobility speed on increasing link breakages and accordingly, higher number of packets gets lost. TCP Karak is able to achieve higher throughput than TCP NewReno for almost all values of mobility speeds and traffic rates. The set of figures in **Table 3** shows that the throughput of TCP Karak becomes more apparently higher than that of TCP NewReno as the mobility speed and traffic rate increases. This is because the number of link breakages increases as the mobility speed increases and as a consequence the number of dropped packets becomes higher.

Sending more packets also increases the number of packets that get affected by link breakages as well as the probability of getting more duplicated ACKs back to the sender node. TCP Karak is able to detect situations where packet loss is due to link breakages and not due to severe congestion by existence of some duplicated ACKs that has arrived before the RTO expiration.

TCP Karak makes the difference in throughput by lowering the amount of multiplicative decrease in cases where one or two duplicated ACK have been received before the RTO expiration. In such cases, TCP Karak is able to reach the fast recovery point faster than backing off to the normal slow start process where only one packet is sent. With the combination of very low mobility speed and traffic rate, we can notice that TCP Karak achieves lower throughput than TCP NewReno does. This is due to that fact that TCP Karak reacts in the two mention cases above by quick recovery of the congestion window, while slowly moving nodes are unable to find, as much quickly, other alternative paths to pass the transmitted packets to the destination.

Table 4 shows a set of figures that presents the evaluation of the Goodput of TCP Karak in comparison with TCP NewReno. The goodput evaluation results in these figures prove that TCP Karak does not get higher throughput without actual delivery of the transmitted data to the destination. As with measured throughput, the goodput of both protocols decreases as the mobility speed increases because of the increasing mobility effect on increasing the number of link breakage and lost packets. TCP Karak achieves higher goodput than TCP NewReno for almost all values of mobility speeds and especially as the traffic increases due to the same justification explain for the throughput results.

The set of figures in **Table 4** shows that with low traffic rates, TCP Karak goodput needs higher mobility speeds to outperform TCP NewReno. This is more apparent than what we can notice in the throughput results in **Table 3**. The reason behind this is that with low mobility speed there would be low number of link breakages and with low traffic rate there is low number of duplicated ACKs. This prevents TCP Karak from taking the advantage over TCP NewReno and makes the performance of the two approaches comparable.

Table 3. Throughput results for various traffic rates and mobility speeds.

Number of Packets Sent	TCP Throughput (Kbps)																					
1000	<p>This graph shows throughput for 1000 packets. TCP New Reno (blue line with asterisks) starts at ~20.5 Kbps at 5 m/s and decreases to ~13.5 Kbps at 30 m/s. TCP Karak (red dashed line with squares) starts at ~18.5 Kbps, peaks at ~20.5 Kbps at 10 m/s, and ends at ~18 Kbps at 30 m/s.</p> <table border="1"> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>20.5</td><td>18.5</td></tr> <tr><td>10</td><td>18.5</td><td>20.5</td></tr> <tr><td>15</td><td>17.5</td><td>19</td></tr> <tr><td>20</td><td>16</td><td>19</td></tr> <tr><td>25</td><td>15</td><td>20.5</td></tr> <tr><td>30</td><td>13.5</td><td>18</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	20.5	18.5	10	18.5	20.5	15	17.5	19	20	16	19	25	15	20.5	30	13.5	18
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	20.5	18.5																				
10	18.5	20.5																				
15	17.5	19																				
20	16	19																				
25	15	20.5																				
30	13.5	18																				
2000	<p>This graph shows throughput for 2000 packets. TCP New Reno starts at ~24 Kbps at 5 m/s and decreases to ~17 Kbps at 30 m/s. TCP Karak starts at ~22 Kbps, peaks at ~25 Kbps at 10 m/s, and ends at ~21 Kbps at 30 m/s.</p> <table border="1"> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>24</td><td>22</td></tr> <tr><td>10</td><td>22</td><td>25</td></tr> <tr><td>15</td><td>20.5</td><td>23</td></tr> <tr><td>20</td><td>18.5</td><td>22</td></tr> <tr><td>25</td><td>18</td><td>24</td></tr> <tr><td>30</td><td>17</td><td>21</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	24	22	10	22	25	15	20.5	23	20	18.5	22	25	18	24	30	17	21
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	24	22																				
10	22	25																				
15	20.5	23																				
20	18.5	22																				
25	18	24																				
30	17	21																				
3000	<p>This graph shows throughput for 3000 packets. TCP New Reno starts at ~24.5 Kbps at 5 m/s and decreases to ~17.5 Kbps at 30 m/s. TCP Karak starts at ~27 Kbps, peaks at ~27.5 Kbps at 10 m/s, and ends at ~23.5 Kbps at 30 m/s.</p> <table border="1"> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>24.5</td><td>27</td></tr> <tr><td>10</td><td>22.5</td><td>27.5</td></tr> <tr><td>15</td><td>20</td><td>24.5</td></tr> <tr><td>20</td><td>19.5</td><td>24</td></tr> <tr><td>25</td><td>18.5</td><td>25.5</td></tr> <tr><td>30</td><td>17.5</td><td>23.5</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	24.5	27	10	22.5	27.5	15	20	24.5	20	19.5	24	25	18.5	25.5	30	17.5	23.5
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	24.5	27																				
10	22.5	27.5																				
15	20	24.5																				
20	19.5	24																				
25	18.5	25.5																				
30	17.5	23.5																				
4000	<p>This graph shows throughput for 4000 packets. TCP New Reno starts at ~24.5 Kbps at 5 m/s and decreases to ~17.5 Kbps at 30 m/s. TCP Karak starts at ~27 Kbps, peaks at ~27.5 Kbps at 10 m/s, and ends at ~23.5 Kbps at 30 m/s.</p> <table border="1"> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>24.5</td><td>27</td></tr> <tr><td>10</td><td>22.5</td><td>27.5</td></tr> <tr><td>15</td><td>20.5</td><td>25.5</td></tr> <tr><td>20</td><td>20</td><td>25.5</td></tr> <tr><td>25</td><td>18.5</td><td>26.5</td></tr> <tr><td>30</td><td>17.5</td><td>23.5</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	24.5	27	10	22.5	27.5	15	20.5	25.5	20	20	25.5	25	18.5	26.5	30	17.5	23.5
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	24.5	27																				
10	22.5	27.5																				
15	20.5	25.5																				
20	20	25.5																				
25	18.5	26.5																				
30	17.5	23.5																				
5000	<p>This graph shows throughput for 5000 packets. TCP New Reno starts at ~23.5 Kbps at 5 m/s and decreases to ~17 Kbps at 30 m/s. TCP Karak starts at ~27.5 Kbps, peaks at ~28.5 Kbps at 10 m/s, and ends at ~24 Kbps at 30 m/s.</p> <table border="1"> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>23.5</td><td>27.5</td></tr> <tr><td>10</td><td>21.5</td><td>28.5</td></tr> <tr><td>15</td><td>20.5</td><td>25.5</td></tr> <tr><td>20</td><td>19.5</td><td>25.5</td></tr> <tr><td>25</td><td>18</td><td>26.5</td></tr> <tr><td>30</td><td>17</td><td>24</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	23.5	27.5	10	21.5	28.5	15	20.5	25.5	20	19.5	25.5	25	18	26.5	30	17	24
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	23.5	27.5																				
10	21.5	28.5																				
15	20.5	25.5																				
20	19.5	25.5																				
25	18	26.5																				
30	17	24																				

Table 4. Goodput results for various traffic rates and mobility speeds.

Number of Packets Sent	TCP Goodput (Kbps)																					
1000	<table border="1"> <caption>Data for 1000 Packets Sent</caption> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>19.5</td><td>17.0</td></tr> <tr><td>10</td><td>17.5</td><td>18.0</td></tr> <tr><td>15</td><td>16.5</td><td>17.0</td></tr> <tr><td>20</td><td>15.0</td><td>17.0</td></tr> <tr><td>25</td><td>14.0</td><td>18.0</td></tr> <tr><td>30</td><td>12.5</td><td>16.0</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	19.5	17.0	10	17.5	18.0	15	16.5	17.0	20	15.0	17.0	25	14.0	18.0	30	12.5	16.0
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	19.5	17.0																				
10	17.5	18.0																				
15	16.5	17.0																				
20	15.0	17.0																				
25	14.0	18.0																				
30	12.5	16.0																				
2000	<table border="1"> <caption>Data for 2000 Packets Sent</caption> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>23.0</td><td>21.0</td></tr> <tr><td>10</td><td>21.5</td><td>23.0</td></tr> <tr><td>15</td><td>19.5</td><td>21.0</td></tr> <tr><td>20</td><td>17.5</td><td>20.0</td></tr> <tr><td>25</td><td>17.0</td><td>21.5</td></tr> <tr><td>30</td><td>16.5</td><td>19.0</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	23.0	21.0	10	21.5	23.0	15	19.5	21.0	20	17.5	20.0	25	17.0	21.5	30	16.5	19.0
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	23.0	21.0																				
10	21.5	23.0																				
15	19.5	21.0																				
20	17.5	20.0																				
25	17.0	21.5																				
30	16.5	19.0																				
3000	<table border="1"> <caption>Data for 3000 Packets Sent</caption> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>24.0</td><td>25.0</td></tr> <tr><td>10</td><td>21.5</td><td>25.0</td></tr> <tr><td>15</td><td>19.5</td><td>22.0</td></tr> <tr><td>20</td><td>18.5</td><td>21.5</td></tr> <tr><td>25</td><td>17.5</td><td>23.0</td></tr> <tr><td>30</td><td>16.5</td><td>20.5</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	24.0	25.0	10	21.5	25.0	15	19.5	22.0	20	18.5	21.5	25	17.5	23.0	30	16.5	20.5
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	24.0	25.0																				
10	21.5	25.0																				
15	19.5	22.0																				
20	18.5	21.5																				
25	17.5	23.0																				
30	16.5	20.5																				
4000	<table border="1"> <caption>Data for 4000 Packets Sent</caption> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>23.5</td><td>25.5</td></tr> <tr><td>10</td><td>21.5</td><td>26.0</td></tr> <tr><td>15</td><td>19.5</td><td>23.0</td></tr> <tr><td>20</td><td>19.0</td><td>23.0</td></tr> <tr><td>25</td><td>17.5</td><td>24.0</td></tr> <tr><td>30</td><td>16.5</td><td>21.0</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	23.5	25.5	10	21.5	26.0	15	19.5	23.0	20	19.0	23.0	25	17.5	24.0	30	16.5	21.0
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	23.5	25.5																				
10	21.5	26.0																				
15	19.5	23.0																				
20	19.0	23.0																				
25	17.5	24.0																				
30	16.5	21.0																				
5000	<table border="1"> <caption>Data for 5000 Packets Sent</caption> <thead> <tr> <th>Mobility Speed (m/s)</th> <th>TCP New Reno (Kbps)</th> <th>TCP Karak (Kbps)</th> </tr> </thead> <tbody> <tr><td>5</td><td>22.5</td><td>26.0</td></tr> <tr><td>10</td><td>20.5</td><td>27.0</td></tr> <tr><td>15</td><td>19.5</td><td>23.5</td></tr> <tr><td>20</td><td>18.5</td><td>23.5</td></tr> <tr><td>25</td><td>17.5</td><td>24.0</td></tr> <tr><td>30</td><td>16.5</td><td>21.5</td></tr> </tbody> </table>	Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)	5	22.5	26.0	10	20.5	27.0	15	19.5	23.5	20	18.5	23.5	25	17.5	24.0	30	16.5	21.5
Mobility Speed (m/s)	TCP New Reno (Kbps)	TCP Karak (Kbps)																				
5	22.5	26.0																				
10	20.5	27.0																				
15	19.5	23.5																				
20	18.5	23.5																				
25	17.5	24.0																				
30	16.5	21.5																				

With relatively high mobility speed and high data rate TCP Karak throughput and goodput becomes apparently higher than that of TCP NewReno also since a lost packet could be at the end of a sender window size. In order for the fast retransmit event to be activated, enough number of duplicated ACKs is needed, three duplicated ACK according to the standard. This could not be possible when the lost packet is positioned near the end of a sender window size. In this case, TCP Karak takes the less number of received duplicated ACK into consideration to decide the extent of multiplicative decrease that is necessary.

4.2. Results with Various Bit Error Rates

Table 5 shows a set of experiments that have been conducted to demonstrate the effect of varying the Bit Error Rate (BER) on the achieved throughput of TCP Karak and comparing this with TCP NewReno. When increasing the bit error rate, more packets are subject to be corrupted. This result in lower throughput for both protocols, since the higher number of packets gets dropped due to bit errors the lower the amount of data that can be transmitted. This is true since TCP does not differentiate between packet loss due to congestion and that due to bit error.

TCP Karak is able to achieve higher throughput than TCP NewReno for almost all values of BERs where the difference in throughput becomes more apparently higher when increasing the traffic rate. When a packet is lost due to bit error, there is good opportunity for the next packet or set of packets to pass through the network to the destination and so causing duplicated ACK to be sent back to the source. A severe congestion on the other hand, results in multi consecutive packets dropping which belong to the same window and so preventing duplicated ACKs from being sent back to the source node.

TCP NewReno waits for a time out event or the reception of three duplicated ACKs to invoke fast retransmit. TCP Karak, additionally, tracks the reception of one or two duplicated ACKs when a time out occurs and exploits this information to decide the amount of multiplicative decrease that it should perform. As the BER increases the difference in throughput between the two protocols increases since larger number of packets get lost. This fact lowers the chance of getting duplicated ACKs back to the source node. Consequently, the achieved goodput of both protocols degrades as the BER increases.

Despite this degradation, TCP Karak is still able to maintain better goodput than TCP NewReno and to make wider difference in performance as the BER increases. This is because TCP Karak is able to be more accurate in differentiating between packet losses due to a severe congestion and that of other reasons, such as bit error, since it considers the number of received duplicated ACK when a time out event occurs.

Table 6 contains experiments figures that present the goodput evaluation TCP Karak and TCP NewReno with various traffic rate and BER. These figures prove that TCP Karak does not get higher throughput without actual delivery of the transmitted data to the destination as most of these goodput evaluation results confirm. As the bit error rate increases the goodput of both protocols decreases. TCP Karak achieves higher goodput than TCP NewReno for almost all values of BER and especially as the traffic rate increases where TCP Karak goodput becomes, progressively, higher than that of TCP NewReno.

The set of figures in **Table 6** shows that with low traffic rates and high BER the goodput of TCP NewReno outperforms that of TCP Karak. This indicates that TCP Karak, in such cases, unnecessarily recovers the sender congestion window size faster than TCP NewReno. Aggressive multiplicative decrease done by TCP NewReno could protect TCP sender from consecutive packet loss due to high bit error rate and especially with low traffic rate.

One final comment on all the obtained results is that there is more fluctuation in the results of TCP Karak than that of TCP NewReno. This could be justified in reference to the way TCP Karak estimates the level of contention in the network by counting the number of duplicated ACKs which could be as the sole factor and in few number of cases too dynamic even though it gives, overall, good estimation of the level of network contention. Further work could be done to refine such an estimation by tacking other relevant factors into account in order to get more stable behavior of TCP Karak algorithm.

5. Conclusions

In this paper, TCP Karak has been presented. TCP Karak is a modification of TCP NewReno and improves its performance using duplicated ACKs received by a TCP source node over MANET. The main aim of TCP Karak is to utilize the arrival of partial number of duplicated ACKs, when a timeout event occurs, in deciding the level

Table 5. Throughput for various traffic rates and BERs.

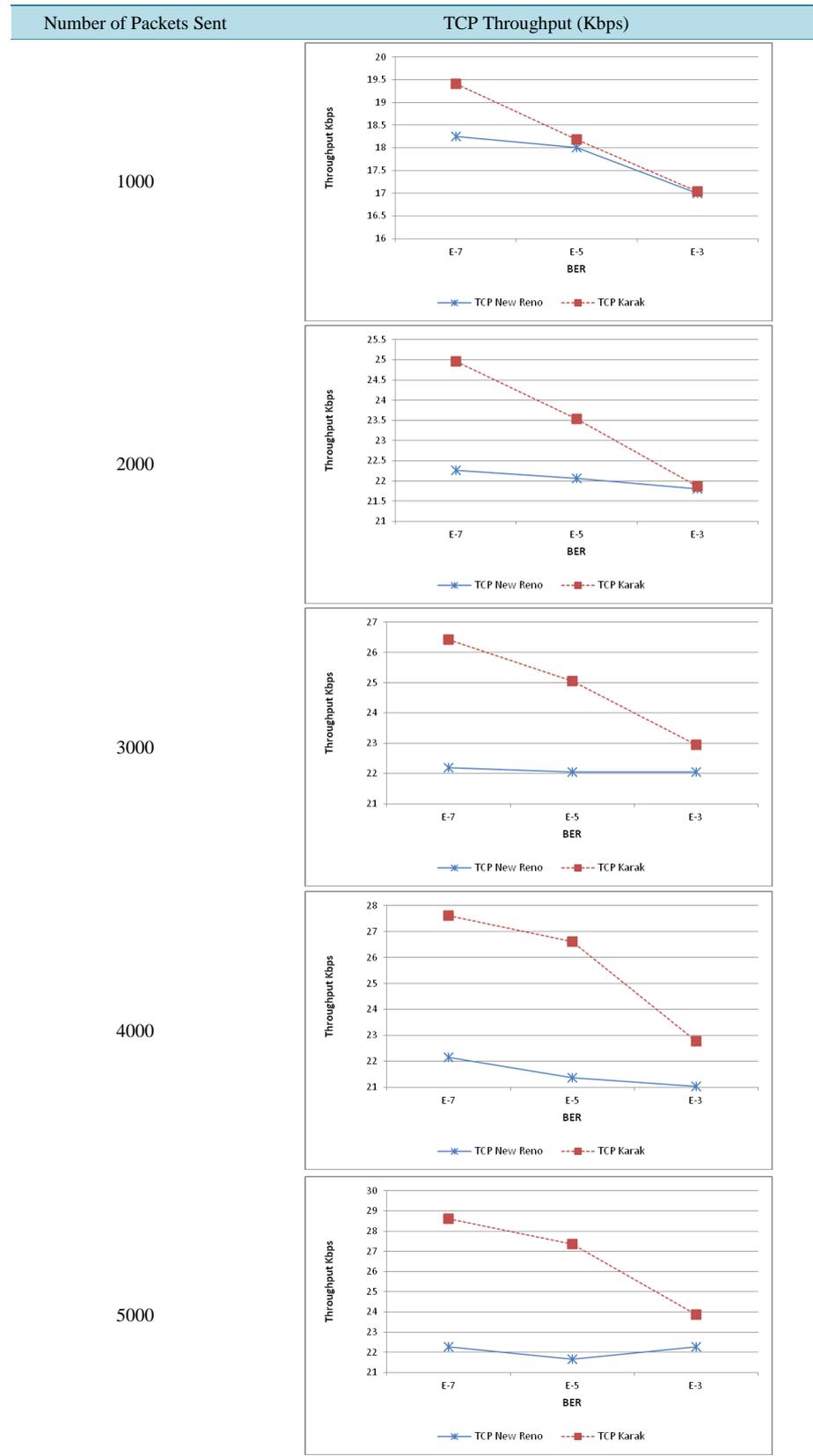
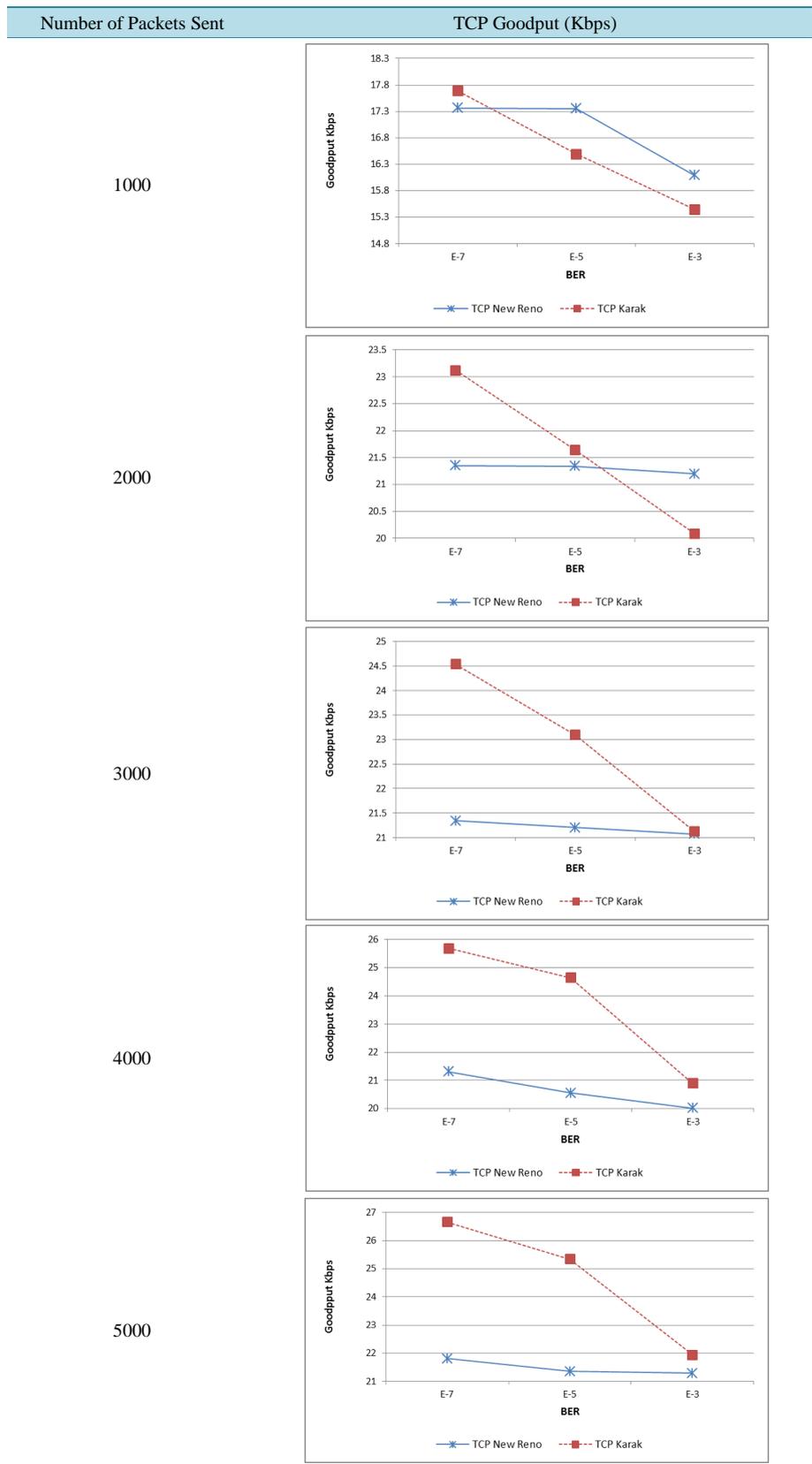


Table 6. Goodput for various traffic rates and BERs.



of multiplicative decrease that should be performed. This results in increased performance by allowing TCP Karak to differentiate between packet loss due to link breakage or bit errors and that due to severe congestion.

In case of packet loss that results from severe congestion and detected by TCP Karak by a time out event without any duplicated ACK, then TCP Karak aggressively reacts by multiplicative decrease which returns the sender congestion window size to one packet as a starting point of Slow-start. In the other cases, where a time out event occurs after receiving one or two duplicated ACKs, then TCP Karak reacts less aggressively by reducing the current congestion window to be one eighth or one quarter the current congestion window size, respectively. Then it continues by Slow-start to reach the point of additive increase.

Results of simulated experiments show that TCP Karak has achieved better throughput and goodput than TCP NewReno under various traffic loads, mobility speeds, and bit error rates.

References

- [1] Allman, M., Paxson, V. and Stevens, W. (1999) TCP Congestion Control. IETF RFC 2581.
- [2] Lai, C., Leung, K.-C. and Li, V.O.K. (2010) Enhancing Wireless TCP: A Serialized-Timer Approach. 2010 *Proceedings IEEE INFOCOM*, San Diego, 14-19 March 2010, 1-5.
- [3] Psaras, I., Tsaoussidis, V. and Mamatas, L. (2005) CA-RTO: A Contention-Adaptive Retransmission Timeout. 14th *IEEE International Conference on Computer Communications and Networks*, **32**, 174-184.
- [4] Henderson, T., Floyd, S., Gurtov, A. and Nishida, Y. (2012) The NewReno Modification to TCP's Fast Recovery Algorithm, RFC 6582.
- [5] Almobaideen, W., Al-Soub, R. and Sleit, A. (2013) MSDM: Maximally Spatial Disjoint Multipath Routing Protocol for MANET. *Communications and Networks*, **5**, 316-322. <http://dx.doi.org/10.4236/cn.2013.54039>
- [6] Qaddoura, E., Almobaideen, W. and Omari, A. (2006) Distributed Clusterhead Architecture for Mobile Ad Hoc Networks. *Journal of Computer Science*, **2**, 583-588. <http://dx.doi.org/10.3844/jcssp.2006.583.588>
- [7] Casetti, C., Gerla, M., Mascolo, S., Sanadidi, M.Y. and Wang, R. (2001) TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. *Proceedings of ACM MOBICOM*, 287-297.
- [8] Qaddoura, E., Daraiseh, A., Almobaideen, W., Muammar, R. and Al-Walaie, S. (2006) TCP Optimal Performance in Wireless Networks Applications. *Journal of Computer Science*, **2**, 455-459. <http://dx.doi.org/10.3844/jcssp.2006.455.459>
- [9] Kesselman, A. and Mansour, Y. (2003) Adaptive AIMD Congestion Control. *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*.
- [10] Tuan Tran Thai, Changuel, N., Kerboeuf, S., Fauchaux, F., Lochin, E. and Lacan, J. (2013) Q-AIMD: A Congestion Aware Video Quality Control Mechanism. 20th *International Workshop on Packet Video (PV)*, **1**, 12-13 <http://dx.doi.org/10.1109/PV.2013.6691457>
- [11] Saucez, D., Grieco, L. and Barakat, C. (2012) AIMD and CCN: Past and Novel Acronyms Working Together in the Future Internet. *CSWS'12 Proceedings of the 2012 ACM Workshop on Capacity Sharing*, 21-26.
- [12] Sreenivas, B., Bhanu Prakash, G. and Ramakrishnan, K. (2012) An Adaptive Congestion Control Technique for Improving TCP Performance over Ad-Hoc Networks. *Elixir International Journal*, **44**, 7391-7395.
- [13] Lai, C., Leung, K.C. and Li, V.O.K. (2013) Design and Analysis of TCP AIMD in Wireless Networks. 2013 *IEEE Wireless Communications and Networking Conference (WCNC)*, 1422-1427.
- [14] Almobaideen, W., Al-Khateeb, D., Sleit, A., Qataweh, M., Al-Khdour, R. and Abu Hafeeza, H. (2013) Improved Stability Based Partially Disjoint AOMDV. *International Journal of Communications, Networks, and System Sciences*, **6**, 244-250. <http://dx.doi.org/10.4236/ijcns.2013.65027>
- [15] Zeng, X., Bagrodia, R. and Gerla, M. (1998) GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks. *PADS 98 Proceedings of Twelfth Workshop on Parallel and Distributed Simulation*, 154-161.

Scientific Research Publishing (SCIRP) is one of the largest Open Access journal publishers. It is currently publishing more than 200 open access, online, peer-reviewed journals covering a wide range of academic disciplines. SCIRP serves the worldwide academic communities and contributes to the progress and application of science with its publication.

Other selected journals from SCIRP are listed as below. Submit your manuscript to us via either submit@scirp.org or [Online Submission Portal](#).

