Experimental Analysis of TCP Behaviors against Bursty Packet Losses Caused by Transmission Interruption

Weikai Wang, Celimuge Wu, Satoshi Ohzahata, Toshihiko Kato
Graduate School of Information Systems
University of Electro-Communications
Chofu-shi, Tokyo, Japan
e-mail: *ohigai@net.is.uec.ac.jp, clmg@is.uec.ac.jp, ohzahata@is.uec.ac.jp, kato@ is.uec.ac.jp*

Abstract— **Although TCP was originally designed to provide the reliable data transfer over the Internet, packet losses detected in TCP are considered as an indication of network congestion due to the high quality of data transmission provided by recent transmission technologies and media access control technologies. However, packet losses can be caused by transmission interruptions such as handoffs in the mobile networks and protection switching in the transport networks. These packet losses are bursty because the transmission interruptions continue for tens of miliseconds through several seconds. In this paper, we describe the experimental analysis of TCP behaviors by inserting errors such that all packets are lost during transmission interruptions. We have tested various TCP versions including those in Linux, that in Windows and that in Mac OS. This paper suggests (1) that the tested TCPs in Linux follow the similar procedure and retransmit lost packets quickly, (2) that TCP in Windows also behaves well but the increase of congestion window seems to be limited, and (3) that TCP in Mac OS has shown some problems in retransmitting contiguously lost packets.**

Keywords-TCP; Transmission Interruption; Bursty Packet Losses; Retransmission; SACK Based Loss Recovery.

I. INTRODUCTION

Transmission Control Protocol (TCP) is widely used as a transport protocol for the reliable data transfer. TCP recovers from packet losses by retransmitting lost packets and guarantees that the information sent is safely delivered to the receivers. But, recent transmission technologies and media access control technologies provide high quality of data transmission, and so, packet losses detected in TCP are considered as the indication of network congestion.

Although the possibility of packet losses caused by random bit errors is extremely low, it is possible that data are lost due to transmission interruptions. For example, packets will be lost during a handoff among base stations in the 3rd generation mobile telecommunication networks [1]. Similar packet losses occur during a channel switch in the protection switching systems [2], [3].

These packet losses are bursty, because such a transmission interruption continues in the order of tens of miliseconds through several seconds. TCP, of course, has the functionality to recover from those bursty packet losses, but it seems that the research activities on TCP performance focus on the congestion control scheme during light congestion situation where the number of lost packets is limited [4].

This paper describes the results of experimental analysis of TCP behaviors when a TCP data transmission suffers from bursty packet losses during a transmission interruption. We have tested several TCP versions; TCP implemented in the Linux operating system [4], TCP in the Windows 7 operating system, and TCP in the Mac OS X operating system. For those TCP versions, the TCP communication traces are examined in detail. As a result, we suggest that
(1) the tested TCPs in Linux follow the similar procedure and retransmit lost packets quickly, that
(2) TCP in Windows 7 also behaves well, but the increase of congestion window seems to be limited compared with those in Linux, and that
(3) TCP in Mac OS X sometimes takes longer time than the others to retransmit the packets lost during a transmission interruption.

So far, there have been some papers published focusing on the TCP behaviors against packet losses [5] – [7]. In [5], TCP over a 3G wireless system, IS2000, is discussed. Especially, it describes the periodical data transmission timing in IS2000 and its impact on TCP, and the effectiveness of selective acknowledgment (SACK) [8] and timestamp TCP options. In [6], TCP performance over commercial WiMAX-based network is presented. It compares New Reno, Cubic [9], Vegas and Veno TCP variants in terms of throughput, rount-trip time and retransmission rate, and points out that a WiMAX link is not well-suited for the aggressive Cubic and window auto-tuning. Zhu and Bai [7] compared the performance of Tahoe, Reno and SACK TCP when multiple packets are dropped, and shows that Reno suffers from performance problems at multiple drops while SACK works well. On the contrary, this paper gives the detailed packet level analysis of TCP behaviors against burst errors using the timeline charts and points out the problems in Mac OS X TCP which are not discussed in the other papers.

The rest of this paper consists of the following sections. Section 2 specifies the conditions of the transmission interruption test. Section 3 gives the results of various TCP versions. Section 4 describes a packet level behavior analysis for the results of TCP Reno in the Linux operating system and Mac OS X TCP. Section 5 gives the conclusions of this paper.

II. Test Conditions

Fig. 1 shows the configuration of the experiment. The TCP program to be tested is implemented in a personal computer (PC under test in the figure). It is connected to a wireless LAN (IEEE 802.11g) through an access point (AP), which is connected to the bridge emulating transmission interruptions through Gigabit Ethernet. The bridge injects a 200 milisecond interruption at every five second. During the interruption, the bridge discards all packets transferred in both directions. The bridge is connected the ftp server through Gigabit Ethernet.

The TCP communication is traced using tcpdump. The trace is taken in the PC under test and ftp server, and two traces are examined for each experiment.

In this test, PC under test works as an ftp client and sends a 10 megabyte file to the ftp server. The specification of the equipment is listed in Table I.

The TCP versions adopted in this test are as follows:

- TCP Reno: a traditional additional increase and multiplicative decrease (AIMD) control of congestion window with fast recovery.
- Cubic TCP [9]: congestion window control as a cubic function of time elapsed since a last congestion event. It has been the default of Linux TCP suite since 2006.
- TCP Westwood [10]: designed for wireless network by estimating the available bandwidth from ACK arrival intervals.
- TCP in Windows 7: default TCP in the Windows 7 operating system. It is said to combine slow and scalable way in the congestion window calculation (compound TCP [11]).
- TCP in Mac OS X: default TCP in the OS X operating system.

The configuration of TCP options, such as whether to use the window scale option and the SACK option or not, follows the default setting of the individual operating systems.

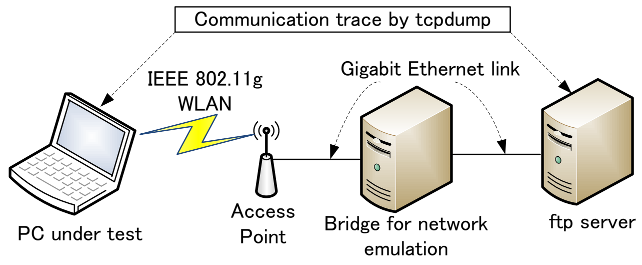

Figure 1. Configuration of Experiment

TABLE I. SPECIFICATION OF EQUIPMENT

Node	OS	Hardware specification
PC under test	Linux (Ubuntu 10.04)	Centrino2 CPU (2.53GHz) and 2GB memory
	Windows 7	
	Mac OS X (10.7.5)	MacbookPro with Core i7 CPU (2.4Ghz) and 8GB memory
bridge	Linux	Pentium4 HT CPU (2.4 GHz) and 2GB memory
ftp server	Linux	Core i5 CPU and 8GB memory

III. Results of Experiments

We executed several test runs for each of TCP versions. This section shows typical results for those test runs as the graphs plotting the sequence number of transferred data segments (transferred bytes) versus the elapsed time since the SYN segment. The graph is generated by Wireshark [12] using the trace captured at the PC under test.

Fig. 2 shows the result of TCP Reno. The figure shows four discontinuous sections in the increase of sequence number. Three of them are labeled as “Reno (1),” “Reno (3)” and “Reno (4).” It is considered that they are caused by bursty packet losses injected at the bridge. For confirmation, Fig. 3 shows the similar graph generated from the trace captured at the ftp server side. Fig. 3 shows that the increasing status of the sequence number at the server side is similar with that at the PC under test side, and that there are parts where packets are lost contiguously in the four discontinuous sections.

There are two types of discontinuous sections in Fig. 2. One is the type for the first through the third sections. In this type, packets are lost in the middle of a continuous data sending in the TCP flow control. There is no time lag between the normal data transmission and the retransmission.

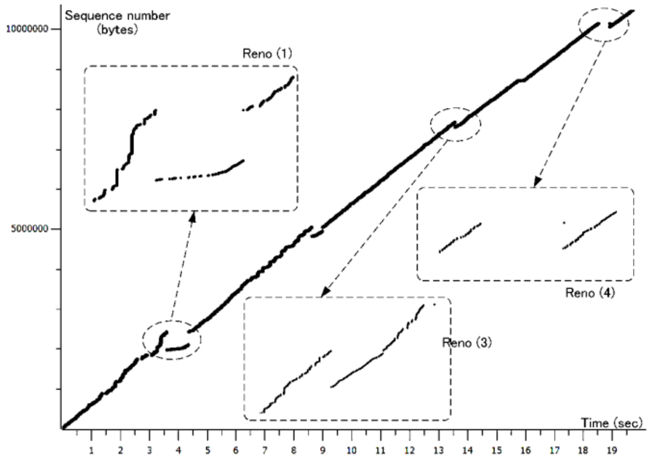

Figure 2. Sequence number vs. time for TCP Reno

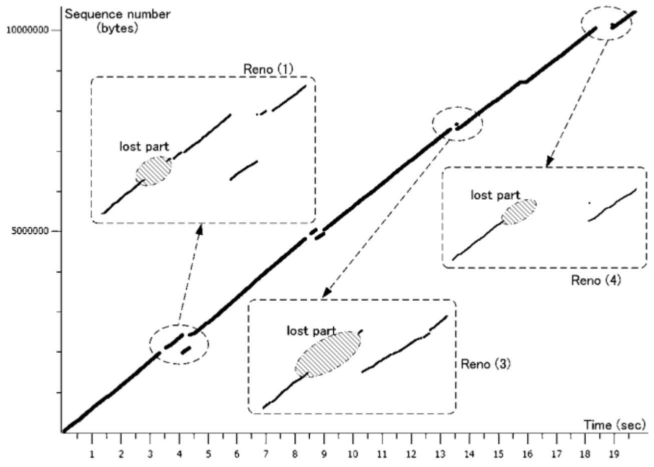

Figure 3. Sequence number vs. time for TCP Reno in ftp server side

The other type is that for the fourth discontinuous section (Reno (4)) in the figure. Packets are lost in the end of continuous data sending and there is a time lag before the retransmission starts. We will examine the packet level behaviors of these two types in the next section.

Fig. 4 shows the result of Cubic TCP. The sequence number versus time graph is similar with that of TCP Reno. There are four discontinuous sections; the first, and the second (Cubic (2)) and the fourth ones are of type with time lag, and the third one (Cubic (3)) is of type without time lag.

Fig. 5 shows the result of TCP Westwood. The graph is similar with those of TCP Reno and Cubic TCP. As the results of our experiment, it can be said that the TCP versions in the Linux operating system behave similarly for bursty packet losses, although they have different congestion control mechanisms.

Fig. 6 shows the result of TCP in Windows 7 operating system. This graph is also similar with those of TCP versions in the Linux operating system. But, there are only discontinuous sections with type of time lag. Besides the experiment described in Fig. 6, we executed three runs of the experiment and obtained the result that all the discontinuous sections are of type with time lag.

The reason for this difference is analyzed as follows. For the discontinuous section without time lag, the data sending needs to continue longer than the transmission interruption injected by the bridge, i.e., 200 miliseconds. So, the window size (advertised window and congestion window) needs to be large enough to allow PC under test to keep sending data segments. So, we have checked the advertised window size in the TCP Reno case and the Windows case and show the result in Fig. 7. In the case of TCP Reno, the advertised window goes to 451,840, but it goes to only 55,488 in the case of Windows TCP. It should be noted that the window scale option is used in both cases and that it is possible to specify a large window size. Generally, TCP receiver adjusts its window size dynamically to twice of the congestion window size which it estimated. This is called auto-tuning or dynamic right sizing [13]. So, it is considered that, in this experiment, the ftp server (receiver) estimated the congestion window of Windows 7 TCP much smaller than that of Linux TCP, and that Windows 7 TCP did not continue data sending longer than the 200 milisecond transmission interruption. So, all the discontinuous sections were of type with time gap. However, Windows 7 TCP also behaves well and recovers quickly from the bursty packet losses caused by transmission interruption.

In the end, Fig. 8 shows the result of TCP in Mac OS X.

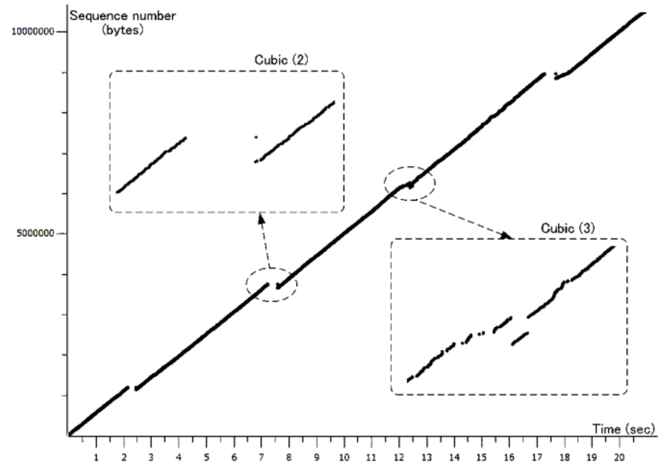

Figure 4. Sequence number vs. time for Cubic TCP

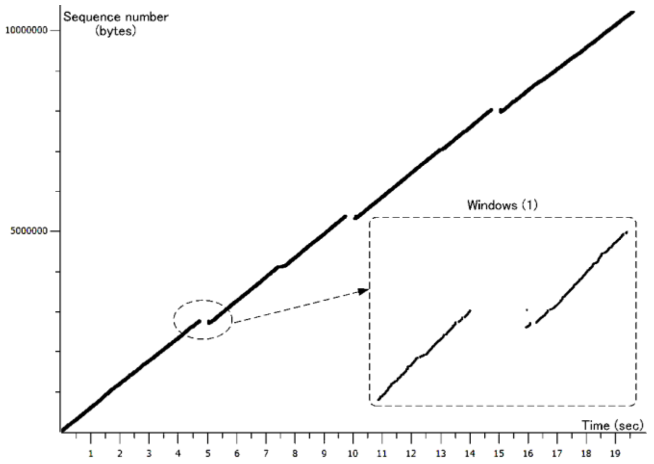

Figure 6. Sequence number vs. time for TCP in Windows

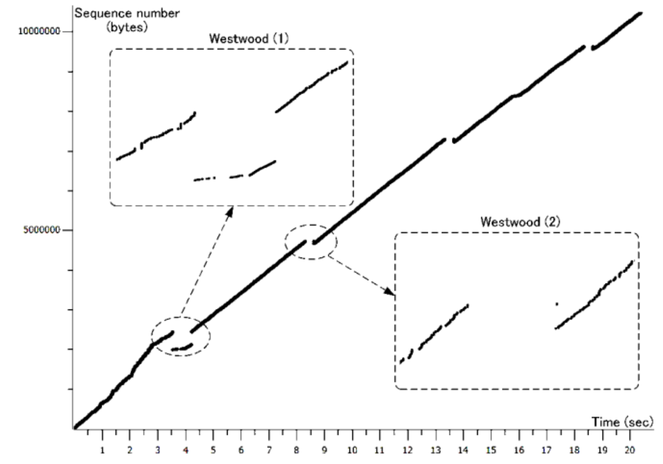

Figure 5. Sequence number vs. time for TCP Westwood

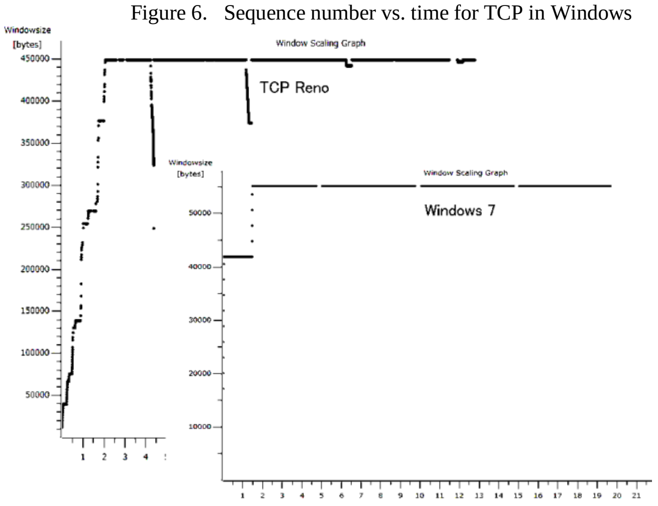

Figure 7. Advertized window for TCP Reno and Windows

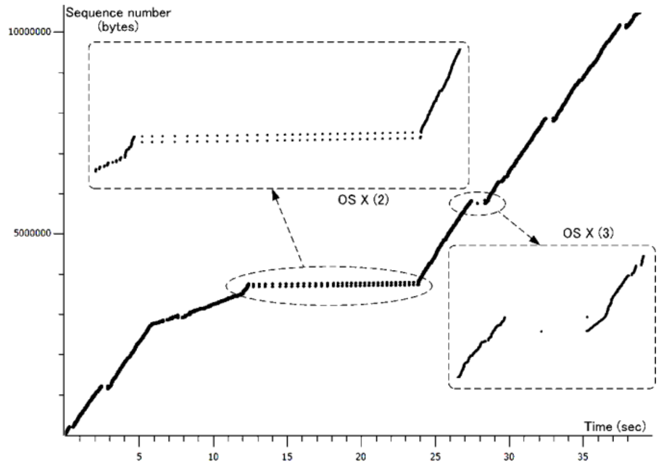

Figure 8. Sequence number vs. time for TCP in OS X

The graph is different from those of other TCP versions. In the second discontinuous section (OS X(2)), the graph is flat, i.e., the data sending rate is low, and data segments are sent intermittently. This section takes more than ten seconds and is considered to be worse error recovery than the other TCP versions described above. The detailed packet level analysis is given in the next section.

IV. Detailed Analysis of TCP Traces

A. *Packet Level Analysis of TCP Reno's First Discontinuous Section*

Fig. 9 shows the timeline of segment exchanges at the discontinuous section of Reno (1). This section is of type without time lag. The figure is described based on the trace at the PC under test side.

The data and ACK segments are specified according to the text format of tcpdump. For a data segment, the figure uses a representation such as "1967433:1968881(1448)," which means that the sequence number specified in the header of this segment is 1967433 (relative value from the SYN segment) and the number of bytes in this segment is 1448. The number 1968881 is the sequence number assigned the last byte in this segment plus 1, i.e., the sequence number in the header of the next data segment. For an ACK segment, this figure shows the acknowledgment number in the style of "ack 1965985" and, in addition, the other parameters such as a SACK option are also specified.

In this discontinuous section, 88 data segments are lost during the transmission interruption (*9(a)* in the figure). After that, the next data segment 2093409:2094857(1448) is sent to the ftp server (*9(b)*). In response to that, the ftp server returns an ACK segment 1967433 (SACK2094857-2096305) (*9(c)*). This ACK segment says that the sequence number of the next data which the receiver expects is 1,967,433, and that the receiver has received data from 2,094,857 to 2,096,304 [8]. Responding to this SACK segment, PC under test retransmits the data from sequence number 1,967,433. This is retransmitted because the ACK segment with SACK option says that the receiver received all of data up to 1,967,432 and that, in addition, it has

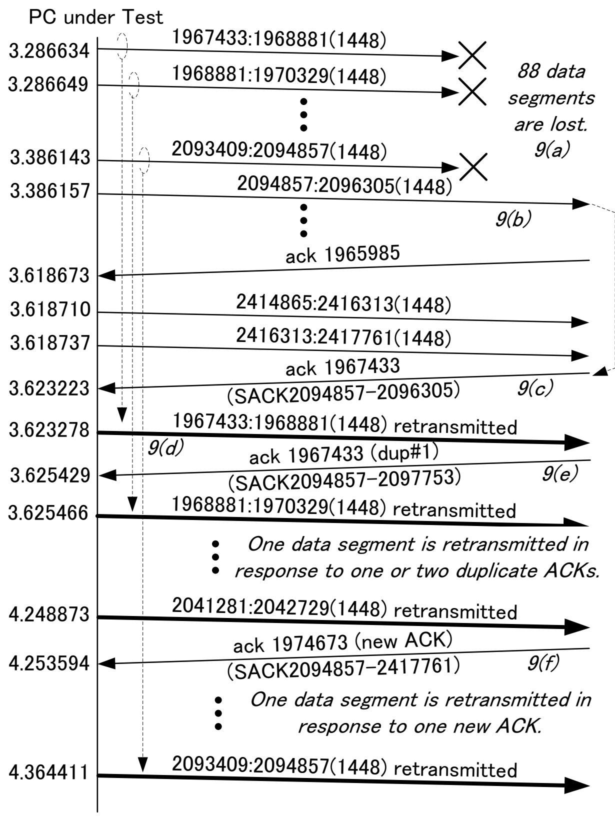

Figure 9. Timeline of segment exchanges at Reno (1)

received some SACKed data segments [14]. This means that it is possible that data from 1,967,433 to 2,094,856 are missing.

Before PC under test receives this ACK segment, it sends data segments up to 2416313:2417761(1448). Responding to those data segments, the ftp server returns other ACK segments with acknowledgment number set to 1967433 with SACK option whose range is increasing incrementally. They are duplicate ACKs with SACKs, and invoke the retransmission similarly with ACK *9(c)*.

The ftp server returns another ACK segment (*9(e)*) invoked by the next data segment, 1970329:1971777(1448). As described above, the information on newly received data is included in the SACK option (Please confirm that the range of SACK option becomes wider than that in ACK *9(c)*). After receiving all original data segments up to 2416313:2417761(1448), the ftp server receives retransmitted data segments, and responds to them by sending ACK segments (new ACKs) one by one (e.g., *9(f)*). Since these ACKs include SACK options, PC under test retransmits the next unacknowledged data segment one by one, until all the lost data segments are retransmitted.

In summary, at the discontinuous section without time lag, the receiver retransmits data segments just after it receives the first ACK with SACK option indicating a missing data gap. The retransmission seems to be based on the SACK. The requirement for this type of section is that the sender has a window size large enough to send data longer than the period of a transmission interruption. Similar ones can be found in Reno (3), Cubic (3), Westwood (1), and so on.

B. Packet Level Analysis of TCP Reno's Fourth Discontinuous Section

Fig. 10 shows the time line of segment exchanges at the discontinuous section Reno (4) in Fig. 2. This section is of type without time lag.

In this discontinuous section, 58 data segments and one ACK segment are lost during the transmission interruption (*10(a)* in the figure). A data segment 10058857:10060305 (1448) is delivered to the ftp server, but the correspondent ACK segment (ack 10060305) is lost. The interruption losses up to the last data segment in continuous data sent in one TCP window. So, after an interruption, PC under test has any segments to send, and it just waits. When the retransmission timeout period passes, the sender retransmits the oldest unacknowledged data segment (*10(b)*).

Then, the receiver receives this segment, and it returns a corresponding ACK segment. But, the receiver already received this data segment before. In order to inform the sender of the duplicate receipt of this data segment, the receiver specify the duplicate range in the first block of the

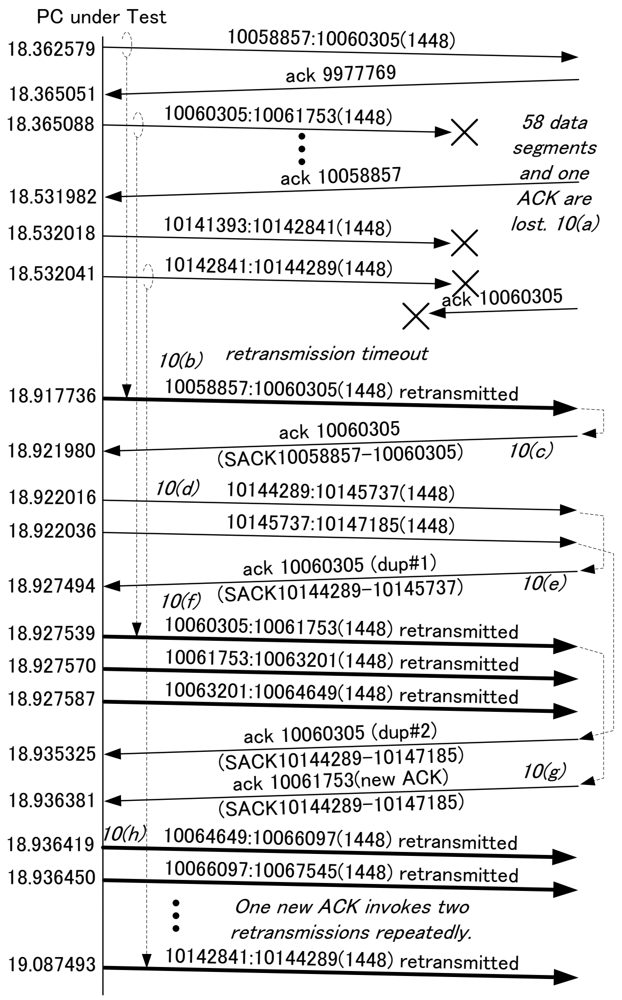

Figure 10. Timeline of segment exchanges at Reno (4)

SACK option, as shown in the figure (*10(c)*). This mechanism is called DSACK [15].

After receiving this ACK segment, the sender retransmits two data segments following the last data segment which it has sent (*10(d)*). For each of these data segments, the receiver responds with an ACK segment, which will be a duplicate ACK with a SACK option (*10(e)*). When PC under test receives this ACK segment, it retransmits data segments (*10(f)*). This is considered as the retransmission by SACK option, which was also used in Fig. 9. These retransmissions generate new ACKs with the SACK option (*10(g)*), and again, they introduce the retransmissions (*10(h)*).

In summary, at the discontinuous section with time lag, the receiver will start to retransmit data segments due to the retransmission timeout. But, after the first retransmissions, the continuing retransmissions are invoked by the SACK based recovery. Similar discontinuous sections can be found in Cubic (2), Westwood (2), Windows (1), and so on.

C. Packet Level Analysis of Mac OS X TCP's Second Discontinuous Section

The discontinuous section OS X (2) is different from the others obtained in this experiment. Fig. 11 shows the time line of segment exchanges at this section.

At first, 46 data segments are lost during the transmission interruption (*11(a)* in the figure). Similarly with Fig. 10, the interruption losses up to the last data segment in one TCP window. So, the retransmission timeout occurs and the oldest unacknowledged data segment, 3675809:3677257 (1448), is retransmitted (*11(b)*). Then, the receiver returns a corresponding ACK segment for this data segment. It is a new ACK segment without any SACK options (*11(c)*). After receiving this ACK, the sender transmits a (new) data segment following the last data segment it sent (*11(d)*). For this data segment, the receiver responds an ACK segment which will be a duplicate ACK with a SACK option (*11(e)*).

So far, the timeline is very similar with that of Reno (4). But, when PC under test receives this ACK segment, it does not retransmit any data segments immediately. That is, the retransmit by the SACK option is not invoked. Instead, the sender waits for the retransmission timeout period and retransmits the oldest unacknowledged data segment (*11(f)*).

In the timeline, this sequence, a timeout retransmission, a new ACK, a new data, a duplicate ACK, and another timeout retransmission, is repeated. So, the intermittent data sending occurs. The reason for this sequence is considered to be the fact that the SACK based retransmission does not work well.

However, in the end of this sequence, PC under test receives a new ACK with SACK option (*11(g)*), and it retransmits next unacknowledged data segment (*11(h)*). In this part, it seems that the SACK based loss recovery works well. At another discontinuous section, OS X (3), the behavior of type with time lag is observed. Here, it seems that the SACK based retransmission is working.

In summary, Mac OS X TCP shows an intermittent type discontinuous section for bursty packet losses in a transmission interruption. The reason is that the loss recovery based on the SACK option does not work well. However, the SACK based loss recovery works in another

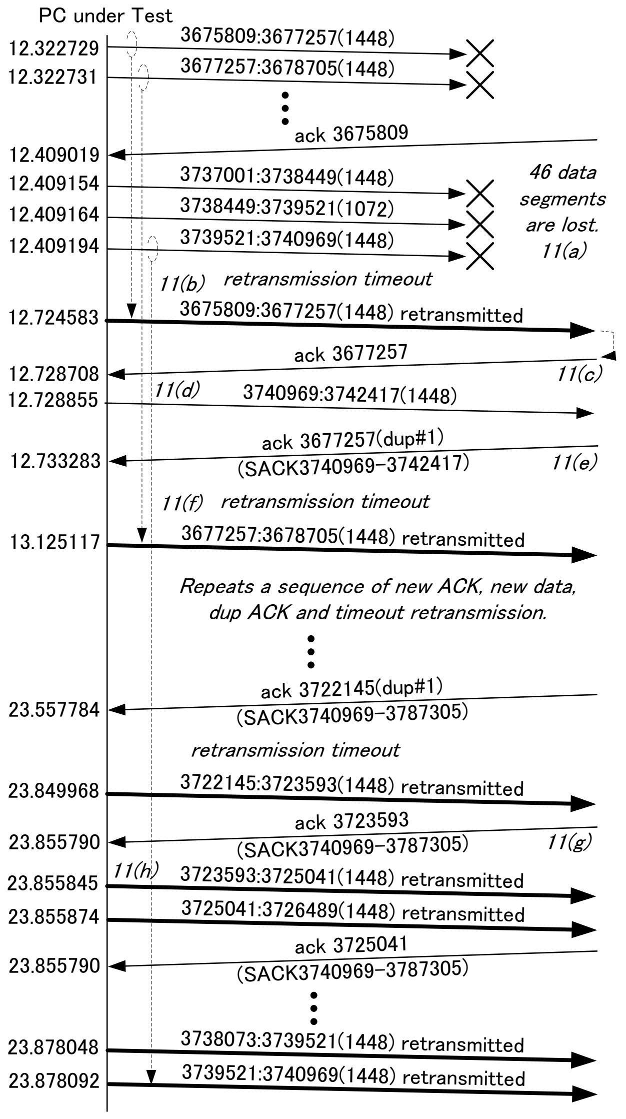

Figure 11. Timeline of segment exchanges at OS X (2)

discontinuous section in Mac OS X TCP, so we cannot say that it is not implemented in the Mac OS X.

V. CONCLUSIONS

This paper described the results of experimental analysis of TCP retransmission behaviors against bursty packet losses caused by transmission interruptions. We focused on several TCP versions; TCP Reno as a standard congestion control, Cubic TCP as a high speed version, TCP Westwood for a wireless network, TCP in Windows 7 and TCP in Mac OS X. The packet level detailed analysis for the TCP communication traces found the followings.

(1) The tested TCPs in Linux seem to follow the similar procedure and retransmit lost packets quickly. They behave as recovery types with time lag and without time lag, depending on whether timeout retransmission is used or not for the first missing data segment.

(2) TCP in Windows 7 also behaves well, but discontinuous sections caused by transmission interruptions are with type of time lag. The reason seems to be that the increase of congestion window of Windows 7 TCP much smaller than that of Linux TCP, and that the receiver does not advertise a large window size according to the dynamic right sizing.

(3) TCP in Mac OS X sometimes shows an intermittent type of retransmission which takes longer time than the others. In the experiment, it took several seconds to retransmit all the lost packets. The reason seems to be that the loss recovery based on the SACK option does not work well in Mac OS X. But, in other retransmissions, Mac OS TCP uses SACK based recovery, and so the clarification of Mac OS TCP behaviors is for further study.

REFERENCES

[1] 3GPP TS 23.009 version 7.0.0 Release 7, “Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); Handover procedures,” ETSI, Mar. 2007.

[2] IEEE Std 802.17, “Part 17: Resilient packet ring (RPR) access method and physical layer specifications,” IEEE Standard Association, May 2011.

[3] Recommendation ITU-T G.8031/Y.1342, “Ethernet linear protection switching,” Telecommunication Standardization Sector of ITU, June 2011.

[4] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock, “Host-to-Host Congestion Control for TCP,” IEEE Commun. Surveys Tutorials, vol. 12, no. 3, 3rd quarter 2010, pp. 304-340.

[5] F. Khafizov and M. Yavuz, “Running TCP over IS-2000,” Proc. ICC 2002, April 2002, pp. 3444-3448 vol. 5.

[6] E. Halepovic, Q. Wu, C. Williamson, and M. Ghaderi, “TCP over WiMAX: A Measurement Study,” Proc. IEEE MASCOTS 2008, Sept. 2008, pp. 1-10.

[7] J. Zhu and T, Bai, “Performance of Tahoe, Reno, and SACK TCP at Different Scenarios,” Proc. ICCT ’06, Nov. 2006, pp. 1-4.

[8] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, “TCP Selective Acknowledgment Options,” IETF RFC 2018, Oct. 1996.

[9] I. Rhee and L. Xu, “CUBIC: a new TCP-friendly high-speed TCP variant,” SIGOPS Operating Systems Review, vol. 42, no. 5, July 2008, pp. 64-74.

[10] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, “TCP Westwood: Bandwidth estimation for enhanced transport over wireless links,” Proc. ACM MOBICOM 2001, July 2001, pp. 287-297.

[11] K. Tan, J. Song, Q. Zhang, and M. Sridharan, “A compound TCP approach for high-speed and long distance networks,” Procl IEEE INFOCOM 2006, April 2006, pp. 1-12.

[12] Wireshark Foundation, “WIRESHARK,” http://www.wireshark.org/

[13] M. Fisk and W. Feng, “Dynamic Right-Sizing in TCP,” Proc. Los Alamos Computer Science Institute Symposium, Oct. 2001.

[14] E, Blanton, M. Allman, L. Wang, I. Jarvinen, M. Kojo, and Y. Nishida, “A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP,” IETF RFC 6675, Aug. 2012.

[15] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, “An Extension to the Selective Acknowledgment (SACK) Option for TCP,” IETF RFC 2883, July 2000.