# 例外的な通信シーケンスを生成する TCP 試験システムを用いた SACK の実装の評価

大岸　智彦　　長谷川　亨　　加藤　聰彦

KDDI 研究所

TCP はインターネットにおいて広く普及しているが、なお新しい仕様が提案され実装されており、新旧の実装間の通信に不具合が生じる可能性が考えられる。このため、通信システム内の TCP の実装を試験することが重要がある。これを行うには、テスト系列の生成に労力を要しない試験システムが必要となる。筆者らは、例外的な通信シーケンスのみをテストシナリオに記述することで、TCP の試験を行うシステムを開発した。本システムは、テストシナリオに記述された条件が満たされた場合のみ、例外的な通信シーケンスを実行する。本システムは、PSC が開発した NetBSD 上の SACK を含むカーネル内 TCP モジュールを変更することにより実装している。筆者らは、本システムを用いて SACK の実装を評価した。

# Evaluation of SACK Implementation using TCP Test System Generating Exceptional Packet Sequence

Tomohiko Ogishi, Toru Hasegawa and Toshihiko Kato

KDDI R&D Laboratories

Although TCP is widely used in Internet, new specifications are still proposed and implemented. In the circumstance above, it is highly possible that some errors are detected on the communication between new and old implementations. Several tools for testing TCP implementations were developed so far. However, they does not have enough function to customize test sequence or need significant effort to specify the sequence. We developed a TCP test system which specifies only exceptional packet sequence in the test scenario. The system performs exceptional packet sequence only when the condition specified in the test scenario is satisfied. Otherwise, it performs ordinary TCP behavior. The system is implemented by modifying in-kernel TCP module of NetBSD with SACK code developed by Pittsburgh Supercomputing Center. We also evaluated SACK implementation as an example of recent specification using the test system.

## 1. INTRODUCTION

TCP [1] is widely used as a protocol to provide reliable transfer from the dawn of Internet. The TCP protocol functions and implementations have been modified and extended in the long history of TCP. Currently, the TCP implementations in widely spread PCs and workstations are fairly stable and the users seldom feel inconvenience at TCP communications. However, there are several implementation errors as reported in [2]. Due to those errors, TCP communications in specific situations, such as a long haul TCP communication, suffer from problems like serious throughput degradation. Those well-known errors have been fixed in the higher version of implementations, but, still now, new functions, such

as a new congestion control mechanism in NewReno [3] and the selective acknowledgment (SACK) [4], are proposed and integrated into available TCP implementations. It is highly possible that not detected errors related to conventional functions [5] and errors related to new functions [3,4] bring new problems in TCP communications.

In order to detect such errors, some tools for testing TCP implementation are required. So far, various tools have been developed [6], which are classified into two groups, monitors and testers. The monitors collect packet data exchanged between end systems and analyzes TCP protocol behaviors. TCPAnaly [7] and the Intelligent TCP Analyzer [8] are categorized in this group. However, the monitors cannot control packet sequences in the testing, and it is not easy to detect errors in TCP implementations by monitors.

On the other hand, the testers can generate packets suitable for a specific testing purpose. But, the testers developed so far [9-11] have some issues to perform the TCP implementation testing. Among them, TBIT [9] and Nmap [10] use some predefined test sequences. The purpose of TBIT is to check the compliance and used parameter values of TCP implementations, and that of Nmap is to estimate the version of operating systems from the TCP behaviors. That is, those systems do not allow test operators to use test sequences specific to their test purposes. On the other hand, [11] proposes a TCP test system which can generate test sequences based on test scenario specification written in TTCN [12]. This system allows test operators to use suited test sequences. However, the task to generate a test sequence will be a hard job for test operators because it is required to specify all input and output events used in the sequence.

Considering these issues, it is important to reduce the burden for test operators to specify a test sequence. When a test operator tries to test a normal behavior of TCP module in a system under test, it is possible to use an ordinary TCP module as a tester. A specially ordered mechanism is used only when a test operator tries to test an exceptional behavior of TCP. Therefore, we propose a TCP test system which uses an ordinary TCP module for testing normal TCP behaviors and which allows test operators to specify a test sequence only for testing exceptional TCP sequences. By using our TCP test system, a test operator can perform exceptional test sequence, such as sending SYN segment in ESTABLISHED state, sending ACK segment with smaller acknowledgment number, and sending SACK options with the first and the second SACK blocks misordered. A test operator does not need to specify normal sequences but describe exceptional part in a test scenario. The TCP module in the test system behaves normally if the condition on a test scenario is not satisfied. If the condition is satisfied, it runs the actions specified in the test scenario. It also saves a communication logs which is examined after the test run is over.

In this paper, we describe the design and the implementation of the TCP test system. We also describe the evaluation result of SACK implementation by using the TCP test system. In Section 2, we describe the design and the implementation of our test system. Section 3 and 4 shows an overview of SACK function and the evaluation of SACK implementation by using the test system, respectively. Finally, Section 5 makes a conclusion.

## 2. DESIGN AND IMPLEMENTAION OF TCP TEST SYSTEM

### 2.1. Structure

Figure 1 shows the functional structure of the TCP test system. The test system is composed by test execution part and test analysis part as described in the previous section. The test execution part includes *scenario loader*, TCP application program, in-kernel TCP module and network interface. In the in-kernel TCP module, *scenario interpreter* and *log collector* are implemented. Scenario loader provides the operator to configure the test environment. It selects the test scenario performed at scenario interpreter and whether the communication log is collected or not. TCP application program actually sends or receives user data on TCP by communicating with server or client program running on system under test. Scenario interpreter reads the test scenario and decides whether the action in the test scenario is executed or not. When the action is executed, it directly sends the packets described in the action to network interface through log collector. The

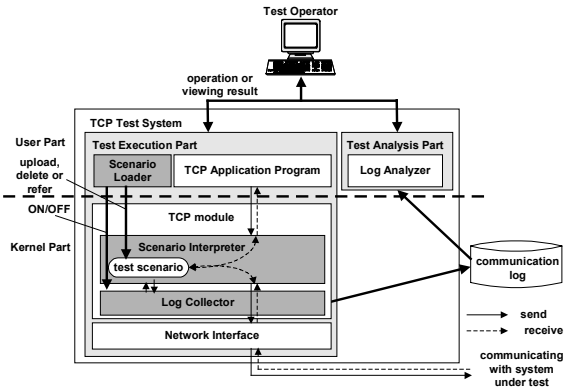


*Figure 1.* Structure of TCP Test System

interpreter maintains the scenario counter indicating the line of test scenario. Log collector is placed on the network interface and monitors the segments received from or sent to network interface, and collects a communication log related to the test. It also refers to the scenario counter when a segment is added to the communication log. *Log analyzer* in the test analysis part provides the operator to analyze the communication log.

### 2.2. Test Scenario

Figure 2 shows an example of test scenario. The test scenario consists of *header* and *content parts*. In the header part, IP addresses and port numbers for source and destination, which indicate TCP connections under test. For example, if 20 is specified as source port number, the TCP behavior of FTP data connections becomes the target. In this case, TCP connections with other source port number such as FTP control connection or WWW client behaves without any influence of test scenario. In other words, the system can protect other application programs from being affected by the test scenario. In the header part, TCP options used at connection establishment and the parameter value of each option are also specified by using *syn-opt* key. When the key is specified, the specified options are forced to be used in SYN or SYN+ACK segment without negotiating with peer end system. In the example of Fig. 2, TCP options are sent with maximum segment size (MSS) option set 10 bytes, sack permitted option, timestamp option and window scale factor (WSF) option set 0.

In the content part, the behaviors after establishing TCP connection are specified. In the each line of content part, the condition and the action are specified by being separated with semi-colon. In the condition, a trigger event and its parameter values are specified. There are three types of trigger events, *recv*, *send* and *wait*, which mean received packet, sending packet and timer expiration, as described above. In the case of *recv* and *send* events, several expressions such as "=", "<=", ">=", ">" and "<" can be used with a compared value. If *any* is specified as the value, the condition always becomes true as far as the parameter exists. At the end of the condition, *var-upd* which means whether internal variables related to the received packet such as *rcv_nxt* are updated or not is also specified. If *var-upd* is OFF at *recv* event, it becomes the same situation that the received packet is lost at network. In the action, a packet to be sent with its parameter values at a true condition is specified. All of the parameter values in the TCP header except urgent pointer and checksum should be specified in the action. TCP options such as MSS and SACK are specified with parameter values if the options are included in the sent packet. At the end of the condition, *var-upd* which means whether internal variables related to the sent packet such as *snd_nxt* are updated or not is also specified. If no packet is desired to be sent, *ignore* is specified as action. If multiple packets are desired to be sent at a time, the second and later actions are specified without conditions.

```
<header>
src-addr=192.168.0.1
dst-addr=192.168.0.2
src-port=20
syn-opt=mss(10), sack, timestamp, wsf(0)

<content>
recv seq=1 var-upd=ON     ; send seq=1 ack=11 flag=(ack) win=20 var-upd=ON
recv seq=11 var-upd=OFF   ; ignore
send seq=1 ack=11 var-upd=yes  ; ignore
wait 1                    ; send seq=1 ack=11 flag=(ack) win=30 var-upd=OFF
recv seq=11 var-upd=ON    ; ignore
recv seq=21 var-upd=OFF   ; ignore
recv seq=31 var-upd=ON    ; send seq=1 ack=21 sack=(31-41) flag=(ack) win=30 var-upd=ON
                          ; send seq=1 ack=21 sack=(31-41) flag=(ack) win=30 var-upd=ON
                          ; send seq=1 ack=21 sack=(31-41) flag=(ack) win=30 var-upd=ON
                          ; send seq=1 ack=21 sack=(31-41) flag=(ack) win=30 var-upd=ON
recv seq>70 var-upd=ON    ; send seq=1 flag=(rst) win=30 var-upd=ON
```

*Figure 2*. Example of Test Scenario

### 2.3. Scenario Interpreter

The flow chart presenting how scenario interpreter works is illustrated in Fig. 3. Scenario interpreter maintains a test scenario loaded to the test system. It examines all packets which are received or going to be sent and checks whether the TCP connection of the packet is under test or not. If the connection is under test, the interpreter checks whether the TCP state is ESTABLISHED or not. If not, the packet is dealt as a normal TCP behavior. Only if *syn-opt* is specified in the header part of the test scenario and SYN or SYN+ACK segments is going to be sent, the TCP options for the sent packet are modified as what

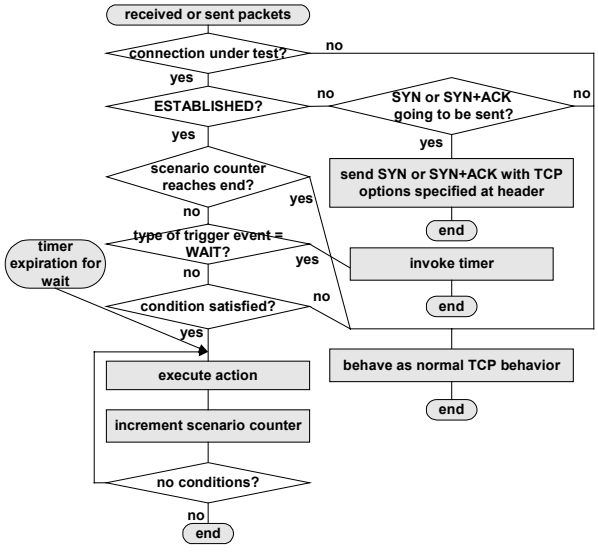


*Figure 3*. Flow Chart for Scenario Interpreter

specified at *syn-opt* before sending it. If the state is ESTABLISHED, the interpreter maintains the *scenario counter*, which indicates the line executed currently. The counter starts at the first line of the content part when the TCP state enters ESTABLISHED and is incremented only when the condition at the counter is satisfied and the action is executed. If the counter reaches to the end, further events are dealt as normal TCP behavior. If the trigger event is *wait*, a timer is invoked. The expired time is set to the number of time slots specified in the condition. While the timer is working, the following sent or received events obey to normal TCP behavior. When the timer is expired, the action for the *wait* is executed and the scenario counter is incremented. Even if the counter reaches to the end, the communication under test does not finish at the time. It continues by normal TCP behavior.

### 2.4. Log Collector

Log collector saves a packet received or going to be sent to the communication log if the TCP connection of the packet is under test and the function is set on. Figure 4 shows a communication log collected at the test of Fig. 2. The output format is almost same as that of *tcpdump* [13]. The differences are as follows:

- The communication log shows the scenario counter on the top of line for each event.
- The communication log can show the packets which are not actually sent or received at TCP of the test system using parenthesis such as line 2 and line 3.
- The communication log shows whether internal variables are updated or not. If updated, *(var-upd)* is added to the line.

```
0: 0.000000     SYN seq=3,325,424 win=32,768 mss=1,460 sack-permitted timestamp wsf=0
0: 0.000128 SYN+ACK seq=1,783,630 ack=1 win=32,768 mss=10 sack-permitted timestatmp wsf=0
0: 0.000233     ACK seq=1 ack=1 win=32,768 (var-upd)
1: 0.001733     DATA seq=1 ack=1 win=32,768 len=10 (var-upd)
1: 0.001855     ACK seq=1 ack=11 win=20
2: 0.001934     (DATA seq=11 ack=1 win=32,768 len=10)
3: 0.002018 (ACK seq=1 ack=11 win=8,760) (var-upd)
4: 0.478445 ACK seq=1 ack=11 win=30
5: 3.712842     DATA seq=11 ack=1 win=32,768 len=10 (var-upd)
6: 3.712932     (DATA seq=21 ack=1 win=32,768 len=10)
7: 3.713015     DATA seq=31 ack=1 win=32,768 len=10 (var-upd)
7: 3.713099 ACK seq=1 ack=21 sack=(31-41) win=30 (var-upd)
8: 3.713131 ACK seq=1 ack=21 sack=(31-41) win=30 (var-upd)
9: 3.713148 ACK seq=1 ack=21 sack=(31-41) win=30 (var-upd)
10: 3.713166 ACK seq=1 ack=21 sack=(31-41) win=30 (var-upd)
11: 3.713275     DATA seq=21 ack=1 win=32,768 len=10
11: 3.713383 ACK seq=1 ack=41 win=8,760
11: 3.713425     DATA seq=41 ack=1 win=32,768 len=10
11: 3.713499     DATA seq=51 ack=1 win=32,768 len=10
11: 3.713548 ACK seq=1 ack=61 win=8,760
11: 3.713552     DATA seq=61 ack=1 win=32,768 len=10
11: 3.713631     DATA seq=71 ack=1 win=32,768 len=10
11: 3.713693 RST seq=1 win=30 (var-upd)
end:3.713755     DATA seq=81 ack=1 win=32,768 len=10
end:3.713830 RST seq=1 win=8,760
```

*Figure 4*. Example of Communication Log

### 2.5. Implementation

We implemented the TCP test system based on NetBSD 1.3.2 with SACK developed by Pittsburgh Supercomputing Center (PSC) [14] by modifying its source code. Scenario loader was implemented as a user program. Scenario interpreter and log collector were implemented in the kernel by modifying PSC's TCP module.

## 3. OVERVIEW OF SACK FUNCTION

SACK function notifies the sender with a range of dropped packets accurately by using SACK options [4], which is the same as that maintained by the receiver. If the sender implements the retransmission algorithm effectively, it can theoretically send all dropped packets in a window in one round trip time (RTT). Therefore, SACK function becomes effective between end systems over a link with long RTT or high packet loss rate such as satellite or wireless link.

The sender who receives ACK segments with SACK options retransmits unacknowledged packets by an algorithm unique to the implementation of in-kernel TCP module. The pipe algorithm proposed in [15] is one of the methodologies to realize sender's behavior.

Recently, an extension to existing SACK options is proposed [16]. A *duplicate SACK* segment is sent when the receiver receives a segment including the range of sequence number previously received. The format of duplicate SACK is the same as *normal* SACK specified in [4] except it uses older ranges of sequence number in SACK block relative to the acknowledgement number. Since the use of duplicate SACK was recently specified, it seems there were few implementations to support this. It is mentioned that the latest Linux implementation supports it.

## 4. EVALUATION OF SACK IMPLEMENTATION

We evaluated SACK implementations in various operating systems using the implemented TCP test system. Figure 5 shows the network configuration of this test. We attached a network simulator between the test system and the system under test, and inserted one-second round trip delay in order to examine how the congestion control works. We used *ftp* as application program and created a file of 16 Kilobytes at the system under test and sent the file to the test system. We tested the sender's behavior when ACK segments with SACK options (which we call SACK

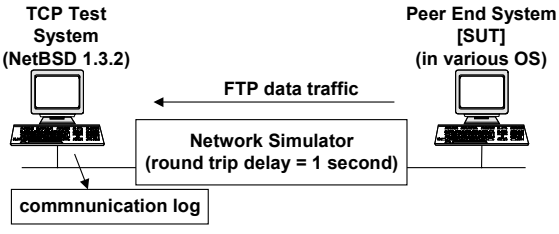


*Figure 5*. Network Configuration for SACK Testing

segments) are received. The operating systems we tested are SPARC Solaris 2.6 with SACK patch (which we call Solaris 2.6), Intel Solaris 8 (which we call Solaris 8), Linux kernel 2.4.2 (which we call Linux), NetBSD 1.3.2 with SACK code developed by PSC (which we call NetBSD), and Windows 98 Second Edition (which we call Windows 98). We performed three different tests described in the following subsections.

### 4.1. Multiple Drops in a Window

We tested retransmission algorithm using the test scenario depicted in Fig. 6. This test scenario sets MSS as 100 bytes and emulates drops at network of every other packet after congestion window extends enough. Since the actions for received packets such as the packet with its *seq* 3501, the test system sends SACK segments against each packet received in this RTT. The aim of the test is to examine how the sender behaves to multiple drops.

```
<header>
src-addr=192.168.0.1
dst-addr=192.168.0.2
src-port=20
syn-opt=mss(100),sack
<content>
recv seq=3401 var-upd=OFF; ignore;
recv seq=3601 var-upd=OFF; ignore;
recv seq=3801 var-upd=OFF; ignore;
recv seq=4001 var-upd=OFF; ignore;
recv seq=4201 var-upd=OFF; ignore;
recv seq=4401 var-upd=OFF; ignore;
```

*Figure 6*. Test Scenario for Multiple Drops in a Window

Table 1 shows the result of the test. Each column represents the operating system (OS), total elapsed time to transmit 16 Kilobytes, initial congestion window (ICW) used at slow start, number of ACK segments the SUT received before receiving duplicate ACK and RTT taken for retransmitting 6 dropped packets. Through this test, we found following facts and problems:

- All implementations deal with SACK options to retransmit dropped packets. If the retransmission algorithm for SACK were not implemented, it might take six RTTs to retransmit all of the dropped packets.
- Solaris 8 took two RTTs and Windows 98 took three RTTs to recover from dropped packets. However, we can not say the retransmission algorithms of these operating systems were worse than those of others. For example, when we set the ICW of Solaris 8 to 2, the RTT becomes one like others. From another test which drops 1501, 1701 and 1901, Linux took two RTTs while Solaris 2.6 and NetBSD took one. We think RTT taken to recover from multiple drops is highly related to ICW and number of ACK because these values decide the value of congestion window when the congestion occurs.
- We found typical implementations, which are inconsistent with [17], on Linux, Solaris 2.6 and Window 98. All of them seem to inflate window before receiving three duplicate ACK segments. In addition, Linux retransmits faster (by one duplicate ACK in this test) than other implementations when it receives SACK segments. Since Windows 98 sent only 96 bytes when it retransmitted the packet with its *seq* 4001, it took another one RTT to retransmit the rest of 4 bytes.

*Table 1*. Result for Multiple Drops in a Window

| OS | Total Time (second) | Initial Congestion Window | number of ACK | RTT |
|---|---|---|---|---|
| Solaris 2.6 | 15.1 | 2 | 23 | 1 |
| Solaris 8 | 14.1 | 4 | 23 | 2 |
| Linux | 15.2 | 2 | 24 | 1 |
| NetBSD | 16.1 | 2 | 23 | 1 |
| Windows 98 | 17.3 | 2 | 24 | 3 |

### 4.2. Delay and Loss of SACK

SACK segments may not be arrived in correct order to the sender. This situation is possible according to the network condition. Figure 7 shows the test for the sender's behavior when SACK segments are delayed and lost in network. In this figure, the SACK segments for DATA with *seq* 3501 and 3901 are lost shown in line 2 and 6, and 4101 is delayed shown in line 4 and 9.

All implementations treated well for these situations. The lost packets were retransmitted in one RTT in sequential order. We can estimate that current SACK implementations do not consider the order of arrival from this result.

```
<header>
src-addr=192.168.0.1
dst-addr=192.168.0.2
src-port=20
syn-opt=mss(100),sack
<content>
recv seq=3401 var-upd=no; ignore;
send ack=any var-upd=no; ignore;
recv seq=3601 var-upd=no; ignore;
send ack=any var-upd=no; ignore;
recv seq=3801 var-upd=no; ignore;
send ack=any var-upd=no; ignore;
recv seq=4001 var-upd=no; ignore;
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(4101-4201,3901-4001,3701-
3801,3501-3601) win=16400 var-upd=no;
; send tcp seq=1 ack=3401 flag=(ack) sack=(3701-3801,3501-3601) win=16400 var-upd=no;
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(4101-4301) win=16400 var-upd=no;
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(4101-4401) win=16400 var-upd=no;
                              :
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(4101-5001) win=16400 var-upd=no;
<end>
```

*any* is used when the condition is true on all value

*Figure 7*. Test Scenario for Delay and Loss of SACK

### 4.3. Duplicate and Identical SACK

The third test we performed was sender's behavior to multiple duplicate ACK segments with duplicate and *identical* SACK. The duplicate ACK segment with the same SACK blocks, which we call *identical* SACK, should not be occurred as far as it is not duplicated in network in the current specifications. However, we think the sender should deal well the receipt of identical SACK. Figure 8 shows the test scenario for duplicate SACK.

The implementations other than Linux seem to inflate window when it receives duplicate or identical SACK. The behavior is not agreeable from the theory in the pipe algorithm. NetBSD decrements *pipe* value every time when it receives duplicate ACK without referring the values in SACK block by referring its source code. Solaris 2.6, Solaris 8 and Windows 98 seem to implement in the same way since the result in the communication log was the same. On the other hand, Linux did not inflate window to duplicate SACK. It maintains the bytes of unacknowledged segments instead of *pipe* value by referring the source code. The result for identical SACK was also the same as above.

```
<header>
src-addr=192.168.0.1
dst-addr=192.168.0.2
src-port=20
syn-opt=mss(100),sack
<content>
recv seq=3401 var-upd=no; ignore;
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(3501-3601) win=16400 var-upd=no;
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(3501-3701) win=16400 var-upd=no;
                                    :
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(3501-4501) win=16400 var-upd=no;
; send tcp seq=1 ack=3401 flag=(ack) sack=(1001-1101,3501-4501) win=16400 var-upd=no;
; send tcp seq=1 ack=3401 flag=(ack) sack=(1101-1201,3501-4501) win=16400 var-upd=no;
                                    :
; send tcp seq=1 ack=3401 flag=(ack) sack=(2001-2101,3501-4501) win=16400 var-upd=no;
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(3501-4601,2001,2101) win=16400 var-upd=no;
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(3501-4701,2001,2101) win=16400 var-upd=no;
                                    :
send ack=any var-upd=no; send tcp seq=1 ack=3401 flag=(ack) sack=(3501,5901,2001,2101) win=16400 var-upd=no;
<end>
```

*Figure 8*. Test Scenario for Duplicate SACK

## 5. CONCLUSION

In this paper, we described the design, implementation and experimental result of a TCP test system. The system is designed to reduce the burden of test operator by specifying only exceptional behaviors which the operator wishes to test in the test scenario among a complete communication. The system supports a variety of description of the test scenario. It provides three types of trigger events as the condition to cause the exceptional behavior, which are received packet, sending packet and timer expiration. The condition is described using parameters in the packet. As the action for an event, a packet with any parameter values can be produced. It is also possible not to send any packet for an action. The function above is useful for testing congestion control, which is one of the most important functions in TCP.

The scenario interpreter selects the action to each event by the condition described in the test scenario. If the condition is satisfied, the action described in the test scenario is executed. If not, the action is determined by the behavior of original TCP module.
The scenario interpreter and the log collector of the test system are implemented into the TCP module by modifying the module itself. It is implemented on NetBSD with SACK code developed by PSC. The test scenario is uploaded to the TCP module before it is executed. This implementation facilitates the combination of normal TCP behavior and scenario-oriented TCP behavior in a communication.

For the experimental usage of the system, we described the evaluation of SACK implementations on several operating systems. We selected test cases which can happen in actual environment. We found several facts and problems in SACK implementations in some operating systems. It is considered that the test system is effective on testing of TCP through this experiment.

## REFERENCES


[1] W. Stevens, “TCP/IP Illustrated, Vol. 1: The Protocols,” Addison Wesley, 1994.

[2] V. Paxson, M. Allman, S.Dawson, W.Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke and B. Volz, “Known TCP Implementation Problems,” RFC 2525, Mar. 1999.

[3] S. Floyd and T. Henderson, “The NewReno Modification to TCP's Fast Recovery Algorithm,” RFC2582, Apr. 1999.

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

[5] S. Savage, N. Cardwell, D. Wetherall and T. Anderson, “TCP Congestion Control with a Misbehaving Receiver, ” ACM Computer Communication Review, Oct. 1999.

[6] S. Parker and C. Schmechel, “Some Testing Tools for TCP Implementors,” RFC 2398, Aug. 1998.

[7] V. Paxon, “Automated Packet Trace Analysis of TCP Implementations,” in Proc. of SIGCOMM ’97, Aug. 1997.

[8] T. Kato, T. Ogishi, A. Idoue and K. Suzuki, “Design of Protocol Monitor Emulating Behaviors of TCP/IP Protocols,” in Proc. of IWTCS ’97, Sep. 1997.

[9] J. Padhye and S. Floyed, “On Inferring TCP Behavior,” in Proc. of SIGCOMM ’2001, Aug. 2001.

[10] Fyodor, “Remote OS Detection via TCP/IP Stack Fingerprinting,”http://www.insecure.org/nmap/nmap-fingerprinting-article.html, Dec. 1998.

[11] R. Gecse and P. Krémer, “Automated Test of TCP Congestion Control Algorithm,” in Proc. of IWTCS ’99, Sep. 1999.

[12] “OSI - Open System Interconnection, Conformance testing methodology and framework,” ISO/IEC 9646, 1997

[13] “tcpdump/libpcap Homepage,” http://www.tcpdump.org.

[14] “Pittsburgh Supercomputing Center (PSC) Homepage,” http://www.psc.edu/networking/tcp.html

[15] K. Fall and S. Floyd, “Simulation-based Comparisons of Tahoe, Reno, and SACK TCP,” ACM Computer Communication Review, Jul. 1996.

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

[17] M. Allman, V. Paxson and W. Stevens, “TCP Congestion Control,” RFC 2581, Apr. 1999.