Intelligent Protocol Analyzer with TCP Behavior Emulation for Interoperability Testing of TCP/IP Protocols

*Toshihiko Kato, Tomohiko Ogishi, Akira Idoue and
Kenji Suzuki*

KDD R&D Laboratories

2-1-15, Ohara, Kamifukuoka-shi, Saitama 356, Japan

E-mail : {kato, ogishi, idoue, suzuki}@hsc.lab.kdd.co.jp

Abstract

Recently, the TCP/IP protocols are widely used and it is mentioned that, in some cases, throughput is limited due to problems such as network congestion. To solve such problems, the details of communication need to be examined. In order to support such examination, we are developing an 'intelligent' protocol analyzer which can estimate what communication has taken place by emulating the behaviors of the TCP protocol entity in a pair of communicating computers. Since modern TCP includes some internal procedures for the flow control, such as the slow start algorithm, this analyzer can emulate these procedures as well as the state transition based behaviors of the basic TCP. This paper describes the overview of the analyzer, the detailed design of the TCP behavior emulation function and the implementation results.

Key words

Protocol testing, Interoperability testing, TCP/IP, Protocol Behavior Emulation, Industrial usage report

1 INTRODUCTION

Recently, the TCP/IP protocols[1,2] are widely used in various computer communications. Here, most users of computers use the communication functions installed in operating systems or commercial software products as they are, and do not pay attentions to their details. So, the protocol testing is not focused on for the TCP/IP protocols. For example, the conformance testing is not performed for TCP/IP software products, and the only tools for the interoperability testing are commercially available protocol analyzers which just monitor networks.

Actually, most of the commercial TCP/IP software products can communicate with each other without the conformance testing. However, it is mentioned that the TCP/IP protocols, especially TCP (Transmission Control Protocol), have some problems such that the performance is degraded due to the congestion avoidance algorithms[2,3], and due to the incompatibility between send and receive socket buffers[4]. When such problems occur, the details of communications need to be examined to detect the problem sources. Those problems occur in the actual communications, and therefore the interoperability testing needs to be performed for the TCP/IP protocols. As mentioned above, commercial protocol analyzers such as [5] are used for this purpose, but they do not have enough functionality. They only have the functions to capture PDUs (Protocol Data Units) transmitted over networks, to analyze only their formats, and to present those results. The analysis of protocol behaviors and the investigation of problem sources need to be performed manually by TCP experts.

In order to support of the interoperability testing of TCP/IP communications, we have proposed an 'intelligent' protocol analyzer which can estimate what communication has taken place[6]. This analyzer maintains the specification of the state transition based behaviors of TCP and emulates the behaviors of the TCP protocol entity in communicating computers. Since modern TCP includes some internal procedures for the flow control, such as the slow start algorithm, this analyzer can emulate these procedures as well. Based on the design described in [6], we have finished the implementation of the analyzer as software running in workstations. This paper describes the overview of our intelligent protocol analyzer, the mechanism of the TCP behavior emulation and the implementation results.

2 OVERVIEW OF INTELLIGENT PROTOCOL ANALIZER

- (1) As depicted in Fig. 1, our intelligent protocol analyzer is attached to a LAN and observes it. It captures PDUs transmitted over the LAN, and analyzes PDU formats and emulates the TCP behavior of a specific pair of computers, e.g. computers A and B in the figure.

Figure 1 Network configuration using intelligent protocol analyzer.

- (2) The analyzer function is implemented as a software running in UNIX workstations. Figure 2 depicts the software structure of the analyzer. It consists of the capture module and the TCP emulation module. The capture module captures PDUs transmitted over the LAN, analyzes their format and parameter values according to the TCP/IP protocols, and saves those results in the PDU log. The TCP emulation module selects PDUs which a pair of computers sent and received into the event sequence log for individual computers, and emulates the TCP behaviors of these computers according to the event sequence logs.

Figure 2 Software structure of intelligent protocol analyzer.

- (3) The event sequence log contains TCP protocol events, each of which is a “sent TCP segment” or a “received TCP segment”. The TCP emulation module maintains the state transition specification for received events and sent events, and processes each event according to the following procedure.
- When the event is a received TCP segment, it looks up the specification for received events, and performs a corresponding state transition. If it sends out a segment, the module checks a sent TCP segment in the event sequence and emulates the received and sent segments.
 - When the event is a sent TCP segment, the TCP emulation module looks up the specification for sent events and checks whether the TCP protocol entity can send out the segment. If there is no input to generate the segment, it

decides that the TCP protocol entity has some protocol errors.

3 DETAILS OF TCP EMULATION

3.1 Overview of TCP Behaviors

Behaviors of Basic TCP

TCP provides a reliable data transfer over an unreliable network layer protocol, IP. In order to transfer data, it establishes a connection which is identified by source and destination ports together with IP addresses in the IP header. The connection establishment is performed by the three way handshake using SYN, SYN+ACK and ACK segments. Data transferred over a TCP connection is identified by use of 32 bit sequence number, and acknowledged by 32 bit acknowledgment number. The flow control is performed by use of 16 bit window size advertised by a receiver. In the end of the data transfer, a TCP connection is released by the exchange of FIN and ACK segments by both side.

Internal Procedures of Modern TCP

Most of Modern TCP software products contain four internal procedures to perform a flow control with congestion control and retransmission; slow start, congestion avoidance, fast retransmit, and fast recovery[3]. The basic ideas are ;

- In order to solve the problem that some segments are lost by the network congestion in the case that there is a slower link between sender and receiver, the sender uses a congestion window, called *cwnd*, which control the sending rate internally in the sender, and a slow start threshold size, called *ssthresh*.
- In order to improve the throughput under moderate congestion, especially for large windows, a sender retransmits a segment which is consider to be lost detected by the reception of duplicate ACKs and, after that, it invokes congestion avoidance.

The procedures are summarized as follows.

- (1) The sender never send more than the minimum of *cwnd* and the advertised window from the receiver.
- (2) Congestion is indicated by a timeout or the reception of duplicate ACKs. If the congestion is indicated by a timeout, the sender sets *ssthresh* to one-half of current window size (the minimum of *cwnd* and the advertised window), and sets *cwnd* to one segment.
- (3) For duplicate ACKs, that is, when the sender receives three duplicate ACKs, it considers that these ACKs indicate a lost segment, sets *ssthresh* to one-half of the current *cwnd*, and retransmits the missing segment. After that, when the first ACK arrives that acknowledges new data, the sender sets *cwnd* to

ssthresh.

- (4) Every time when new data is acknowledged by the receiver, cwnd is increased in the following way. If cwnd is less than or equal to ssthresh, TCP is in slow start and cwnd is incremented by one segment every time an ACK is received. The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, cwnd is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, cwnd is increased to four. This opens the window exponentially. If cwnd is greater than ssthresh, congestion avoidance is being performed and cwnd is incremented by $\text{segsize} * \text{segsize} / \text{cwnd}$ each time an ACK is received, where segsize is the segment size. This is a linear growth of cwnd, and the increase in cwnd is at most one segment for each round-trip time.

3.2 Principles of TCP Emulation

- (1) The TCP emulation module maintains a specification of TCP behaviors in response to sent and received segments for individual states of a TCP connection. For a sent segment, it specifies what occurs in a TCP protocol entity to send the segment. For a received segment, it specifies what occurs when the segment is processed. The specification includes both the basic TCP behaviors and the internal procedures of modern TCP.
- (2) In order to specify the basic TCP behaviors, the TCP emulation module maintains the state and internal variables corresponding to those defined in RFC 793[1].
- (3) The invocation of the internal procedures is estimated from the input and output of TCP segments. For example, the invocation of the slow start algorithm is estimated when a connection is established or when it detects a DATA segment retransmission caused by timeout (not duplicated ACKs).
- (4) In order to specify the internal procedures, the TCP emulation module maintains a status which specifies what procedure is being performed, and the internal variables associated with them.
- (5) The TCP emulation module estimates the behavior of a TCP protocol entity by tracing a state transition, based on the specification, for events saved in the event sequence log on an event by event basis.

3.3. Specification of TCP Behavior

State and Internal Variables for Basic TCP

The state takes the following values representing the behavior of basic TCP for the connection establishment and the data transfer phases:

CLOSED, SYN_SENT, SYN_RCVD, and ESTABLISHED.

The internal variables include the followings:

send sequence variables such as

- SND.NXT : the send sequence number to be sent next,
- SND.UNA : the least send sequence number which is unacknowledged,
- SND.WND : the advertised window offered by the remote side,
- SND.MAX : the highest sequence number which has been already sent,
- SND.WL1 : the sequence number of the latest received segment which updated SND.WND,
- SND.WL2 : the acknowledgment number of the latest received segment which updated SND.WND,
- ISS : the initial send sequence number, and
- MSS : the maximum segment size, and

receive sequence variables such as

- RCV.NXT : the receive sequence number to be received next,
- RCV.WND : the advertised window offering to the remote side,
- RCV.UNA : the least receive sequence number which is unacknowledged,
- RCV.ADV : the maximum receive sequence number which can be received by the advertised window, and
- IRS : the initial receive sequence number.

Status and Internal Variables for Internal Procedures

The TCP emulation module maintains the status on the internal procedures, STATUS, which indicates what algorithm is being emulated. It takes the values: NORMAL, SS (slow start), CA (congestion avoidance), and FR (fast retransmit). It also maintains the following internal variables associated with the slow start and congestion avoidance algorithms and the fast retransmit and fast recovery algorithms:

variables associated with the slow start and congestion avoidance

- CWND : the estimated congestion window,
- SSTHRESH : the estimated slow start threshold, and

variables associated with the fast retransmit and fast recovery

- D_ACK : number of the received duplicated ACK.

State Transitions of TCP

By use of the state and internal variables described above, the specification of TCP behavior for the TCP behavior emulation is defined as follows. Figure 3 shows the state transitions of TCP for the connection establishment and data transfer phases.

A state transition for one event and for one state is associated with one or more possibilities. A possibility includes condition, output and requirements for its parameters (only for a received event), next state, and variable update. The

State Transition 1 : *SYNsent* in CLOSED

```
variable update:  ISS = sequence number (SEQ) in SYN; SND.UNA = ISS;
                  SND.NXT = ISS+1; SND.MAX = SND.NXT; RCV.WND = window size
                  (WND) in SYN; MSS = maximum window size (MSS) in SYN;
                  CWND = MSS; SSTHRESH = 65535; STATUS = SS;
```

1) output: *ACK* with $SEQ = SND.NXT$
and acknowledgment number (*ACK*) = SEQ in $SYN+ACK+1$;

```

variable update:  IRS = SEQ in SYN+ACK;
                  SND.UNA = ACK in SYN+ACK;
                  SND.WND = WND in SYN+ACK; RCV.NXT = IRS+1;
                  RCV.WND = WND in ACK;  RCV.UNA = ACK in ACK;
                  RCV.ADV = RCV.UNA + RCV.WND;
                  MSS = min (current MSS, MSS in SYN+ACK);

```

State Transition 6 : *DATA*sent in ESTABLISHED

```
variable update:  SSTHRESH =
                  max (2*MSS, 1/2 * min (CWND, SND.WND);
                  CWND = MSS; STATUS = SS;
```

2) condition: $SND.NXT \leq SEQ$ in $DATA < SND.UNA + SND.WND$

next state: ESTABLISHED:

```

variable update:  SND.NXT = SEQ in DATA + length (LEN) of DATA;
                  RCV.NXT = ACK in DATA; RCV.WND = WND in DATA;
                  if (STATUS != NORMAL && SEQ in DATA + LEN of DATA
                      > SND.UNA + CWND)  STATUS = NORMAL;
                  /* Check whether the internal procedures are used at every DT sending. */

```

State Transition 9: ACKrcv in ESTABLISHED

```

1) condition: D_ACK == 2 and ACK in ACK == SND.UNA
output: DATA with SEQ in DATA == SND.UNA;
next state: ESTABLISHED;
variable update:  SSTHRESH =
                  max (2*MSS, 1/2 * min (CWND, SND.WND) );
                  CWND = SSTHRESH+3*MSS; D_ACK = 0; STATUS = FR;
                  /* This corresponds to the fast retransmit. */
2) condition: D_ACK < 2 and ACK in ACK == SND.UNA
next state: ESTABLISHED;
variable update:  if ( STATUS != FR ) D_ACK = D_ACK + 1;
                  else CWND = CWND+MSS;
3) condition:  SND.UNA < ACK in ACK <= SND.NXT
next state: ESTABLISHED;
variable update:
                  SND.UNA = ACK in ACK; SND.WND = WND in ACK; D_ACK = 0;
                  if ( STATUS == FR ) CWND = SSTHRESH; STATUS = CA;
                  /* This is the start of congestion avoidance following the fast retransmit. */
                  if ( STATUS == SS )
                      if ( CWND <= SSTHRESH ) CWND = CWND+MSS;
                      else CWND = CWND+MSS*MSS/CWND; STATUS = CA;
                  if ( STATUS == CA )
                      CWND = CWND+MSS*MSS/CWND;
                      if ( CWND > 65535 ) STATUS = NORMAL; CWND = 65535;
                  /* These two are the slow start and congestion avoidance. */

```

3.4 Emulation Based on Specification

By use of the specification defined in the previous subsection, the TCP emulation module traces the behaviors of TCP protocol entity. The algorithm is summarized as follows.

- (1) The TCP emulation module reads out an event from the event sequence log one by one.
- (2) If the event is a sent TCP segment, the module looks up the corresponding transition, and selects a possibility considering values of segment parameters and internal variables. If a correct one is found, the module emulates it, including changing the state to the next state and updating the internal variables.

If such a possibility does not exist, the TCP emulation module considers that there may be some protocol errors, and resets the state and internal variables to restart the emulation.

- (3) If the event is a received TCP segment, the TCP emulation module looks up the state transition for the current state and the received segment. If the transition does not send out any outputs, then the transition is emulated.

If the transition sends out some outputs, the TCP emulation module looks for the next sent event in the event sequence log. If the next sent event is the correct output for the transition being traced, then the TCP emulation module reads out the sent event, and emulates the current received event and the sent event.

If the correct output is not found, the TCP emulation module supposes that there are no outputs for the event. If there is a possibility with no output for the transition, then the possibility is emulated. If there are no possibilities which do not send out any output for the transition, then the TCP emulation module considers that the corresponding frame is lost in the network and that the current received event did not take place in this protocol entity.

According to this algorithm, the behaviors of computer A in Fig. 4 are emulated in the following way.

- (i) First, the TCP emulation module reads out an event *SYNsent* and selects **State Transition 1** for this event. The new state is *SYN_SENT*.
- (ii) Next, the module reads out an event *SYN+ACKrecv* and looks up **State Transition 4**. Since this transition includes some outputs, then the module looks for the next sent event in the event sequence log, and finds an event *ACKsent*. This output conforms to possibility 1) in **State Transition 4**, *SYN+ACKrecv* and *ACKsent* are emulated. The new state is *ESTBLISHED*.
- (iii) Then, the module reads out an event *DATAsent* and looks up **State Transition 6**. According to the value of sequence number, possibility 2) is selected and the check of the slow start in this possibility is also passed.

Figure 4 Example of communication sequence.

- (iv) Then, the module reads out an event *ACKrecv* and looks up **State Transition 9**. According to the value of acknowledgment number, possibility 3 is selected. Since current STATUS is SS, the value of CWND is updated from 1460 to 2920.
- (v) Similarly with step (iii), the following two *DATAsent* is emulated according to possibility 2) of **State Transition 4**.

4 IMPLEMENTATION RESULTS

4.1 Program Structure

According to the detailed design described in section 3, we have implemented the intelligent protocol analyzer. The analyzer is implemented as a software running over SUN workstations with Solaris 2.5. As depicted in Fig. 5, the software works together with the functions in the kernel space. The Capture module uses the DLPI (Data Link Provider Interface) module in the kernel in order to read frames through the Ethernet device driver. It relies on the BUF module in the kernel to assign timestamps to captured frames. Asking the kernel to assign timestamps makes the assigned timestamp values accurate.

The software of the intelligent protocol analyzer consists of capture module, TCP emulation module and GUI module. This structure corresponds to that described in Fig. 2. The capture module uses a dedicated thread for reading frames from the network, reading thread in the figure. The rest of this module stores the information of captured frames in the PDU log. The TCP emulation module performs the protocol emulation for a specified pair of computers according to the method described in the previous section. The GUI module provides a graphic window interface for operators of the analyzer. This module uses Tcl/Tk for implementing GUI.

Figure 5 Program modules of intelligent protocol analyzer.

4.2 User Interface

Figure 6 shows an example of a window for TCP emulation. In the top part of the window, there are some buttons for display control commands. The middle part of the window gives a list of communication sequence between a pair of computers, which are indicated by 'SYSTEM A' and 'SYSTEM B'. For one computer, there are two rectangles assigned, the left of which indicates sent or received segments and timestamp, and the right of which indicates the parameter values of the segment and the estimated value for the state and the internal variables. The bottom part of the window shows the full information of the selected line in the rectangles of the middle part.

Figure 6 An example of graphical user interface.

4.3 Example of TCP Estimation

This subsection gives examples of the results of TCP emulation generated actually by the intelligent protocol analyzer. When SYSTEM A sends a SYN segment, the analyzer gives the following estimation.

```
A 20:12:46.046386 SYN->
  DstP=1686 SEQ=737600201 CTL=S WIN=24820 MSS=1460
+ [STATE=SYN-SENT snd_una=737600201 snd_wnd=???
```

```

+ snd_nxt=737600202 snd_max=737600202 snd_wl1=0
+ snd_wl2=0 rcv_nxt=??? rcv_wnd=24820 rcv_una=???
+ rcv_adv=??? iss=737600201 irs=0 mss=1460 cwnd=1460 ssthresh=65535]
+ *** Starting slow start emulation. ***

```

Note that the internal variables are written in lower case and that the state names are different from those in section 3. When a SYN segment is sent, some internal variables such as `snd_wnd` and `rcv_nxt` are not determined in the initiator side and denoted as '???'. The emulation of the slow start in this side is started from this timing, and `cwnd` is set to 1460 (one MSS) and `ssthresh` is set to 65535.

Next, the SYN segment sent by SYSTEM A is received by SYSTEM B and a SYN+ACK segment is replied. The analyzer emulates this sequence and produce the following results.

```

B 20:12:46.046422 ->SYN
  DstP=1686 SEQ=737600201 CTL=S WIN=24820 MSS=1460
B 20:12:46.046830 <-SYN,ACK
  SrcP=1686 SEQ=37298289 ACK=737600202 CTL=SA
+ WIN=8760 MSS=1460
+ [STATE=SYN-RCVD snd_una=37298289 snd_wnd=24820
+ snd_nxt=37298290 snd_max=37298290 snd_wl1=0
+ snd_wl2=0 rcv_nxt=737600202 rcv_wnd=8760
+ rcv_una=737600202 rcv_adv=737608962 iss=37298289
+ irs=737600201 mss=1460 cwnd=1460 ssthresh=65535]
+ *** Starting slow start emulation. ***

```

By this emulation, all the internal variables have the estimated values in the responder side and the emulation of the slow start is also started.

When SYSTEM A receives the SYN+ACK segment, it replies an ACK segment which is then received by SYSTEM B. The results of emulation for those are as follows.

```

A 20:12:46.046866 SYN,ACK<-
  SrcP=1686 SEQ=37298289 ACK=737600202 CTL=SA
+ WIN=8760 MSS=1460
A 20:12:46.047122 ACK->
  DstP=1686 SEQ=737600202 ACK=37298290 CTL=A WIN=24820
+ [STATE=ESTABLISHED snd_una=737600202 snd_wnd=8760
+ snd_nxt=737600202 snd_max=737600202
+ snd_wl1=37298289 snd_wl2=737600202 rcv_nxt=37298290
+ rcv_wnd=24820 rcv_una=37298290 rcv_adv=37323110
+ mss=1460 cwnd=1460 ssthresh=65535]
B 20:12:46.047158 ->ACK
  DstP=1686 SEQ=737600202 ACK=37298290 CTL=A WIN=24820
+ [STATE=ESTABLISHED snd_una=37298290 snd_wnd=24820
+ snd_nxt=37298290 snd_max=37298290 snd_wl1=737600202
+ snd_wl2=37298290 rcv_nxt=737600202 rcv_wnd=8760

```

```
+ rcv_una=737600202 rcv_adv=737608962 mss=1460
+ cwnd=1460 ssthresh=65535]
```

Since the communication monitored is the data connection for ftp, SYSTEM A then begin transmitting data. Now the current cwnd is 1460 in the initiator size, and only one data segment is sent. The sending and receiving of the data segment are emulated in the following way.

A 20:12:46.049571 DT->

```
  DstP=1686 SEQ=737600202 ACK=37298290 CTL=PA
+ WIN=24820 LEN=1460
+ [STATE=ESTABLISHED snd_una=737600202 snd_wnd=8760
+ snd_nxt=737601662 snd_max=737601662
+ snd_wl1=37298289 snd_wl2=737600202 rcv_nxt=37298290
+ rcv_wnd=24820 rcv_una=37298290 rcv_adv=37323110
+ mss=1460 cwnd=1460 ssthresh=65535]
```

B 20:12:46.050771 ->DT

```
  DstP=1686 SEQ=737600202 ACK=37298290 CTL=PA
+ WIN=24820 LEN=1460
+ [STATE=ESTABLISHED snd_una=37298290 snd_wnd=24820
+ snd_nxt=37298290 snd_max=37298290 snd_wl1=737600202
+ snd_wl2=37298290 rcv_nxt=737601662 rcv_wnd=8760
+ rcv_una=737600202 rcv_adv=737608962 mss=1460
+ cwnd=1460 ssthresh=65535]
```

Then SYSTEM B sends an ACK segment for the received data segment as follows. We can estimate that this ACK is sent by 200 msec timeout from the timestamp for this segment.

B 20:12:46.222792 <-ACK

```
  SrcP=1686 SEQ=37298290 ACK=737601662 CTL=A WIN=8760
+ [STATE=ESTABLISHED snd_una=37298290 snd_wnd=24820
+ snd_nxt=37298290 snd_max=37298290 snd_wl1=737600202
+ snd_wl2=37298290 rcv_nxt=737601662 rcv_wnd=8760
+ rcv_una=737601662 rcv_adv=737610422 mss=1460
+ cwnd=1460 ssthresh=65535]
```

When SYSTEM A receives this ACK segment, it increments its cwnd from 1460 to 2920 and sends two data segments. These behaviors in SYSTEM A are emulated as follows.

A 20:12:46.222828 ACK<-

```
  SrcP=1686 SEQ=37298290 ACK=737601662 CTL=A WIN=8760
+ [STATE=ESTABLISHED snd_una=737601662 snd_wnd=8760
+ snd_nxt=737601662 snd_max=737601662
+ snd_wl1=37298290 snd_wl2=737601662 rcv_nxt=37298290
+ rcv_wnd=24820 rcv_una=37298290 rcv_adv=37323110
+ mss=1460 cwnd=2920 ssthresh=65535]
```

A 20:12:46.223298 DT->

```

      DstP=1686 SEQ=737601662 ACK=37298290 CTL=A
+   WIN=24820 LEN=1460
+   [STATE=ESTABLISHED snd_una=737601662 snd_wnd=8760
+   snd_nxt=737603122 snd_max=737603122
+   snd_wl1=37298290 snd_wl2=737601662 rcv_nxt=37298290
+   rcv_wnd=24820 rcv_una=37298290 rcv_adv=37323110
+   mss=1460 cwnd=2920 ssthresh=65535]
A 20:12:46.224529 DT->
      DstP=1686 SEQ=737603122 ACK=37298290 CTL=PA
+   WIN=24820 LEN=1460
+   [STATE=ESTABLISHED snd_una=737601662 snd_wnd=8760
+   snd_nxt=737604582 snd_max=737604582
+   snd_wl1=37298290 snd_wl2=737601662 rcv_nxt=37298290
+   rcv_wnd=24820 rcv_una=37298290 rcv_adv=37323110
+   mss=1460 cwnd=2920 ssthresh=65535]

```

When SYSTEM B receives these two data segments, it sends one ACK segment immediately according to the rule that every other data segment is acknowledged[2]. When SYSTEM A receives this ACK segment, then it increments its cwnd by one segment size from 2920 to 4380 in the following way.

```

A 20:12:46.227211 ACK<-
      SrcP=1686 SEQ=37298290 ACK=737604582 CTL=A WIN=8760
+   [STATE=ESTABLISHED snd_una=737604582 snd_wnd=8760
+   snd_nxt=737604582 snd_max=737604582
+   snd_wl1=37298290 snd_wl2=737604582 rcv_nxt=37298290
+   rcv_wnd=24820 rcv_una=37298290 rcv_adv=37323110
+   mss=1460 cwnd=4380 ssthresh=65535]

```

5 DISCUSSIONS

(1) It is considered that the intelligent protocol analyzer is used effectively for the detailed analysis of TCP/IP communications. Especially, it is helpful to analyze the behavior of TCP internal procedures for flow control. For example, the analyzer can estimate the numbers of invocation of the slow start and congestion avoidance algorithms. It can also estimate the number of DATA segment retransmission. Since TCP/IP is rather a mature protocol, it is considered that available protocol software does not include so many protocol errors. The intelligent protocol analyzer is used to analyze the details of communication in the case of problems such as the throughput degradation.

(2) It is possible that the intelligent protocol analyzer performs a wrong estimation of event sequence and a wrong TCP emulation. For example, the estimation of event does not take account of the buffering delay in routers. When such buffering delay is larger than propagation delay and transmission time, estimated

event sequence by the analyzer may be wrong in the actual processing order. In order to cope with such cases, it is required to reorder event sequence when any protocol error is detected during the emulation. It is possible to apply the rule based programming to implement such reordering based on heuristic algorithm[7].

(3) By analyzing the behaviors of commercial TCP/IP software by the analyzer, we have obtained some information such as:

- that the TCP software in the Window 95 operating system does not support the fast retransmit and fast recovery algorithms, and
- that the TCP software in Solaris 2.5 might not conform to the congestion avoidance after the fast retransmit algorithm.

6 CONCLUSION

In this paper, we have described the intelligent protocol analyzer which supports the interoperability testing of TCP/IP protocols. It provides the function which can estimate what communication has taken place by emulating the behaviors of the TCP protocol entity in a pair of communicating computers. Since modern TCP includes some internal procedures for flow control, such as the slow start algorithm, our analyzer can emulate these procedures as well as the state transition based behaviors of the basic TCP. We have also presented the implementation results of the analyzer.

The Intelligent protocol analyzer is effective in analyzing TCP/IP communication, including counting how many times the slow start algorithm is invoked and how many DATA segments are retransmitted by the timeout retransmission and the fast retransmit algorithm.

7 ACKNOWLEDGMENT

The authors wish to thank Dr. H. Murakami, Managing Director of KDD, for his continuous encouragement to this study.

8 REFERENCES

- [1] DARPA Internet Program Protocol Specification (1981) Transmission Control Protocol. *RFC 793*.
- [2] Stevens, W.R. (1994) TCP/IP Illustrated, Vol. 1 : The Protocols. Addison Wesley.
- [3] Stevens, W.R. (1997) TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. *Request for Comments: 2001*.
- [4] Moldeklev, L. and Gunningberg, P. (1995) How a Large ATM MTU Causes

Deadlock in TCP Data Transfer. *IEEE Trans. On Networking*, **3**.

[5] Tekelec (1992) Chameleon User's Manual.

[6] Kato, T., Ogishi, T., Idoue, A. and Suzuki, K. (1997) Design of Protocol Monitor Emulating Behaviors of TCP/IP Protocols. To appear in *Proc. of 10th International Workshop on Testing of Communicating Systems*.

[7] Kato, T., Ogishi, T., Idoue, A. and Suzuki, K. (1996) Design of Protocol Interoperability Testing System based on Rule Base Programming. in *IPSJ SIG Notes, DPS 77-5*. (in Japanese)

9 BIOGRAPHY

Toshihiko Kato is the senior manager of High Speed Communications Lab. in KDD R&D Labs. Since joining KDD in 1983, he has been working in the field of OSI, formal specification and conformance testing, distributed processing, ATM and high speed protocols. He received the B.S., M.E. and Dr. Eng. Degrees of electrical engineering from the University of Tokyo, in 1978, 1980 and 1983 respectively. From 1987 to 1988, he was a visiting scientist at Carnegie Melon University. Since 1993, he has been a Guest Associate Professor of Graduate School of Information Systems, in the University of Electro-Communications.

Tomohiko Ogishi is a member of High Speed Communications Lab. in KDD R&D Labs. Since joining KDD in 1992, he worked in the field of computer communication. His current research interests include the protocol testing on TCP/IP communication. He received the B.S. Degree of electrical engineering from the University of Tokyo in 1992.

Akira Idoue is a research engineer of High Speed Communications Lab. in KDD R&D Labs. Since joining KDD in 1986, he worked in the field of computer communication. His current research interests include implementation of high performance communication protocols and communication systems. He received the B.S. and M.E. Degrees of electrical engineering from Kobe University, Kobe, Japan, in 1984 and 1986 respectively.

Kenji Suzuki is the senior manager of R&D Planning Group in KDD R&D Labs. Since joining KDD in 1976, he worked in the field of computer communication. He received the B.S., M.E. and Dr. Eng. Degrees of electrical engineering from Waseda University, Tokyo, Japan, in 1969, 1972 and 1976 respectively. From 1969 to 1970, he was with Philips International Inst. of Technological Studies, Eindhoven, The Netherlands as an invited student. He received Maejima Award from Communications Association of Japan in 1988, Achievement Award from the Institute of Electronics, Information and Communication Engineers in 1993, and Commendation by the Minister of State for Science and Technology (Persons of scientific and technological research merit) in 1995. Since 1993, he has been a Guest Professor of Graduate School of Information Systems, in the University of Electro-Communications.