10

AUTOMATED TEST OF TCP CONGESTION CONTROL ALGORITHMS

Roland Gecse, Péter Krémer
Conformance Center, Ericsson Ltd.
H-1037 Budapest, Laborc u. 1., Hungary
E-mail: {Roland.Gecse, Peter.Kremer}@eth.ericsson.se

Abstract This paper introduces a new uniform, overall well applicable method for testing TCP implementations fully automatic. Our primary intention was to give an Abstract Test Suite (ATS) for TCP congestion control to provide the Means of Testing (MoT), and to demonstrate it on some of today's implementations. First, we give a short introduction to TCP, and a brief section presents the former trials in automating TCP testing. Then we explain three test cases in detail, and propound the results of their execution on four different platforms. Some interesting faults are also presented. Finally, we point out the benefits of the introduced approach.

Keywords: Conformance test, TCP, TTCN, Internet

1. INTRODUCTION

Nowadays, it is not easy to predict the way that evolution of communication technology goes. Current tendency shows the convergence of telecom and datacom world as Internet technology gets widespread use in commercial telephony applications. It is no question that this will have great influence on both of them. Everybody knows the reliability and robustness of telecom solutions, which are partly achieved by serious testing procedures. Internet, on the contrary, does not put high demands on conformity of its protocols and applications. While it is common in telecom world to black box test communication equipment against standards, conformance testing of Internet protocols

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35567-2_25

G. Csopaki et al. (eds.), *Testing of Communicating Systems*

has no roots. Although products undergo interoperability testing[1] in order to prove the ability to interwork with other vendors' devices, this does not seem to be enough. Certain protocol implementations contain sophisticated features, which can be hardly tested when erroneous. This holds especially for communication procedures of Transmission Control Protocol (TCP) [Postel, 1981]. The Network Working Group of Internet Engineering Task Force (IETF) is writing an Internet draft that documents some known TCP implementation problems [Paxson, 1997]. The outline of this draft closely resembles to the structure of a test purpose. It describes desired procedures and impacts to the communication together with traces of both correct and faulty behaviour. But there is no sufficient instruction for testers on how to perform the tests besides connecting a network analyser and observing long traces. On one hand it can be very difficult to trigger the problem, on the other hand it takes a lot of skilled work to find it in the enormous amount of data. Inspired by this draft, we tried to get another perspective and use our knowledge in Conformance Testing Methodology and Framework (CTMF) [ISO 9646, 1997] to develop automated TCP tests.

After a short summary of TCP we give a state of the art overview on Internet testing and describe some previous work. Then, we present some Test Cases (TCs) in the area of TCP's congestion control algorithms. The rest of the paper summarises our test results and draws a conclusion on our work.

2. TRANSMISSION CONTROL PROTOCOL (TCP)

TCP provides a connection-oriented, reliable, byte stream transport service in the TCP/IP stack. Today's most popular applications, like World Wide Web (WWW) and e-mail, are built upon it, which makes TCP the most used transport protocol of Internet.

The term connection-oriented denotes that every TCP communication takes place through connections. A pair of source and destination side IP addresses and TCP port numbers constitute a connection, which provides a full duplex communication between two endpoints. A connection must be set up before two entities can exchange messages, and it has to be released once the communication having finished. The number of simultaneous connections is limited by the capacity of the underlying media and the number of available ports. This concept makes it possible for more applications to share the same TCP service.

TCP's reliability is expressed by procedures, which make it possible to exchange data over the unreliable Internet Protocol (IP). The application data

[1] The term interoperability testing has a different meaning in Internet. It means examining whether two or more different implementations can interwork, rather than the comparision with a reference implementation. In this paper we keep on using interoperability testing in this context.

is broken into best-sized chunks (i.e. Maximum Segment Size – MSS) and put into the payload of IP packets. Each data byte has a sequence number assigned that is used for keeping track of state (sent/received/lost). Whenever a packet is sent, an acknowledgement (ACK) is expected. That means the other end "acks" the received data. If this ACK does not arrive in a given time, TCP considers the packet as lost, and retransmits data bytes beginning with the last ACKed sequence number. The receiver TCP is required to delay the ACK for sending it piggybacked with data if possible.

Since IP is unreliable, packets can be modified, lost, duplicated, or can arrive out of order. TCP detects damaged packets by maintaining header and data checksum. Lost packets are retransmitted; duplicated packets are deleted; above sequence data is buffered within reasonable bounds.

Unlike other transport layer entities, TCP connections do not occupy a predefined, constant bandwidth. Neither do they give quality of service guaranties. TCP is said to provide a best effort service, which makes it able to operate over networks having a wide range of transmission capacity. The ability to use as much bandwidth as possible is achieved by flow and congestion control algorithms, such as slow start, congestion avoidance, fast retransmit - fast recovery.

According to [Braden, 1989], all these algorithms must be present in each TCP. Although they did not belong to the original recommendation [Postel, 1981], they became part of it during the evolution of the protocol. All three algorithms operate in the ESTABLISHED state of the TCP state transition diagram. Most problems concerning these algorithms do not occur in interoperability workshops, because – from the user point of view – TCP seems to work well, even if these procedures are ignored. Indeed, a faulty TCP, which lacks their implementation, usually performs better than a correct one! But this is done, of course, at the expense of suppressing parallel connections, which are competing for resources in vain. In the worst case scenario a faulty implementation could cause serious problems, such as congestion collapse [Jacobson, 1988].

The congestion control algorithms introduce an interesting non-deterministic feature of TCP, that is, we cannot predict how the message flow of a given connection will look like, even if we are aware of all sender and receiver parameters of the network. This means, unlike many other protocols, we cannot define one particular Message Sequence Chart – MSC for a given operation because there are numerous existing good alternatives.

3. FORMER TRIALS IN TCP TESTING

Many papers deal with TCP test both from telecom and Internet points of view. [Kato, 1997] shows the development of a TCP protocol monitor, which

was later used for HTTP monitoring [Ogishi, 1998]. [Hintelmann, 1997] describes TCP's congestion control algorithms in SDL, and compares several different implementations.

On the Internet side, the latest efforts include [Paxson, 1997], which tried to give a general solution to this problem. There are some more TCP tester applications described in [Parker-Schmechel, 1998]. But they are mainly trace analysers and performance monitors. However, none of them managed to give a uniform, overall well applicable method.

3.1 TESTING VS. MONITORING

Current Internet test procedures include packet capturing in real, congested environment, and then they examine the obtained traces. Measurements are triggered either manually or using special applications to cause a particular action. Some of them also requires modifications in Implementation Under Test (IUT), which is unacceptable for us because we focus sharply on black box testing. In addition, most vendors do not provide the source code along with their product.

We see the most significant problem in monitoring is that whenever a packet is lost, we cannot know whether it was because of network overload or measurement error. It is described in [Paxson, 1997; Ogishi, 1998] that packet filters often include ambiguity in the measurement. This could be due to packet drops, additions, resequencing, timing, or even the placement of packet filters. The authors tried to give a workaround by introducing resampling and prediction. In this case the analyser must take numerous possibilities into consideration. Then some features of IUT have to be included in the analyser as well. This has a great disadvantage of losing the general applicability since analyser gets dependent on IUT. And the original idea of testing disappears behind.

All referenced papers are based on monitoring. This could be because of several reasons. First, since all platforms provide some kind of packet capturing functions, it is easy to acquire and filter packets. The obtained traces can be evaluated easily off-line. Second, they have not thought about testing conformity against recommendation. Monitoring is a good solution for interoperability testing, but not for conformance testing.

Third, emulating a TCP is not easy. The difficulty arises from the architecture of TCP implementations. Almost all of today's TCP/IP stacks are written as a whole unit and hidden inside the kernel. Application Programming Interfaces (APIs) do not allow routines to reroute TCP packets to them.

4. TEST METHODS AND CONFIGURATION

We tried another approach. We use black box method; we do not make any modification to IUT. We use methods and notations of CTMF because we

would like to give a general, widely applicable solution. We test the system in laboratory environment, where we do not have to cope with packet loss because of network overload or other unpredictable events. This prevents us from uncertainty coming from measurement errors described earlier. Instead of processing traces documenting former message exchange on a given connection we emulate the peer communicating entity, and control and observe other end's behaviour remotely. Doing so we can focus entirely on the problem given.

For all of our tests we use the remote test method. The UT role is played by an HTTP server. We use the Web server for bulk data transfer. For test purposes we defined a simple WWW site with some documents having different length and data.

5. MEANS OF TESTING

For the execution of our tests we use the Ericsson proprietary System Certification System (SCS). SCS consists of a set of tools: TTCN Translator translates MP files into EXTEL format, which is then used by Test Component Executor (TCE) that interprets and executes test cases selected from TTCN Manager. The most interesting feature of SCS is the Test Port concept. All Abstract Service Primitives (ASPs) of a given Point of Control and Observation (PCO) have to be mapped onto the Service Primitives (SPs) of the underlying service provider in order to exchange data with IUT. Since different protocols can be based on different services, which use different types of connections. This mapping needs serious programming skills. In order to simplify the task, SCS provides the so called Test Ports. A Test Port is a piece of software that holds routines, which are called after an ASP is received from or before ASPs are sent to TCE. Thus, we get the full control over the SPs.

We developed a new test port for TCP tests. It provides two ASPs for communication with the socket interface: "recv" can only be received, and "send" that can only be sent. The TCP header length and checksum are calculated automatically in the test port, so we do not have to waste time on defining them in the constraints.

6. TEST CASES

We derived an ATS consisting of 12 test cases (TCs). Altogether these provide almost full coverage of TCP congestion control. Our work was, in part, inspired by problems described in [Paxson, 1998], thus some of our TCs are based on scenarios described therein. The ATS has four groups; the first three groups belong to one of the introduced congestion control algorithms. The fourth group contains all the other test cases. For the sake of this paper we describe the three most important congestion control algorithms: slow start,

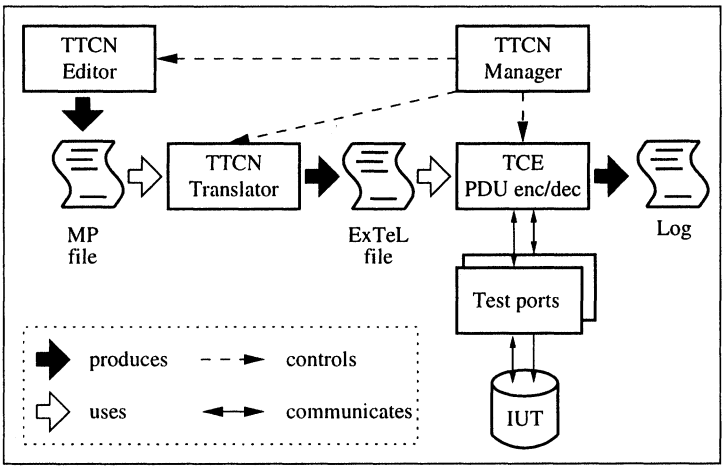

Figure 1 The structure of SCS

congestion avoidance, fast recovery - fast retransmit. But first we should get acquainted with the general ideas of the ATS.

We use Test Case Variables (TCVs) for saving and keeping track of TCP state variables in both sides of the connection. We try to use consistent naming and use the variable names of original TCP recommendation [Postel, 1981] where it is possible. The direction of data flow is tracked from the tester's point of view. Some examples: rcv_mss, for instance, contains the IUT's offered maximum segment size, while snd_mss is the tester's one. The rcv_cwnd holds the value of the current size of IUT's congestion window in bytes, rcv_nxt holds the sequence number of the next data expected from IUT. On the other side, snd_wnd represents the tester's advertised receive window size.

Each receive statement checks whether the received frame has the expected sequence number with the correct source and destination port addresses. The payload of segments is also tested when it is necessary.

Most time we emulate the receiving side of the connection. To fulfill the requirements of [Braden, 1989] we must send acknowledgement messages (ACKs) for incoming data in order to signal the other end that its data has been accepted. The rules for sending ACK are the following: The receiver must send ACK:

1 after the receipt of two full-sized segments or
2 on ACK timer (we call it T_ACK in our ATS) expiry or
3 if there is any data to be sent on the connection.

We must violate the first rule because of our TTCN executor. If we send ACK for every second full-sized segment, we increase IUT's $cwnd$ too aggressively.

But the executor cannot process such amount of data, so sooner or later IUT's RTO timer will expire. We try to avoid this situation since most of the time we examine IUT's transmission (and not retransmission) behaviour. To compensate this deficiency our T_ACK timer is not set to the usual $200\,ms$, but only to $10\,ms$. Our experiences show that TCP implementations send all the segments to be transmitted at once. It means that the delay between the first and the last segment is below $1\,ms$. The third situation rarely takes place since we are mostly receiving data.

We implemented active open and both active and passive close. That means, all kinds of connection set-up and release procedures – including simultaneous and half-close – can be performed regardless of TCP options used.

6.1 SLOW START

The slow start and congestion avoidance [Stevens, 1997] were the first TCP congestion control algorithms. Slow start limits the number of packets that TCP can inject into the network. The algorithm introduces a new TCP state variable called congestion window (*cwnd*). This and the receiver side's advertised window (*rcv_wnd*) give an upper bound to the number of outstanding data bytes per connection. The unit of *cwnd* is measured in bytes or in sender's maximum segment size (*snd_mss*). Its initial value is *snd_mss*, which is increased by at most *snd_mss* bytes for every incoming non-duplicate ACK.

Slow start is used in almost every TC we have written as a preamble, thus we made great effort to implement it as well as possible. But of course, we have to ensure if it is working properly before using it as a test step.

In our test case we observe whether *cwnd* is initialised to 1 *snd_mss*. Nowadays' tendency shows that more and more TCP implementations set this value to 2 *snd_mss* or even higher. Since [Allman, 1999] permits the use of an initial *cwnd* with values less then 2 *snd_mss* but no more than 2 segments, we are prepared to accept initial values, which satisfy these conditions.

Besides the initial value, we also test IUT's rate of *cwnd* increments. The remaining part of this section describes the behaviour from the tester's point of view. After initialising TCVs we do an active open. Having finished the connection set-up we send an HTTP request to the WWW server located on IUT. This request starts downloading a fairly large document. Then we get into a test step performing slow start (Figure 2), till all data is received.

In this test step, we are waiting for the IUT to send the requested data segments while keeping track of its *cwnd*. First, it should send only one full sized segment and wait for acknowledgement. Receiving this ACK, it should increase its *cwnd* by one *snd_mss*. Next, it should send two packets and wait for ACKs and so on. If the IUT transmits more frames than its *cwnd* value, then it does not implement slow start algorithm well, and our TC will fail. We let

Test Step Dynamic Behaviour

Test Step Name : SLOW_START_UNLIMITED (transmitted_bytes:INTEGER)
Group :
Objective :
Default : Default_TMAX
Comments :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1	label1	+RECV_DATAACK			
2		(rcv_data := packet_length – (4 * header.hlen), rcv_nxt := rcv_nxt + rcv_data, snd_una := header.ack, overall_data := overall_data + rcv_data)			
3		[rcv_data <= 0]			
4		GOTO label1			
5		[rcv_data > 0]			
6		(rcv_cwnd := rcv_cwnd + snd_mss)			
7		[rcv_nxt – rcv_una <= rcv_cwnd]			
8		GOTO label1			
9		[rcv_nxt – rcv_una > rcv_cwnd]			
10		IP ! send	send_param (tcphdr_rst_s)	F	
11		? TIMEOUT T_ACK			
12		[rcv_cwnd >= 2 * snd_mss]			
13		IP ! send (rcv_una := rcv_nxt)	send_param (tcphdr_ack_s)		
14		GOTO label1			
15		[rcv_cwnd < 2 * snd_mss]			
16		GOTO label1			
17		+RECV_FIN_OR_DATAFIN			
18		(rcv_data := packet_length – (4 * header.hlen), rcv_nxt := rcv_nxt + rcv_data + 1, snd_una := header.ack, overall_data := overall_data + rcv_data)			
19		[overall_data = transmitted_bytes]		(P)	
20		+PASSIVE_CLOSE			
21		[overall_data <> transmitted_bytes]		(I)	
22		+PASSIVE_CLOSE			
23		+RECV_ANYACK			
24		IP ! send (rcv_una := rcv_nxt)	send_param (tcphdr_ack_s)		
25		? OTHERWISE		(F)	
26		+ACTIVE_CLOSE			

Detailed Comments :

Figure 2 The test step of slow start

IUT increase its *cwnd* constantly, since the data size to be received (100 $kbytes$) is not so much that we should count on packet loss because exceeding the LAN capacity. Moreover, we have to set *snd_wnd* to an appropriate large value (64 $kbytes$) so that IUT's send rate is limited only by its *cwnd*.

Though we do not expect packet loss in this case, the TC can handle it because our TTCN executor is rather slow. When we transmit very large files (more than 1 $Mbytes$), the delay caused by the executor gets the RTO timer expired and IUT will retransmit the appropriate segments.

We run the algorithm until all data has arrived. Then we either do active or passive close and set the verdict to pass. Figure 3 shows the trace of a normal execution from the Tester's point of view. The diagram is based on a trace, which was taken using tcpdump on IUT's side.

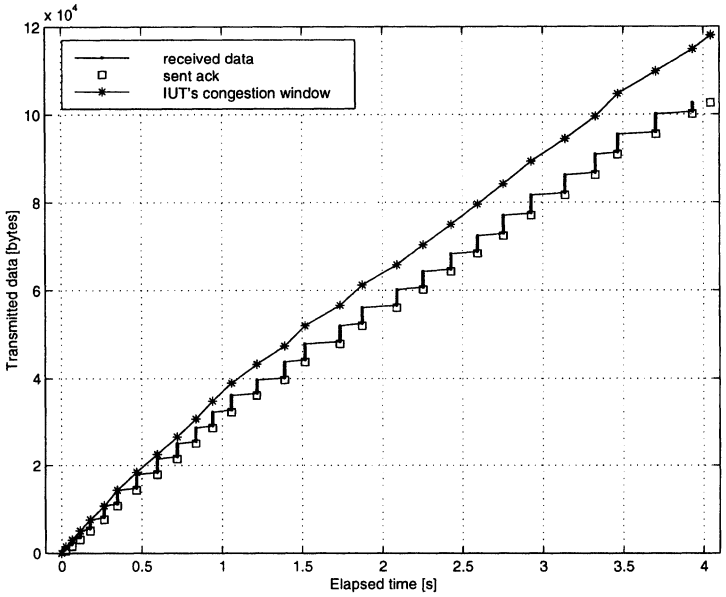

Figure 3 Initial slow start

The dotted line marks the received data (sent by IUT), the squares show the places where the Tester sent acknowledgements (to IUT), and the line with asterisks indicates IUT's congestion window. It can be seen that the network's delay is negligible. The elapsed times between respective ACKs are caused by the executor, e.g. at 3.7 s we receive 8 segments, but send ACK for them at 3.9 s, it means that our delay is about 200 ms. Now let us see the most exciting part of the figure, which is IUT's congestion window. It is clear that IUT did not send more data than its congestion window, since the dotted line does not exceed the line with asterisks. In the beginning $cwnd$ moves away from data, which means that IUT does not send more data than it is permitted by its $cwnd$.

6.2 CONGESTION AVOIDANCE

The congestion avoidance algorithm takes over the control whenever the value of $cwnd$ exceeds $ssthresh$. This algorithm slows down the rate of the growth of $cwnd$ by increasing it at most by one segment size per RTT. The congestion avoidance is performed until $cwnd$ congestion is detected.

Current implementations use two methods for counting the actual size of $cwnd$ during congestion avoidance. Those implementations, counting the size of $cwnd$ in maximum-sized segments calculate the current value of $cwnd$ with the formula $snd\_mss^2/cwnd$. The $cwnd$ is updated on each new incoming

ACK. Some TCPs, on the other hand, that maintain the value of *cwnd* in bytes increase *cwnd* if the number of ACKed data bytes exceeds *cwnd*. However, the value of *cwnd* must not be increased by more than snd_mss bytes.

Some implementations do not follow these rules. They, incorrectly, increase *cwnd* more aggressively than it was described above by a small additive constant. This can diminish performance as described in [Paxson, 1998]. The value of this additive constant is usually $snd\_mss/8$.

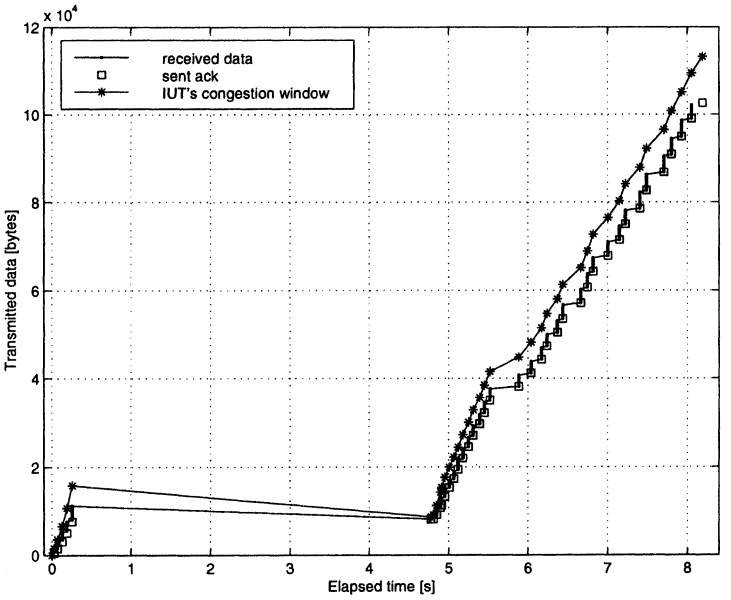

Figure 4 Congestion avoidance

The problem can be hardly found in traces because both *cwnd* and rcv_wnd values have to be large. Its identification requires large amount of data to be sent on a connection. In our TC, we first assist IUT to open its congestion window to 16, then we emulate packet loss without sending three duplicate ACKs until IUT's RTO timer expires. That makes it set $ssthresh$ to 8, *cwnd* to 1 and engage in slow start till *cwnd* does not exceed 8. After that, we take care whether it sends an extra segment after the first 8 full-sized segments. We preset the size of rcv_wnd to $64\,kbytes$ again so that it does not limit the amount of data sent by IUT. Figure 4 shows the correct behaviour.

6.3 FAST RETRANSMIT - FAST RECOVERY

We describe these two algorithms in one section, since they are closely related and usually implemented together (just like slow start and congestion

control). The fast retransmit - fast recovery is described in [Stevens, 1997]. The idea behind these is to avoid the expensive slow start after retransmission timeout. After the receipt of the third duplicate ACK we can suppose with large probability that the packet has been lost, and retransmit it immediately. The fast recovery allows sending of more segments upon receiving more duplicate ACKs until rcv_wnd or $cwnd$ allows. But as soon as new data is ACKed, the sender must retain its $cwnd$ to the half of its original value and continue with congestion avoidance.

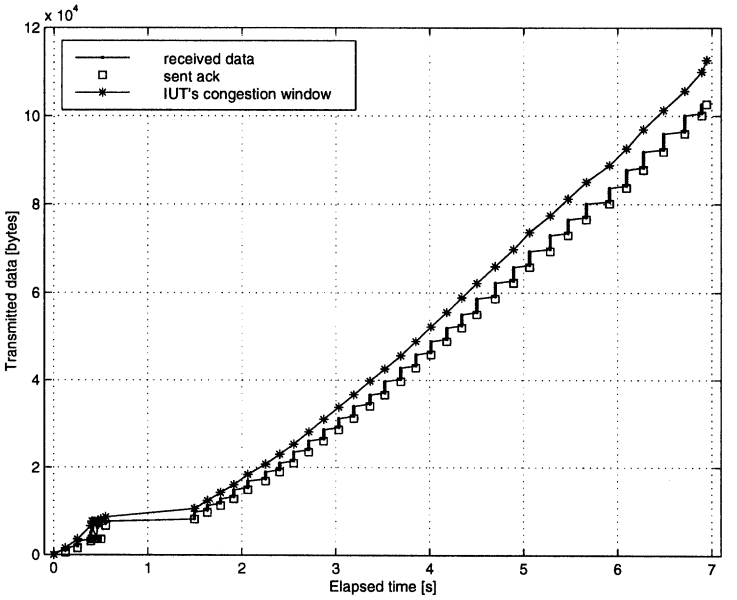

Figure 5 Fast retransmit - fast recovery

The chosen TC was proposed in [Paxson, 1998] as failure of window deflation after loss recovery. This error is quite sophisticated. It occurs after recovery when three subsequent ACKs before RTO time expiry cause IUT's TCP to retransmit the missing segment according to the fast recovery algorithm.

To trigger the problem, we run slow start till IUT's $cwnd$ is 8. That means up to 8 segments of unacknowledged data can be out on the network, if the receiver's advertised window does not limit it. We wait for "some" more frames to arrive and then send 3 subsequent ACKs to a former but still unacknowledged segment. After receiving these ACKs, IUT should set $ssthresh$ to $cwnd/2$, retransmit the missing segment, and increase $cwnd$ by 3. Each time another duplicate ACK arrives it should increase $cwnd$ by one and transmit a packet if its $cwnd$ or our advertised receive window allows. However, if new data is

acknowledged, IUT must set its *cwnd* to the half of its original value, which is stored in *ssthresh*, and continue with congestion avoidance.

Figure 5 shows that IUT applies fast retransmit - fast recovery algorithms well. The $1s$ delay is caused by the TC; we ensure this way that IUT does not send more data than allowed.

7. TEST RESULTS

We executed the 12 TCs on four different platforms. Although most test cases are passed we experienced some minor bugs or non-conformities. Implementation A, for example, failed the TC 'Retransmission sends doubled data' and too low initial RTO was observed on B, C and D platforms. The next subsections discuss the arisen problems in detail. The summary of the results is shown in Table 1.

Test Cases \ Platforms	A	B	C	D
No initial slow start	P	P	P	P
No initial slow start with large window	P	P	P	P
No slow start after retransmission timeout	P	P	P	I
No congestion avoidance	P	P	P	P
Uninitialised cwnd	P	P	P	P
Failure to retain above sequence data	P	P	P	P
Initial RTO too low	P	F	F	F*
Failure of window deflation after loss recovery	P	P	P	P
Failure to back off retransmission timeout	P	P	P	P
Strecth ACK violation	P	P	P	P
Failure to send FIN notification promptly	P	P	P	P
Retransmission sends doubled data	F	P	P	F

F: Fail, I: Inconclusive, P: Pass

Table 1 Results of Test Cases

7.1 RETRANSMISSION SENDS DOUBLED DATA

This TC examines the IUT's retransmission behaviour. We send one ACK after every retransmitted segment, and do not acknowledge the retransmitted segment but an older one. Since this ACK goes for an older data, it does not increase the congestion window, thus IUT must ignore it. The TCP implementation on platform A goes another way as the following trace (made by tcpdump) shows:

```
1. 11:19:11.045884 IUT > Tester: . 3073:3585(512) ack 33 win 9216 (DF)
2. 11:19:11.045952 IUT > Tester: . 3585:4097(512) ack 33 win 9216 (DF)
3. 11:19:11.046022 IUT > Tester: P 4097:4609(512) ack 33 win 9216 (DF)
4. 11:19:11.066923 Tester > IUT: . ack 3585 win 2048 (DF)
5. 11:19:11.067005 IUT > Tester: . 4609:5121(512) ack 33 win 9216 (DF)
6. 11:19:11.067050 IUT > Tester: P 5121:5633(512) ack 33 win 9216 (DF)
```

```
 7. 11:19:15.756634 IUT > Tester: . 3585:4097(512) ack 33 win 9216 (DF)
 8. 11:19:15.769998 Tester > IUT: . ack 3585 win 4096 (DF)
 9. 11:19:15.770056 IUT > Tester: . 3585:4097(512) ack 33 win 9216 (DF)
10. 11:19:15.782244 Tester > IUT: . ack 3585 win 4096 (DF)
11. 11:19:25.216864 IUT > Tester: . 3585:4097(512) ack 33 win 9216 (DF)
12. 11:19:25.230479 Tester > IUT: . ack 3585 win 4096 (DF)
13. 11:19:44.117747 IUT > Tester: . 3585:4097(512) ack 33 win 9216 (DF)
14. 11:19:44.131287 Tester > IUT: . ack 3585 win 4096 (DF)
15. 11:20:21.927916 IUT > Tester: . 3585:4097(512) ack 33 win 9216 (DF)
16. 11:20:21.943187 Tester > IUT: . ack 3585 win 4096 (DF)
```

In row 7, IUT resends segment 3585 : 4097 for the first time. We reply with an ACK for the former segment and IUT retransmits segment 3585 : 4097 because of our ACK. Note that the elapsed time between the last two retransmissions (row 7 and 9) is only 13.422 ms. It is also remarkable that implementation A performs this "doubled retransmission" only after the first retransmission. This inconsistency causes only one single segment to be transmitted in vain, and this single packet also increases the load of the network unnecessarily.

Implementation D also exhibits the similar problem.

7.2 INITIAL RTO TOO LOW

It is quite surprising that almost all examined implementation failed this TC. This phenomenon arises from early BSD TCP implementations [Wright-Stevens, 1995]. It measures the elapsed time using two different timers. One of them is set to expire every 200 ms, the other every 500 ms. The 500 ms timer is used for RTO measurement. For instance the initial RTO is measured by waiting for this timer to expire six times. It means that actual delay could fall between 2500 and 3000 ms. Although the TCs failed, we do not consider this clock skew as a serious failure since – as the following dumps show – this technique is used in almost all of today's TCP implementations.

In implementation B, the measured RTO value is 2976.8 ms, which is fairly close to the recommended 3s:

```
1. 14:15:58.007950 IUT > Tester: . 513:1025(512) ack 33 win 16384 (DF)
2. 14:16:00.984750 IUT > Tester: . 1:513(512) ack 33 win 16384 (DF)
```

On the C platform, the initial value of the RTO timer is probably 3s. The 2994 ms measured RTO by the test case was less then the real value (2999.204 ms), but it is still below the desired 3s:

```
1. 13:20:36.836817 IUT > Tester: P 1:513(512) ack 33 win 32736 (DF)
2. 13:20:39.836021 IUT > Tester: P 1:513(512) ack 33 win 32736 (DF)
```

We also observed this deficiency on our fourth tested platform. The retransmission timer of implementation D is initially set above 3s. The TTCN executor measured the RTO to be 2998 ms, but the real value was 3001.525 ms. The cause of this problem is probably the time-variant feature both of the network

and the executor or the delay in the test case. In the test case we can only start to measure the elapsed time after processing the received data. This inherent delay may consumes the missing $\approx 2\,ms$:

```
1. 18:25:50.534270 IUT > Tester: . 1:513(512) ack 33 win 8160 (DF)
2. 18:25:53.535795 IUT > Tester: . 1:513(512) ack 33 win 8160 (DF)
```

7.3 NO SLOW START AFTER RETRANSMISSION TIMEOUT

Implementation D has a major fault in its TCP implementation. The problem occurs when we send ACK to a retransmitted segment. In row 8 IUT performs the second retransmission (the first was in row 6), we send ACK to an older data and IUT replies with a new segment (row 10). When the retransmission timer expires again, IUT replies to our ACK with two new segments (row 15 and 16). Since we send ACKs for older data, these ACKs do not allow increasing IUT's congestion window. It means that under some circumstances IUT does not perform slow start after retransmission timeout! But if the first retransmission is successful, IUT works well. Although the mentioned test case passed we had to modify the final result to inconclusive, because of this problem.

```
 1. 08:38:27.504181 IUT > Tester: . 3073:3585(512) ack 33 win 8160 (DF)
 2. 08:38:27.504655 IUT > Tester: P 3585:4097(512) ack 33 win 8160 (DF)
 3. 08:38:27.539684 Tester > IUT: . ack 3073 win 2048 (DF)
 4. 08:38:27.540300 IUT > Tester: . 4097:4609(512) ack 33 win 8160 (DF)
 5. 08:38:27.540777 IUT > Tester: . 4609:5121(512) ack 33 win 8160 (DF)
 6. 08:38:29.559949 IUT > Tester: . 3073:3585(512) ack 33 win 8160 (DF)
 7. 08:38:29.573549 Tester > IUT: . ack 3073 win 4096 (DF)
 8. 08:38:33.766114 IUT > Tester: . 3073:3585(512) ack 33 win 8160 (DF)
 9. 08:38:33.779216 Tester > IUT: . ack 3073 win 4096 (DF)
10. 08:38:33.779834 IUT > Tester: P 3585:4097(512) ack 33 win 8160 (DF)
11. 08:38:42.178411 IUT > Tester: . 3073:3585(512) ack 33 win 8160 (DF)
12. 08:38:42.191494 Tester > IUT: . ack 3073 win 4096 (DF)
13. 08:38:42.192108 IUT > Tester: . 3073:3585(512) ack 33 win 8160 (DF)
14. 08:38:42.205000 Tester > IUT: . ack 3073 win 4096 (DF)
15. 08:38:42.205620 IUT > Tester: P 3585:4097(512) ack 33 win 8160 (DF)
16. 08:38:42.206096 IUT > Tester: . 4097:4609(512) ack 33 win 8160 (DF)
```

8. CONCLUSION

Although, in this article, we addressed only a subset of possible problems in TCP, we think that we managed to take the first step towards automated TCP tests. We pointed out the major drawbacks of monitoring techniques, which are in use for current Internet interoperability tests. Then, we introduced how conformance tests could be carried out in TCP implementations. However not excepted, we found some errors while performing our tests on today's most widespreadly used TCPs. This fact justified our thought that interoperability test, as it is used in Internet, does not cover all protocol features. One could think that 12 test cases are not enough to cover the field of congestion control,

but remember that these algorithms could have several good outcomes. Our test cases verify all the correct possibilities, not only one. Hopefully, we could show that there is a high need for testing Internet protocols for conformance.

References

M. Allman (editor): TCP congestion control, <draft-eitf-tcpimpl-cong-control-05.txt>, TCP Implementation Working Group, February 1999.

B. Baumgarten, A. Giessler: OSI conformance testing methodology and TTCN, North holland, 1994.

R. Braden (editor): Requirements for Internet hosts – communication layers, RFC 1122, IETF Network Working Group, October 1989.

R. Gecse: Conformance testing methodology of Internet protocols, Testing of Communicating Systems, Tomsk, Russia, September 1998.

J. Hintelmann, R. Westerfeld: Performance analysis of TCP's flow control mechanisms using queueing SDL, SDL '97: TIME FOR TESTING - SDL, MSC and Trends, Elsevier Science B. V., 1997.

V. Jacobson: Congestion avoidance and control, ACM SIGCOMM '88, Stanford, California, August 1988.

T. Kato, T. Ogishi, A. Idoue and K. Suzuki: Design of protocol monitor emulating behaviours of TCP/IP protocols, Testing of Communicating Systems, Cheju Island, Korea, September 1997.

T. Ogishi, A. Idoue, T. Kato and K. Suzuki: Intelligent protocol analyzer for WWW server accesses with exception handling function, Testing of Communicating Systems, Tomsk, Russia, September 1998.

OSI - Open System Interconnection, Conformance testing methodology and framework, ISO/IEC 9646, 1997.

S. Parker, C. Schmechel: Some testing tools for TCP implementors, RFC 2398, IETF Network Working Group, August 1998.

V. Paxson: Automated packet trace analysis of TCP implementations, ACM SIGCOMM'97, Cannes, France, September 1997.

V. Paxson (editor): Known TCP implementation problems, <draft-ietf-tcpimpl-prob-05.txt>, IETF Network Working Group, November 1998.

J. Postel (editor): Transmission Control Protocol, RFC 793, September 1981.

W. R. Stevens: TCP/IP Illustrated, Volume 1, The Protocols, Addison-Wesley, 1994.

W. R. Stevens: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, RFC 2001, IETF Network Working Group, January 1997.

G. R. Wright, W. R. Stevens: TCP/IP Illustrated, Volume 2, The Implementation, Addison-Wesley, 1995.