# SSFNET TCP Simulation Analysis by tcpanaly

Hongbo Liu
hongbol@winlab.rutgers.edu

Apr. 16, 2000

**Abstract**

SSFNET is a collection of SSF-based models for simulating Internet protocols and networks. It is designed at the logical IP-network level, and the simulation quantum is an IP packet. SSFNET TCP is a independently developed package mainly based on RFCs related to TCP. *tcpanaly* is a tool for automatically analyzing the behavior of a TCP implementation. Use of *tcpanaly* for validation of simulation results can be used with arbitrarily complex topologies and traffic scenarios. With *tcpanaly*, we successfully verified major features of SSFNET TCP implementation. This result represents a major validation stage of the SSFNET TCP implementation. This report concludes with a few suggestions concerning development of new protocol analysis tool building on the experience with *tcpanaly*.

## 1 Introduction

### 1.1 SSFNET

SSFNET [1] is a collection of SSF-based models for simulating Internet protocols and networks. SSFNET models are self-configuring - that is, each SSFNET class instance can be autonomously configured by querying a parameter database, which may be locally resident or available over the Web.

The network configuration files are in the DML format. They are used to synthesize a model and instantiate a simulation with the help of a scalable configuration database package.

To study the dynamics of IP traffic over wired networks, SSFNET is designed at the logical IP-network level, and the simulation quantum is an IP packet. An SSFNET model then consists of realistic IP hosts and routers, abstracted LANs, and wide-area links, but ignores the details of link-level transmission beyond gross characterizations of the bandwidth and transmission delays on links. Link layer and physical layer modeling can be provided in separate packages.

SSFNET TCP is a independently developed package mainly based on RFCs related to TCP. We also refer to the publicly available source code of BSD TCP and related publications [2]. To reduce the complexity and improve the efficiency of simulation, we do not implement all the features of TCP. For example, in TCP header we implemented sequence number, acknowledgment number, source and destination port number, window size. Available flags include SYN, ACK and FIN. We do not have URG, PUSH, RST flags and urgent pointer, options and checksum. As a result, SSFNET TCP never have push and reset operations. In spite of these simplifications, SSFNET can simulate TCP behavior with great details. The interested reader can refer to the SSFNET webpage about TCP implementation [3].

As most of Internet research is based on the trace analysis, *tcpdump* take an important role in these research. To make use of the available trace analysis tools based on *tcpdump*, SSFNET can provide trace in a pseudo binary tcpdump format. It is same as the binary tcpdump except the byte count field in IP header of each packet includes the bytes of pseudo payload which does not exist. That is the reason why there is no way to calculate the checksum. The description of tcpdump format is available everywhere. Here we won't elaborate on it.

### 1.2 tcpanaly

*tcpanaly* is a tool developed by Vern Paxson [4] for automatically analyzing behavior of TCP implementation. It's very powerful in the sense that it analyzes most features of TCP and supports various TCP implementations . It can detect measurement errors and separate it from behavior of TCP. Although, the main objective of this package is to analyze TCP running in real world network and its input is the binary format tcpdump file collected by *tcpdump* and packet filter, we now successfully used it on our pseudo binary tcpdump file. Here "pseudo" means the input file is collected from our network simulator SSFNET instead of a real world network. Our binary file is written in the same format as tcpdump but contains only protocol headers without data payload. That's the reason why we need to adapt *tcpanaly* a bit to our case. Because we have no measurement problem in our trace we only need to focus our attention on *tcpanaly*'s capability to analyze TCP behavior in our simulation.

Basically, *tcpanaly* works in three steps.

The first step is checking if the packet is corrupted during transmission and filtering. The checking list includes IP header length, IP packet checksum, IP option length, IP packet length, offset, protocol number, TCP header length, TCP packet checksum.

The second step is to check if flags within the TCP header are correctly used and consistent with the current state of TCP connection. The diagnostic messages cover almost all possible problems:

- SYN with data
- RST with data
- beginning of connection missing
- connection originator SYN ack
- bad SYN ack
- simultaneous open
- broken establishment
- data with SYN
- SYN FIN
- bad SYN ack ack
- ambiguous initial packet timings
- repeated SYN with ack
- repeated SYN reply without ack
- SYN sequence jump
- FIN before establishment

- RST before establishment
- data sent prematurely
- bad retransmitted SYN ack
- data with SYN ack
- SYN inside connection
- SYN after close
- SYN after reset
- FIN after reset
- data after reset

The third step is analyzing the most important TCP behavior, the sliding window behavior, to find if it sends packets and acknowledges them properly. The possible diagnostic output includes:

- probable measurement duplication
- probable ack measurement duplication
- max offered window grew
- window recision
- ack recision
- duplicate ack of last seq
- ack above a hole
- ack of partial segment
- packet filter ack resequencing
- ack without corresponding segment
- ack of unsent data
- MSS violation
- sub-MSS segment
- receiver window violation
- vantage point problem
- broken retransmission
- partially broken retransmission
- partial retransmission
- partial progress
- skip ahead

Each of above statements is a strong sign of possible implementation error. For example, "window recision" means the acknowledged sequence number added by the advertised window, i.e. the newly allowed sequence number, is less than the maximum allowed sequence number it has ever had. This is a violation of RFC about TCP implementation. And "ack recision" means a packet acknowledges a sequence number that has already been acknowledged before, which is also very suspicious.

In the end, *tcpanaly* will summarize the analysis results and statistical information if the trace passed all the tests. From the result, we also know that which version of TCP is the best match for this trace.

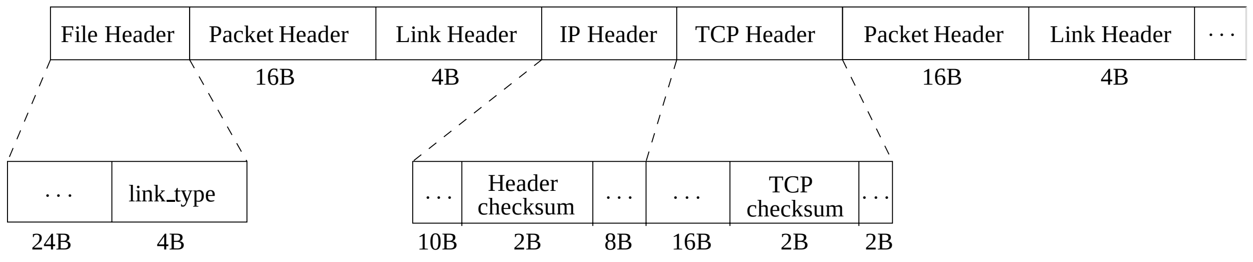

Figure 1: SSFNET pseudo tcpdump file format; B:bytes.

## 2 Verification Procedure

First we make sure the pseudo tcpdump file contains correct header information of the protocol stacks which is the same as the one in real tcpdump file. The underlying guide line for our verification is modifying *tcpanaly* as less as possible. There are still two features need to modify in *tcpanaly*. One is that it need to recognize the link type used by the pseudo tcpdump. *tcpanaly* does define the data link type DLT_NULL, but it does not assign a value to link header length for this link type. We fixed this problem. The other is that it need to ignore the checksum field of the IP and TCP header. SSFNET does not implement the data link layer protocol and assumes no link error happens during transmission. Further more, as indicated in Figure 1, pseudo tcpdump file does not contain TCP packet payload, while the byte count field of IP header contains the length of TCP packet payload. It has no meaning to calculate both the checksum of TCP packet and IP packet.

After these modifications, the pseudo tcpdump file is feed in as the input of *tcpanaly*, then we get the output which is used as a hint to find in what aspect the simulator is different from realistic one. Here are some examples:

```
probable measurement duplication
packets without clock movement
```

The above information tells an important feature of SSFNET, the simulator assume zero processing time which cause the zero time increment.

```
sub-MSS segment
```

It reports this message when it finds a packet with segment length less than one MSS (Maximum Segment Size) but without PUSH flag set in its TCP header. SSFNET TCP doesn't implement PUSH flag and we will see this message whenever the packet with sub-MSS segment is sent. To avoid this message, we use a traffic generator which sends MSS segment only.

```
bidirectional transfer
```

It points out the bidirectional way of our traffic design. Each session begins with a user request in one direction and data transfer in reverse direction.

## 3 Experiments

To verify the correctness of SSFNET TCP implementation using *tcpanaly*, we manage to get pseudo tcpdump of a reasonable big enough network scenario which contains 400 TCP connections.

### 3.1 Experiment settings

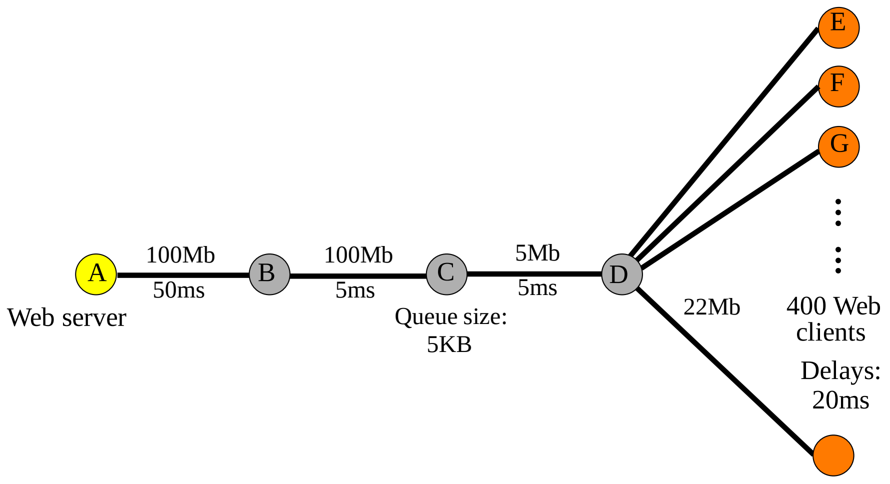

Figure 2: Network topology: CAPBELL/SINGLEBELL; Mb=Mbps, ms=millisecond, KB:Kbytes.

Here we do the experiments based on CAPBELL/SINGLEBELL network model derived from [5]. The basic settings are shown in Figure 2. Node A is a web server. Node B, C, and D are routers. Node E, F, etc are web clients. The traffic source model is of HTTP type. To observe the behavior of TCP under severe congestion, we set the bottleneck bandwidth as 5 Megabits/second and set the drop-tail queue size as 5 Kbytes.

To compare with the analysis results of *tcpanaly*, we list the configuration of TCP as shown below in DML format:

```
tcpinit[
  ISS 10000              # initial sequence number
  MSS 1000               # maximum segment size
  RcvWndSize  32         # receive buffer size
  SendWndSize 32         # maximum send window size
  SendBufferSize 32      # send buffer size
  MaxRexmitTimes 12      # maximum retransmission times before drop
  TCP_SLOW_INTERVAL 0.5 # granularity of TCP slow timer
  TCP_FAST_INTERVAL 0.2 # granularity of TCP fast(delay-ack) timer
  MSL 5.0                # maximum segment lifetime
```

```
    MaxIdleTime 600.0     # maximum idle time for drop a connection
    delayed_ack false     # delayed ack option
    fast_recovery true    # implement fast recovery algorithm
    show_report false     # print a summary connection report
    debug false
  ]
```

## 3.2 Experiment results

The output of *tcpanaly* is shown below:

```
data/hscr5.tcpdump: 1000052.038796/2.097944 2001/18020 probable ack measurement
duplication
**7754 likely measurement duplicates
estimated clock resolution: 100 usec (10 adjustments)
4 packets with no clock movement
104 simultaneous packets
data/hscr5.tcpdump: SERIOUS @1000098.914152/48.973300: ** bidirectional transfer
data/hscr5.tcpdump: SERIOUS @1000141.530083/2.376107: ** bidirectional transfer
                   .
                   .
        (Here we omit 394 lines)
                   .

data/hscr5.tcpdump: SERIOUS @1000107.867772/1.144997: ** bidirectional transfer
data/hscr5.tcpdump: SERIOUS @1000070.086546/39.817511: ** bidirectional transfer
data/hscr5.tcpdump: SERIOUS @1000090.522175/12.389108: ** bidirectional transfer
data/hscr5.tcpdump: SERIOUS @1000151.992078/2.430985: ** bidirectional transfer
```

The above example does not show the statistic result and test summary because *tcpanaly* thinks bidirectional transfer is an abnormal situation and exits prematurely. If we modify *tcpanaly* a little bit to make it ignores this situation, we can see the following message.

```
data/hscr5.tcpdump: 1000052.038796/2.097944 2001/18020 probable ack measurement
duplication
**7754 likely measurement duplicates
estimated clock resolution: 100 usec (10 adjustments)
4 packets with no clock movement
104 simultaneous packets
state 3 (1)
start 1000049.940852, last 1000098.914152, duration 48.9733
size 2000
location: local, min response time 0.040592, max 0.040593 (SYN 0.040229; mean 0.
103363)
estimated RTT 1.608649 sec @1000051.835110/1.894258
        (0.164853/0.160970) (0 Karn)
```

```
estimated rho = 257.533 KB/s, prop time = 0.160760 sec
MSS 1000 (effective 1000)
35 packets, 2 data packets, 2000 bytes
0 retransmit epochs, 0 retransmitted packets (0 unneeded?), 0 bytes
0 redundant acks
min offered window 32000
max offered window 32000
max data in flight 1000
1 closed windows
0 mean IP id advance
0 mean IP id advance
24 IP id reuses
** Only one implementation evaluated
Single unflawed best candidate: reno
reno implementation:
        0 timeouts
        2 maxed out
        0 packets with liberation (mean 0.000000)
        1 packets with last liberation (mean 0.000378)
        0 push delays (mean nan)
        worst last liberation delay = 0.000000 @0.000000/-1000049.940852
        0 fast retransmits
        final cwnd = 3000, final ssthresh = 65536, max cwnd = 3000
        sender window of 1000 (limiting)

state 3 (2)
start 1000049.940852, last 1000098.873560, duration 48.9327
size 37019
location: remote, min response time 0.124261, max 0.124261 (SYN 0.120741; mean 0
.124261)
estimated RTT 1.608649 sec @1000051.835110/1.894258
        (0.164853/0.160970) (0 Karn)
estimated rho = 257.533 KB/s, prop time = 0.160760 sec
MSS 1000 (effective 1000)
1 sub MSS segments
32 packets, 31 data packets, 37019 bytes
1 retransmit epochs, 4 retransmitted packets (0 unneeded?), 4000 bytes
0 redundant acks
min offered window 31000
max offered window 32000
max data in flight 19000
0 mean IP id advance
31 IP id reuses
               .
               .
    (Here we omit the test summary for the rest 399 connections)
               .
```

We do not see any complains from *tcpanaly*, which means SSFNET TCP passes the verification. This example shows that SSFNET TCP implementation can simulate the typical behavior of TCP used in real networks with considerable accuracy. The simulation result is effective and accurate enough for network research.

## 4 Suggestions and future work

*tcpanaly* gives us a good example on how much we can do in a new generation of protocol analysis tool. The limitation of *tcpanaly* lies in that it was designed to analysis individual connection and does not provide overall statistics for all connections. It cannot deal with bidirectional traffic which is a common case in today's traffic analysis. The new tool should provide more statistics about the interaction between multiple connections such as packet loss ratio, bandwidth usage versus the parameter estimation of individual connection. The graphic representation of the analysis result can give us an easy way to find the possible interesting events. Figure 3 is one of the plots used during the early stage of TCP validation. We use this kind of comprehensive plot about the internal variables of a TCP connection to find if there is any inconsistency with the RFCs. Generation of these kinds of figures can also be considered as the new possible features of future *tcpanaly*. Some more enhancement or other tools are also needed to do more thorough analysis for research purpose.

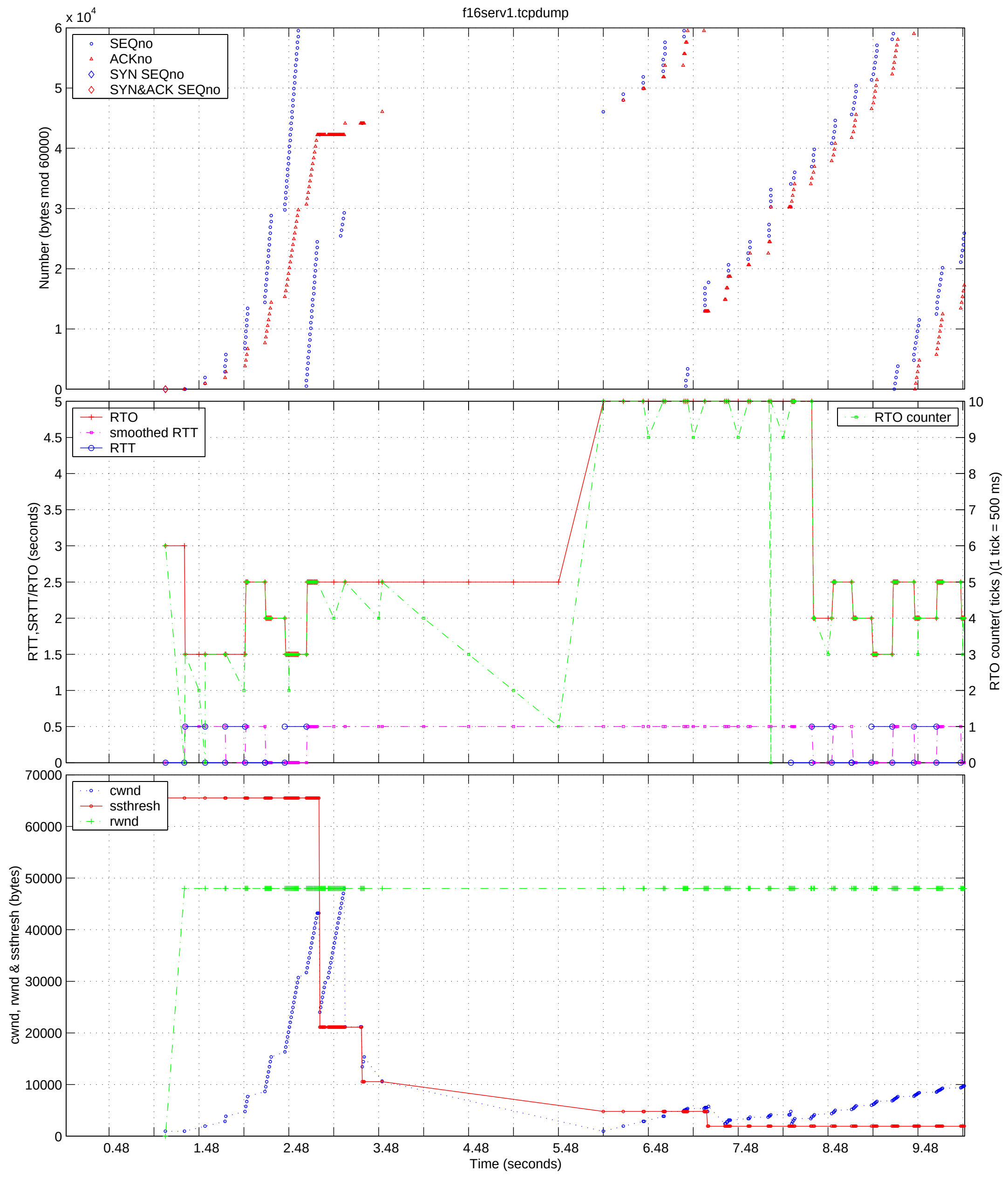

Figure 3: TCP variables time sequence plot for the SSFNET TCP implementation. See [3] for more plots of this type and their analyses for different traffic scenarios

## 5 Acknowledgment

I would like to acknowledge Vern Paxson for sharing the source code of his *tcpanaly* tool, and Anja Feldmann, Andy Ogielski and Walter Willinger for their interest in this work. This work was supported by DARPA grant N66001-96-C-8530 and by AT&T Labs - Research.

## References

[1] SSFNET Homepage, *http://www.ssfnet.org/*

[2] G. Wright and W. Stevens, *TCP/IP Illustrated vol.2: The Implementation*, Addison-Wesley, 1995

[3] SSF TCP Implementation and Validation Tests, *http://ssfnet.org/Exchange/tcp* , Andy T. Ogielski and Hongbo Liu

[4] Vern Paxson *Automated Packet Trace Analysis of TCP Implementations*, Proceedings of SIGCOMM '97

[5] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger, *Dynamics of IP traffic: A study of the role of variability and the impact of control*, Proceedings of ACM/SIGCOMM'99