TCP Socket Options in Android

Vsevolod Geraskin

Table of Contents

INTRODUCTION

The goal of this project is to understand Android TCP/IP socket options and analyze potential scenarios when each option should be used. For some of the socket options, also evaluate the impact on TCP connection. For the above purposes, each TCP socket option is independently set and tested in Android client / Linux server environment on 802.11 wireless network. At the same time, TCP packets are captured and analyzed in Wireshark.

Software used:

- Fedora Release 15 32-bit
- Multiprocess socket server written in C
- Wireshark Packet Capture Analyzing Tool
- Android 4.2.2
- Android socket client written in Java

Hardware used:

- Samsung Tablet: Galaxy Tab 3 8.0
- Compaq Laptop: Intel® Pentium(R) Dual CPU T2390 @ 1.86GHz × 2, 3 gigs of RAM

Both server and client are connected to 802.11n/g/b wireless network provided by Dlink DIR-615 router. Global IP is 24.86.116.244, Linux Server IP is 192.168.0.101, and Android Client IP is 192.168.0.106. Wireshark is capturing packets on the Fedora Linux server.

SOCKET OPTIONS AND TEST SCENARIOS

SO_KEEPALIVE

Description: specifies whether the kernel sends keep-alive messages.
Test: Connect to TCP socket server with SO_KEEPALIVE set to false. Disconnect, then connect to TCP socket server with SO_KEEPALIVE set to true. Using Wireshark, capture and examine keep-alive TCP packet and explore the cases when this options might be used.

SO_LINGER

Description: specifies number of seconds to wait when closing a socket if there is still some buffered data to be sent.
Test: Connect to TCP socket server with SO_LINGER set to false. Send some data, then issue a socket close call right away. Then, connect to TCP socket server with SO_LINGER set to true and interval set to 0 and 5 seconds. Send some data, then issue a socket close call right away. Using Wireshark, capture FIN TCP packet and compare the time intervals.

SO_OOBINLINE

Description: specifies whether sending TCP urgent data is supported on this socket or

not.
Test: Connect to TCP socket server with SO_OOBINLINE set to true. Send urgent data byte to the server. Using Wireshark, capture URG TCP packet and explore the cases when this option might be used.

SO_RCVBUF and SO_SNDBUF

Description: specifies the size in bytes of a socket's receive and send buffers.
Test: Connect to TCP socket server with SO_RCVBUF and SO_RSNDBUF set to 1024, 4096, and 65536 bytes. Set the matching buffer size on the server. For each buffer size, send 10mb file from Android tablet to Linux server. Using Android client, capture the time stamps of start and end of file transfer. Using Wireshark, capture the TCP packets and graph the throughput. Analyze the results, determine what buffer size is optimal (out of three buffer sizes), and explore the reasons why.

SO_TIMEOUT

Description: specifies integer timeout in milliseconds for blocking accept or read/receive operations (but not write/send operations).
Test: Connect to TCP socket server with SO_TIMEOUT set to 0. Then, connect to TCP socket server with SO_TIMEOUT set to 5000 (5 seconds). Attempt to read the socket and wait for more than five seconds. Determine the behaviour in each case, and explore the cases when this option might be used.

TCP_NODELAY

Description: specifies whether data is sent immediately on this socket.
Test: Connect to TCP socket server with TCP_NODELAY set to false. Disconnect, and connect to server with TCP_NODELAY set to true. Using Wireshark, capture the packets and examine the difference in each case. Explore the cases when this option might be used.

TEST RESULTS

TEST 1: SO_KEEPALIVE

a. Connect with SO_KEEPALIVE set to false (Wireshark/test1a.pcap)

Data capture only contains a standard 3-step TCP handshake (SYN, SYN/ACK, ACK packets) and TCP disconnect (FIN/ACK, ACK packets).

b. Connect with SO_KEEPALIVE set to true (Wireshark/test1b.pcap)

Data capture contains a standard 3-step TCP handshake and TCP disconnect. Also, TCP Keep-Alive packet is captured after listening on the wire for two hours (packet 4). Default Keep-Alive interval in Android is two hours, which cannot be changed, thus severely limiting its usability.

TCP Keep-Alive packet:

```
0000  00 1a 73 f6 f3 6e 6c b7  f4 71 14 15 08 00 45 00   ..s..nl. .q....E.
0010  00 34 38 e5 40 00 40 06  7f bf c0 a8 00 6a c0 a8   .48.@.@. .....j..
0020  00 65 a1 99 22 6b 2e 46  64 28 9b 5e ec 7e 80 10   .e.."k.F d(.^.~..
0030  00 e5 ec 51 00 00 01 01  08 0a 02 a2 60 00 08 52   ...Q.... ....`..R
0040  be 21                                             .!
```

TEST 2: SO_LINGER

a. Connect with SO_LINGER set to false (Wireshark/test2a.pcap)

Data capture shows that disconnect is sent right away and 10 MB transfer is interrupted. Connection is normally closed with FIN/ACK packet (red).

Android FIN/ACK packet 404:

```
0000  00 1a 73 f6 f3 6e 6c b7  f4 71 14 15 08 00 45 00   ..s..nl. .q....E.
0010  00 34 31 e8 40 00 40 06  86 bc c0 a8 00 6a c0 a8   .41.@.@. .....j..
0020  00 65 bb 8d 22 6b bf f6  7d dc d9 f6 08 31 80 11   .e.."k.. }....1..
0030  00 e5 d4 5d 00 00 01 01  08 0a 02 52 86 ec 07 31   ...]....  ...R...1
0040  90 f6                                             ..
```

Server ACK packet 405:

```
0000  6c b7 f4 71 14 15 00 1a  73 f6 f3 6e 08 00 45 00   l..q.... s..n..E.
0010  00 34 76 1e 40 00 40 06  42 86 c0 a8 00 65 c0 a8   .4v.@.@. B....e..
0020  00 6a 22 6b bb 8d d9 f6  08 31 bf f6 7d dd 80 10   .j"k.... .1..}...
0030  01 4b 82 46 00 00 01 01  08 0a 07 31 91 24 02 52   .K.F.... ...1.$.R
0040  86 ec                                             ..
```

b. Connect with SO_LINGER set to true and 5 seconds delay (Wireshark/test2b.pcap)

Data capture shows that disconnect is also sent right away and 10 MB transfer is interrupted. Connection is normally closed with FIN/ACK packet (red).. This behaviour is the same regardless of SO_LINGER delay.

Android FIN/ACK packet 740:

```
0000  00 1a 73 f6 f3 6e 6c b7  f4 71 14 15 08 00 45 00   ..s..nl. .q....E.
0010  00 34 7d 32 40 00 40 06  3b 72 c0 a8 00 6a c0 a8   .4}2@.@. ;r...j..
0020  00 65 cb 29 22 6b 63 d0  5b df 95 a3 74 3b 80 11   .e.)"kc. [...t;..
0030  00 e5 aa 45 00 00 01 01  08 0a 02 52 ef 14 07 33   ...E.... ...R...3
0040  99 b4
```

Server ACK packet 741:

```
                                                        ..
0000  6c b7 f4 71 14 15 00 1a  73 f6 f3 6e 08 00 45 00   l..q.... s..n..E.
0010  00 34 1f a1 40 00 40 06  99 03 c0 a8 00 65 c0 a8   .4..@.@. .....e..
0020  00 6a 22 6b cb 29 95 a3  74 3b 63 d0 5b e0 80 10   .j"k.).. t;c.[...
```

```
0030  01 4b 82 46 00 00 01 01  08 0a 07 33 99 ea 02 52   .K.F.... ...3...R
0040  ef 14                                              ..
```

c. Connect with SO_LINGER set to true and 0 seconds delay (Wireshark/test2c.pcap)

Data capture shows that disconnect is also sent right away and 10 MB transfer is interrupted. Connection is immediately forcefully closed with RST packet (red), without waiting for ACK packet from the server.

```
0000  00 1a 73 f6 f3 6e 6c b7  f4 71 14 15 08 00 45 00   ..s..nl. .q....E.
0010  00 28 00 00 40 00 40 06  b8 b0 c0 a8 00 6a c0 a8   .(..@.@. .....j..
0020  00 65 de e7 22 6b 7f 0e  59 42 00 00 00 00 50 04   .e.."k.. YB....P.
0030  00 00 54 1d 00 00                                 ..T...
```

TEST 3: SO_OOBINLINE

Connect with SO_OOBINLINE set to true and send URG byte (Wireshark/test3.pcap)

Data capture shows tcp packet received with PSH/ACK/URG flag set in segment 4 (in red). Data byte sent was “1” (in blue).

Captured URG packet:

```
0000  00 1a 73 f6 f3 6e 6c b7  f4 71 14 15 08 00 45 00   ..s..nl. .q....E.
0010  00 35 d8 a1 40 00 40 06  e0 01 c0 a8 00 6a c0 a8   .5..@.@. .....j..
0020  00 65 93 fe 22 6b cf e4  68 99 5b 84 f4 3c 80 38   .e.."k.. h.[..<.8
0030  00 e5 8f ea 00 01 01 01  08 0a 02 5b bb c6 07 5f   ........ ...[..._
0040  5e 7a 01
```

TEST 4: SO_RCVBUF and SO_SNDBUF

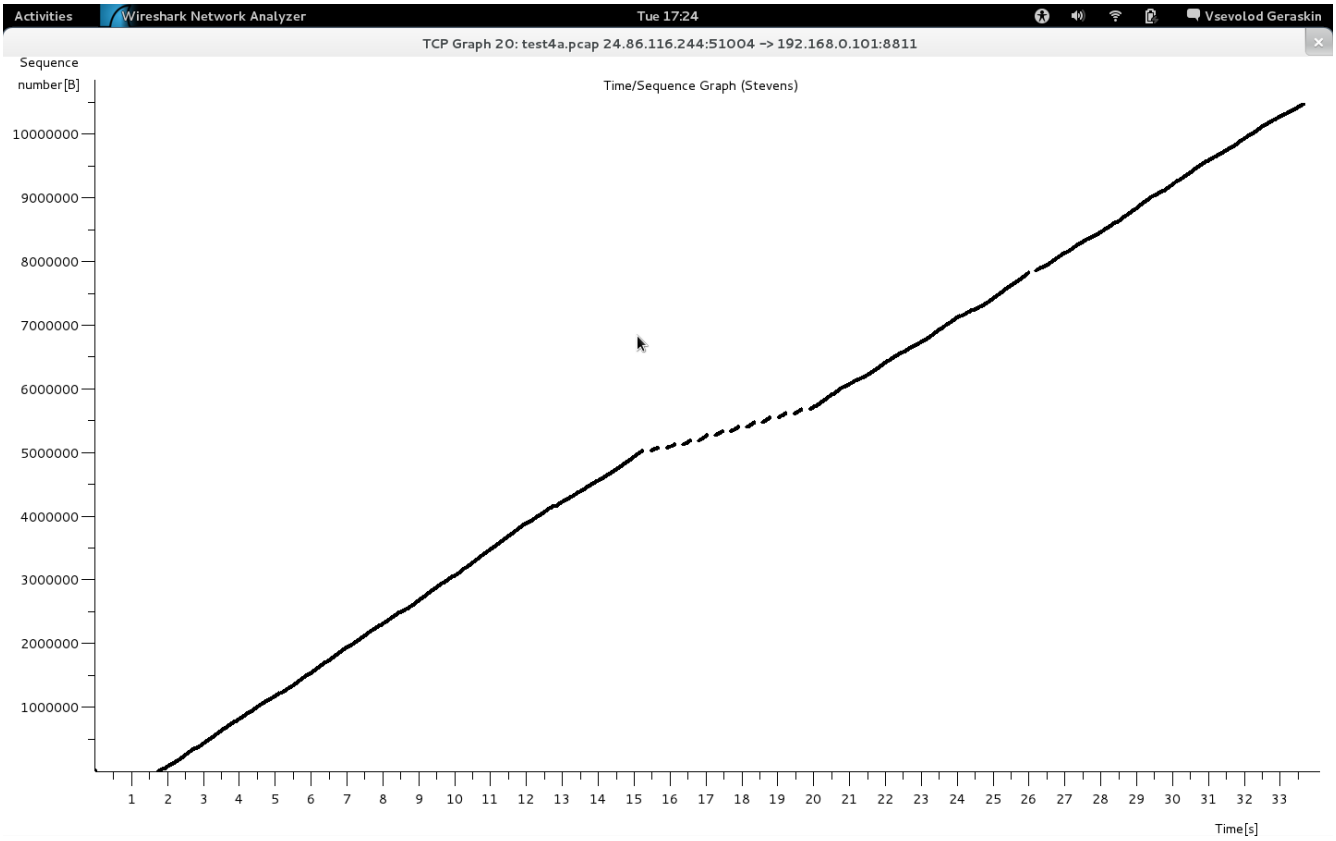
a. Send a 10mb file with buffers set to 1024 bytes (Wireshark/test4a.pcap)

In the Android TCP client log, transfer took 32 seconds. Since 1024 bytes buffer size was only a suggestion to the Android kernel, the system set buffer sizes to 2048 bytes. TCP window size was set at 14600 (red) in the initial SYN packet and the scale factor was set at 6 (blue) (x64). When the data transfer started, client TCP window was shown as 229, but, considering the scale factor, the actual size was 229 x 64 = 14656 bytes.

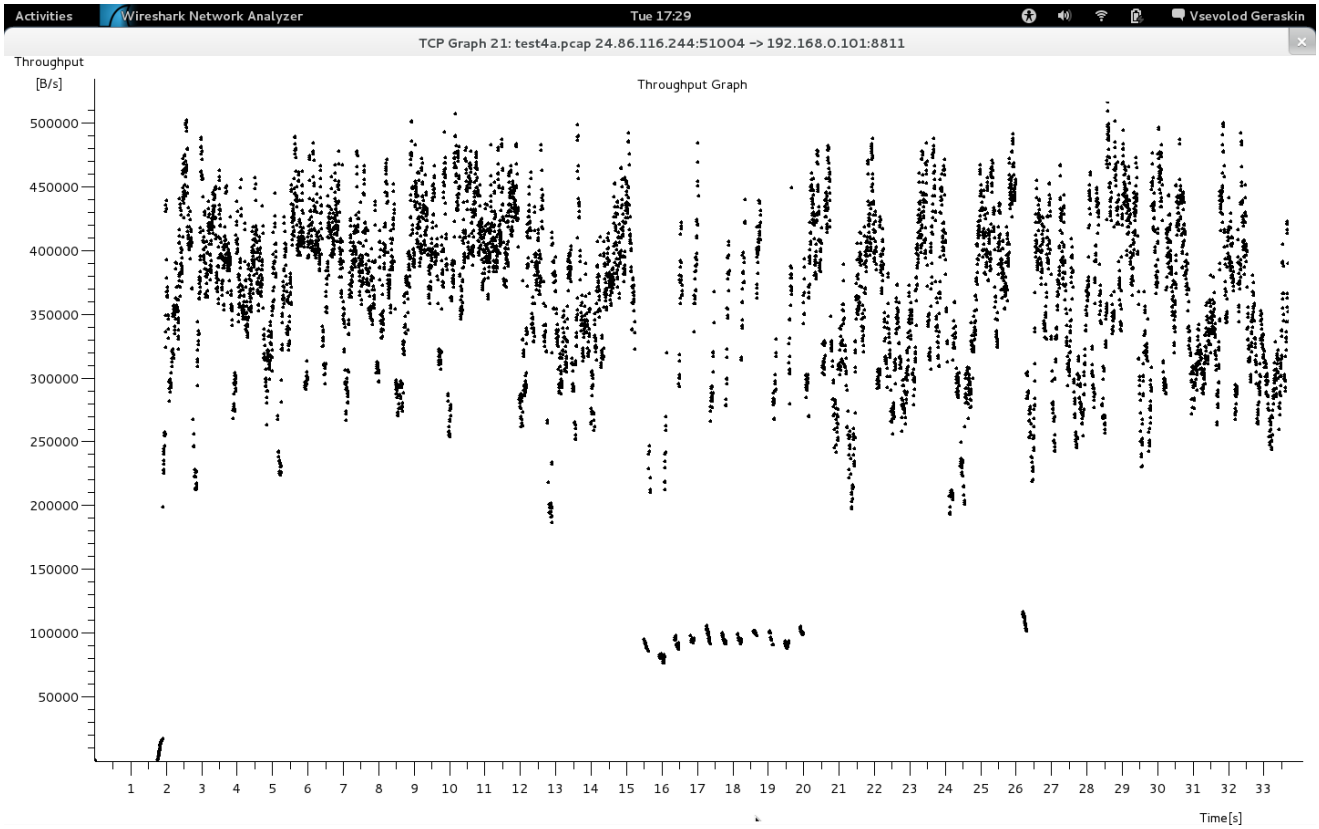
SYN Packet:

```
0000  00 1a 73 f6 f3 6e 00 26  5a ca fa 17 08 00 45 00   ..s..n.& Z.....E.
0010  00 3c db 92 40 00 3f 06  11 d2 18 56 74 f4 c0 a8   .<..@.?. ...Vt...
0020  00 65 c7 3c 22 6b e7 f5  83 e5 00 00 00 00 a0 02   .e.<"k.. ........
0030  39 08 10 44 00 00 02 04  05 b4 04 02 08 0a 02 63   9..D.... .......c
0040  58 77 00 00 00 00 01 03  03 06                     Xw...... ..
```

The following data capture graphs demonstrate TCP data flow over time and throughput of 10mb file transfer. The loss of throughput is evident on the next two graphs between 15 and 20 second marks. Abnormally, there is almost a .25 second delay between transmitted packet 5220 and acknowledgement packet 5221, and some other packets afterwards. This could be attributed to 802.11 CSMA/CA frame collision avoidance algorithm sensing the channel busy and delaying sending TCP packets until deciding the wireless channel is free. Unfortunately, it is difficult to tell if this indeed is observed with certainty because Wireshark is unable to capture wireless frames on most wireless adapters, and the capture is done on the server side.



Test 4a Graph 1 -Time/Sequence



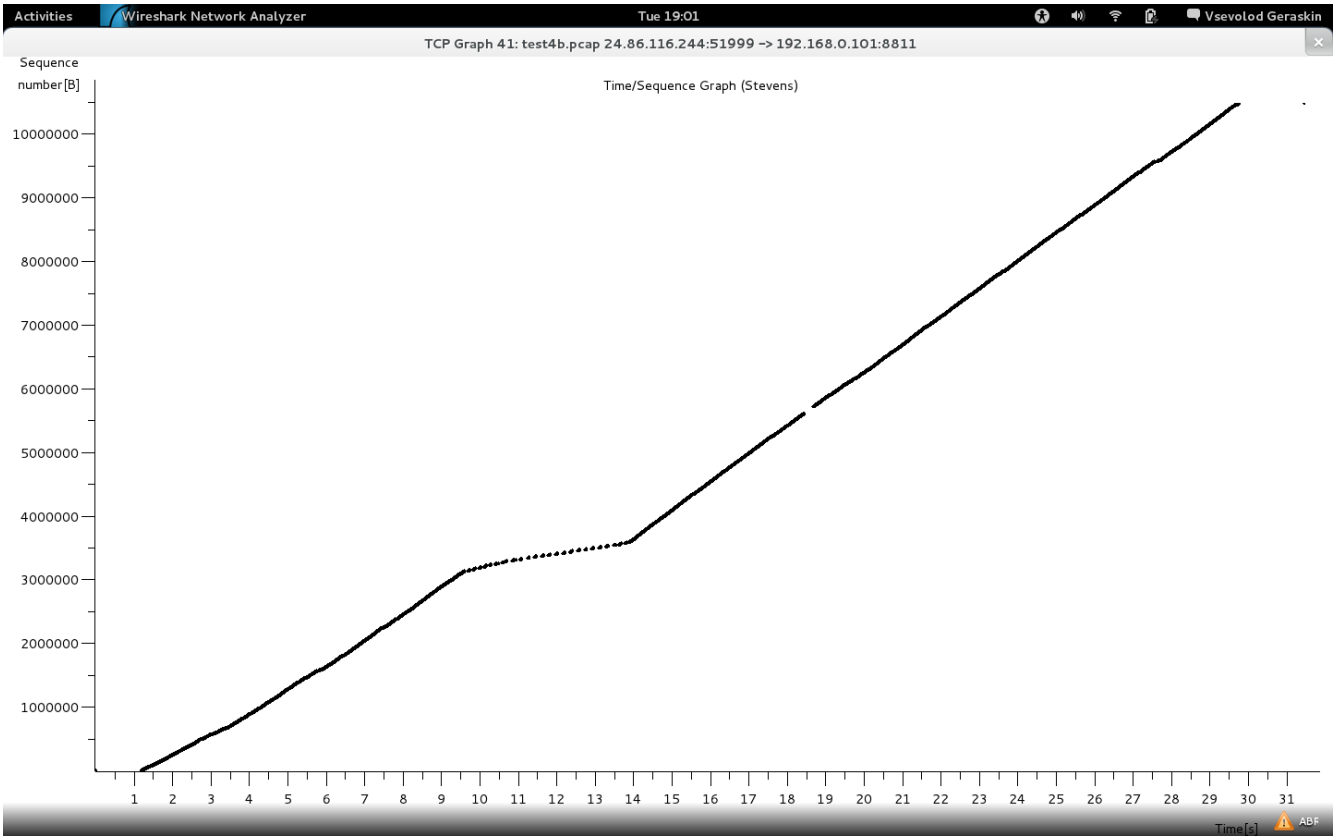
Test 4a Graph 2 – Throughput

b. Send a 10mb file with buffers set to 4096 bytes (Wireshark/test4b.pcap)

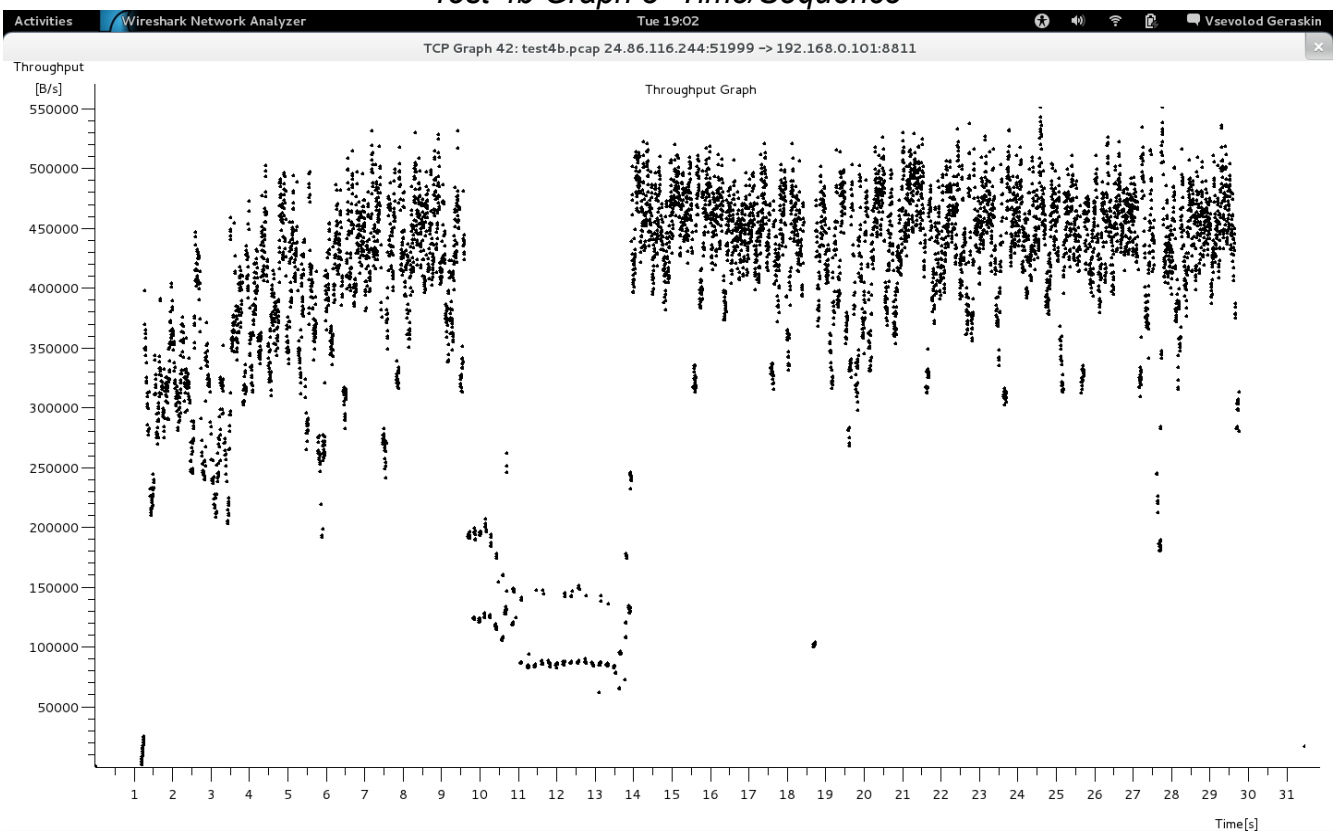
In the Android TCP client log, transfer took 29 seconds. Since 4096 bytes buffer size was only a suggestion to the Android kernel, the system doubled buffer sizes to 8192 bytes to account for bookkeeping overhead. Again, Android TCP window size was set at 14600 in the initial SYN packet and the scale factor was set at 6. When the data transfer started, client TCP window was again 14656 bytes. On the Linux server, using the same socket options resulted in TCP window being set at 2896 (red) bytes in SYN/ACK packet:

```
0000  00 26 5a ca fa 17 00 1a  73 f6 f3 6e 08 00 45 00   .&Z..... s..n..E.
0010  00 3c 00 00 40 00 40 06  ec 64 c0 a8 00 65 18 56   .<..@.@. .d...e.V
0020  74 f4 22 6b cb 1f 14 b4  90 e5 bd f2 34 b5 a0 12   t."k.... ....4...
0030  0b 50 4e 86 00 00 02 04  05 b4 04 02 08 0a 07 f4   .PN..... ........
0040  f0 d8 02 79 9a 25 01 03  03 00                     ...y.%.. ..
```

The following data capture graphs demonstrate TCP data flow over time and throughput of 10mb file transfer. The graphs look almost identical to the ones on previous test. Again we encountered the loss of throughput between 9 and 14 second mark possibly due to CSMA/CA frame collision avoidance algorithm.



Test 4b Graph 3 -Time/Sequence



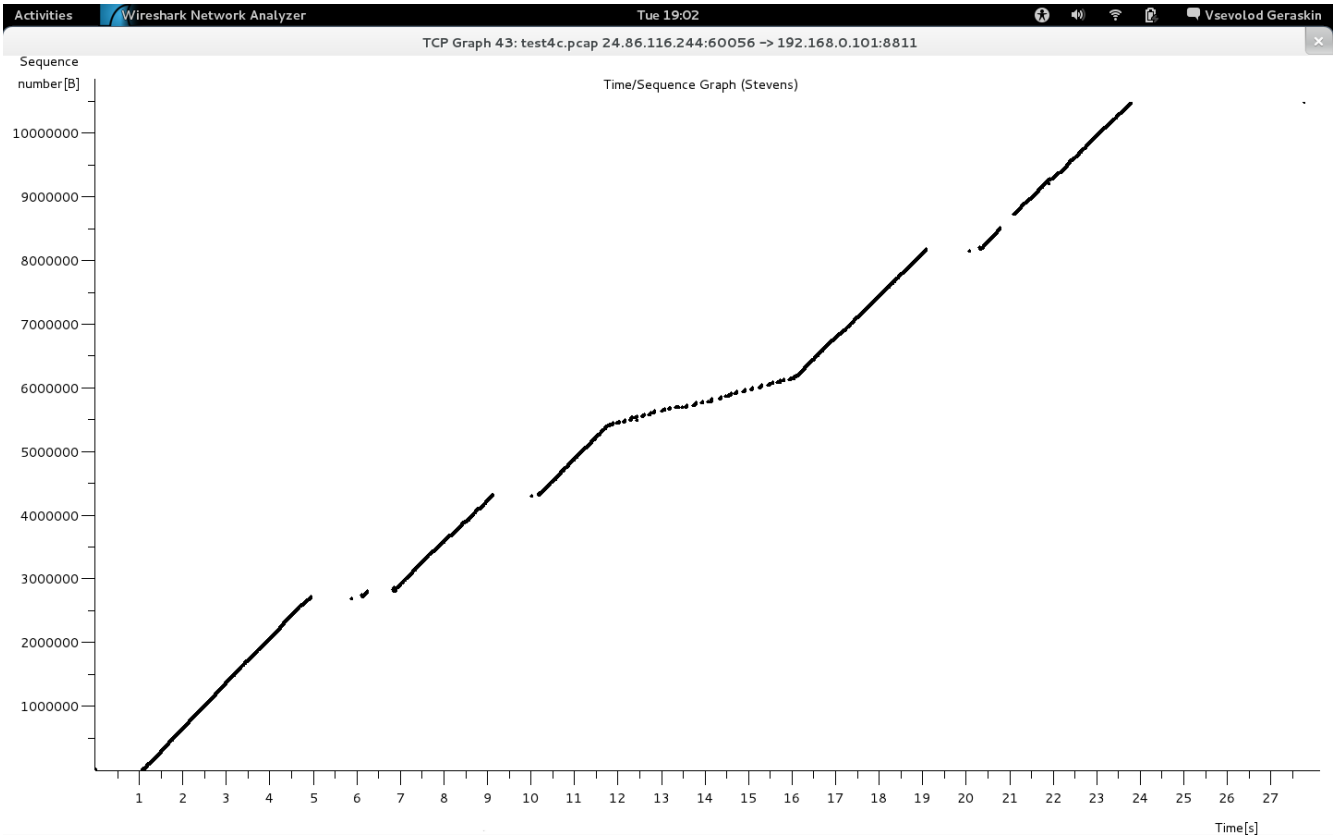
Test 4b Graph 4 -Throughput

c. Send a 10mb file with buffers set to 65536 bytes (Wireshark/test4c.pcap)

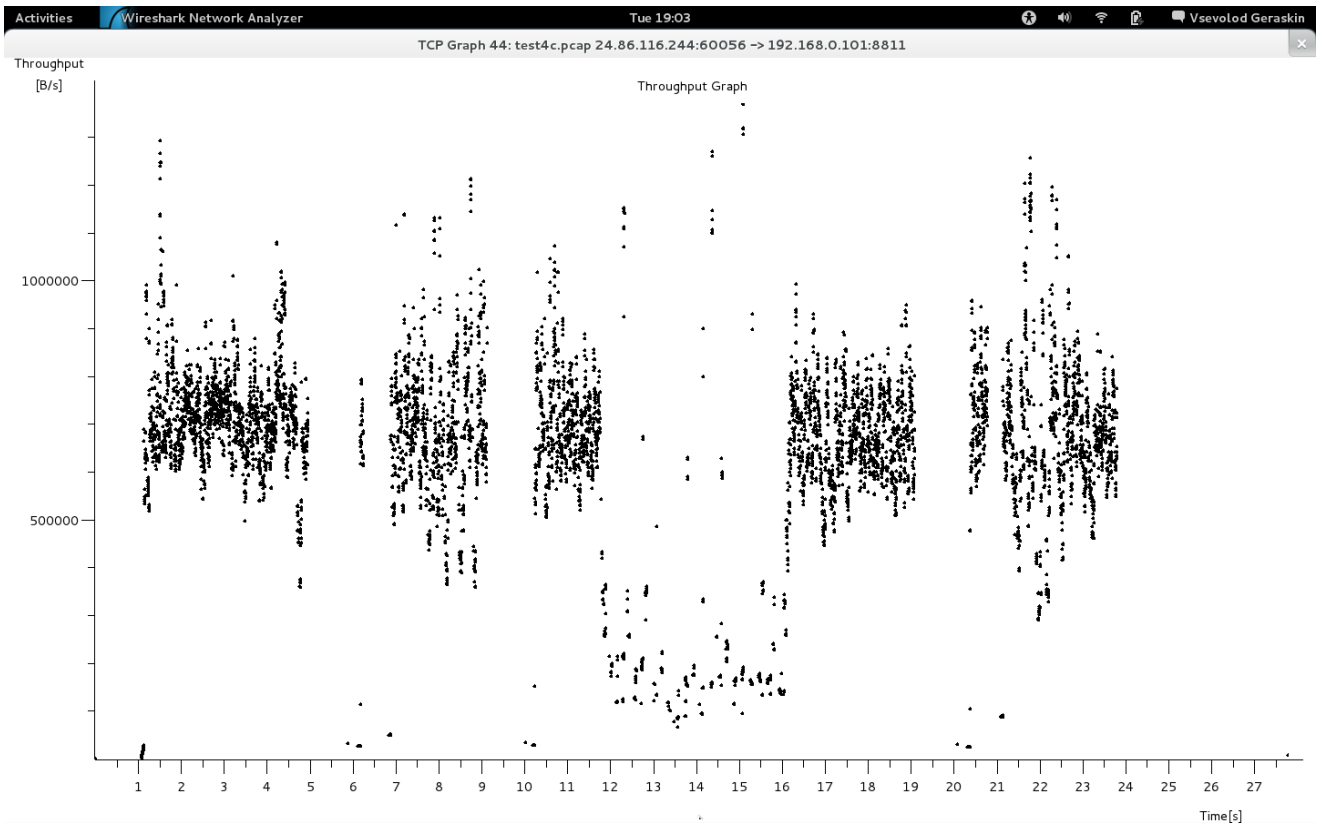
In the Android TCP client log, transfer took 22 seconds. Since 65536 bytes buffer size was only a suggestion to the Android kernel, the system doubled buffer sizes to 131072 bytes to account for bookkeeping overhead. Again, Android TCP window size was set at 14600 and the scale factor was 6. On the Linux server, using the same socket options resulted in TCP window being set at 14480 and the scale factor at 1 (x2). The server TCP window grew until it reached 64088 bytes (32044x2) (red) in packet 39:

```
0000  00 26 5a ca fa 17 00 1a  73 f6 f3 6e 08 00 45 00   .&Z..... s..n..E.
0010  00 34 93 6f 40 00 40 06  58 fd c0 a8 00 65 18 56   .4.o@.@. X....e.V
0020  74 f4 22 6b ea 98 3b b5  2c 87 f4 e9 b2 28 80 10   t."k..;. ,....(..
0030  7d 2c 4e 7e 00 00 01 01  08 0a 07 eb c4 35 02 77   },N~.... .....5.w
0040  c4 6a                                             .j
```

The following data capture graphs demonstrate TCP data flow over time and throughput of 10mb file transfer. Both graphs show a complete loss of throughput at 5, 9, and 19 second marks. Wireshark capture shows three duplicate ACKs in TCP packets 2836-2840, resulting in fast retransmission in packet 2841. This demonstrates TCP congestion mechanism in action, where TCP receives a triple-duplicate ACK event, and initiates fast retransmission. The TCP protocol also reduces congestion window by half after this event. However, we cannot observe this directly because the cwnd value is not stored in TCP/IP packets, but rather a local value on sender side.



Test 4c Graph 5 -Time/Sequence



Test 4c Graph 6 - Throughput

TEST 5: SO_TIMEOUT

a. Connect with SO_TIMEOUT set to 0. Attempt to read the socket for more than 5 seconds, then disconnect. (Wireshark/test5a.pcap)

Data capture contains a standard 3-step TCP handshake, sent string, and TCP disconnect.

b. Connect with SO_TIMEOUT set to 5000. Attempt to read the socket for more than 5 seconds, then disconnect. (Wireshark/test5b.pcap)

Data capture contains a standard 3-step TCP handshake, sent string, and TCP disconnect. In addition, Java throws SocketTimeoutException error after 5 seconds of blocking on read.

Eclipse debugger output:
11-26 19:47:35.525: D/sev client(20477): sending a string...
11-26 19:47:40.525: W/System.err(20477): java.net.SocketTimeoutException

TEST 6: TCP_NODELAY

Connect with TCP_NODELAY set to true (Wireshark/test6.pcap)

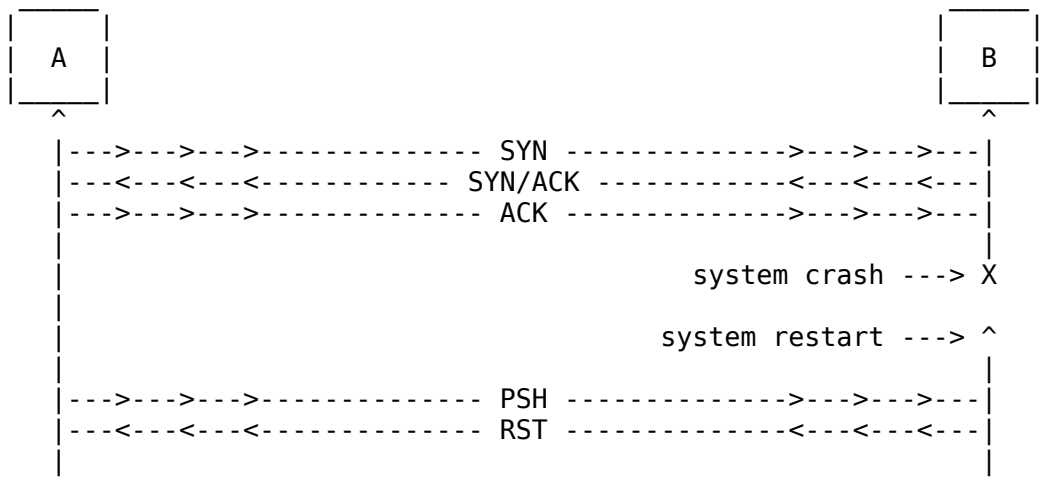
Furthermore to suggested test, the 10mb file was sent with 65536 buffer size and TCP_NODELAY set to true. At first, the transfer with TCP_NODELAY flag took longer than

test 4c, 32 seconds, but consecutive transfers showed little difference between TCP_NODELAY flag set to true or false.

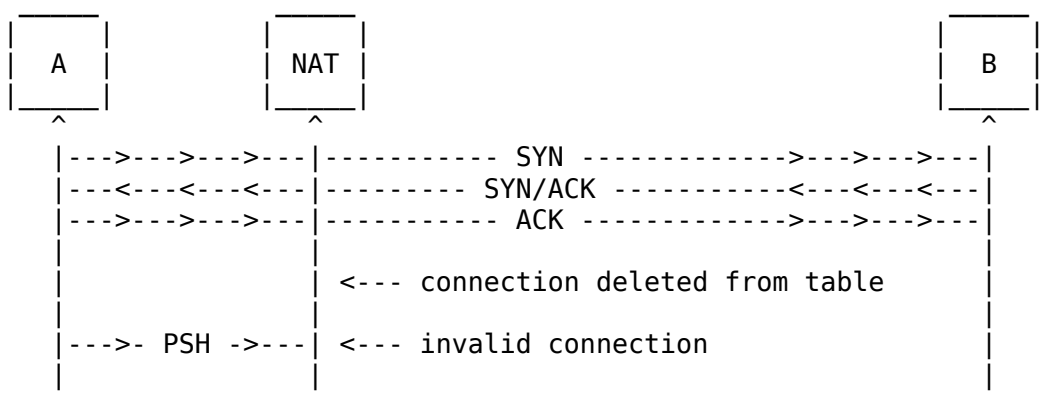
CONCLUSION AND FURTHER DISCUSSION

SO_KEEPALIVE

This option can be used to verify the TCP connection, prevent disconnects from network inactivity, and check for dead peers. The following diagrams show the logic of using the option.



Graph 7: check for dead peers
(source http://tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO)



Graph 8: prevent disconnects from network inactivity
(source http://tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO)

Unlike Linux, Android current implementation does not allow us to set KEEP-ALIVE intervals. The default and only intervals for KEEP-ALIVE packet is set at 2 hours. Thus, in most application, this option is useless, and developers are advised to implement their own application-level “heartbeat”.

SO_LINGER

This option can be used to forcefully terminate TCP connection after a specified timeout in seconds. The behaviour of TCP with this option OFF is attempt to send any queued data with an infinite wait time before socket is closed. This could lead to performance issues where a busy server could wait for a while to close a connection. Turning SO_LINGER option on allows to control this behaviour by terminating connection with RST packet after timeout expires without waiting for acknowledgement from a peer. This option is probably best used when optimizing very busy servers with many simultaneous connections.

SO_OOBINLINE

This option can be used to send out-of-band data that bypasses normal TCP connection queue. This packets are sent with URG flag set, and require implementing a separate process in application in order to send and receive them. Urgent bytes can be sent by a client to a server to cancel an outstanding (unprocessed) request in order to save server resources.

SO_RCVBUF and SO_SNDBUF

Optimal buffer for uploading large files

Setting the larger buffers allowed 10mb file transfer to complete significantly faster on the Android Tablet (32 seconds at 1024 bytes, 22 seconds at 65546 bytes). Further captures using Android emulator instead (Wireshark/test4f.pcap and Wireshark/test4g.pcap) confirmed the improved performance (116 seconds at 1024 bytes, 23 seconds at 65546 bytes). The important note is that the socket buffers were set to the same sizes both on the server and the client.

When using different buffer sizes at the server and the client, performance suffers. With 1024-sized buffers on the Linux server and 65536-byte buffers on the Android client, 10mb transfer took over 70 seconds (Wireshark/test4d.pcap). On the other hand, with 65536-sized buffers on the Linux server and 1024-byte buffers on the Android client (Wireshark/test4e.pcap), 10mb transfer took 31 seconds.

In summary, the optimal buffer size for transferring 10mb file over the specified 802.11 wireless network is the largest possible: 65536 bytes. This seems logical because, when dealing with large transfers, we want to improve throughput and reduce occurrence of flow control. As seen in the tests, the Linux server's TCP window is affected by SO_RCVBUF and SO_SNDBUF values. If the TCP window size is too small, then the buffer could be overrun, and the flow control will stop the data transfer until the window is cleared.

TCP window size in Android

While these socket options can be used on Linux to affect TCP window size, the tests showed that on both Android tablet and Android emulator clients the window size remained constant at around 14600 bytes. This is probably due to Android sending the file and receiving only acknowledgements from the server. Further tests were required to see if SO_SNDBUF and SO_RCVBUF socket options were indeed affecting TCP window in Android. When data flow

was reversed and Android tablet downloaded 10mb of data from the server (Wireshark/test4h.pcap) using 65536-byte buffers, Android grew TCP window and advertised the window as 98304 bytes (1536x64 with the scale factor of 6) in packet 92 (red):

```
0000  00 1a 73 f6 f3 6e 00 26  5a ca fa 17 08 00 45 00   ..s..n.& Z.....E.
0010  00 34 e0 3a 40 00 3f 06  0d 32 18 56 74 f4 c0 a8   .4.:@.?. .2.Vt...
0020  00 65 ca 07 22 6b 59 b5  49 78 76 6c 9c 40 80 10   .e.."kY. Ixvl.@..
0030  06 00 ed 5f 00 00 01 01  08 0a 03 9c ab 4c 0a 33   ..._.... .....L.3
0040  d9 9c
```

Moreover, when Android tablet downloaded 10mb of data from the server (Wireshark/test4i.pcap) using 1024-byte buffers, Android reduced TCP window and advertised the window as 1728 bytes (27x64 with the scale factor of 6) in packet 31 (red):

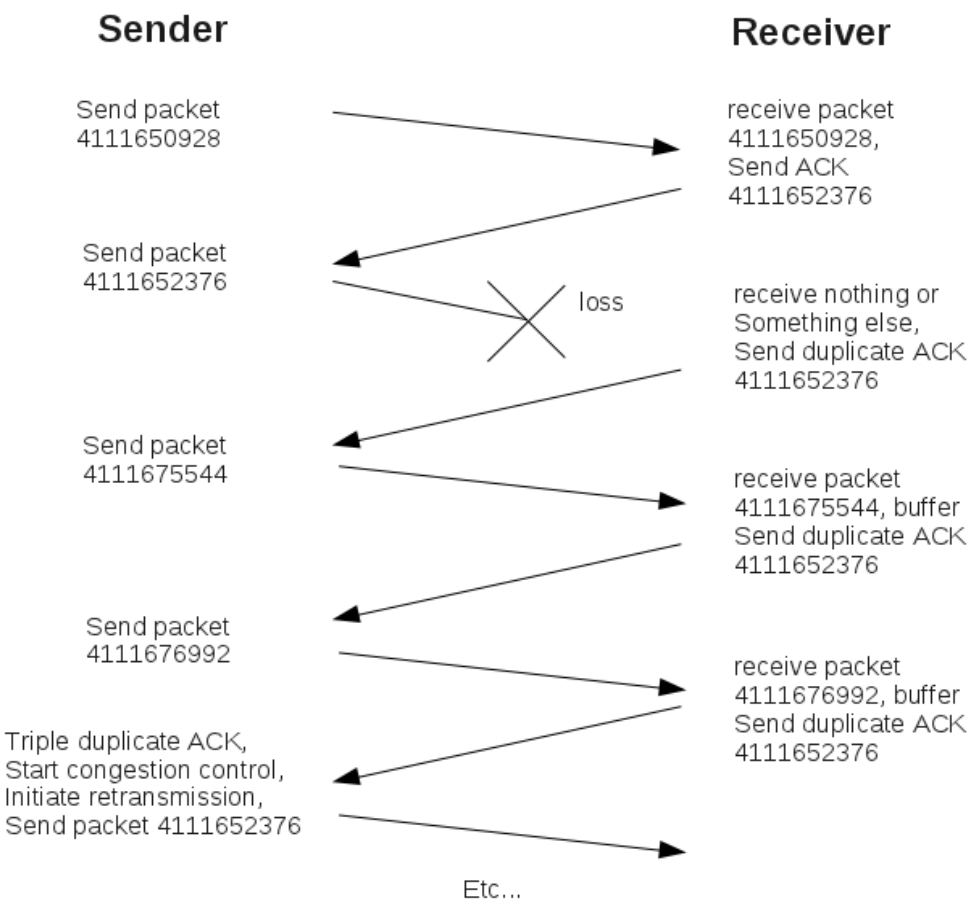
```
0000  00 1a 73 f6 f3 6e 00 26  5a ca fa 17 08 00 45 00   ..s..n.& Z.....E.
0010  00 34 bb aa 40 00 3f 06  31 c2 18 56 74 f4 c0 a8   .4..@.?. 1..Vt...
0020  00 65 9d 34 22 6b d5 23  0a 12 8a 53 19 4e 80 10   .e.4"k.# ...S.N..
0030  00 1b 1d e5 00 00 01 01  08 0a 03 a1 b4 2b 0a 4d   ........ .....+.M
0040  05 d5
```

Now, the report can safely conclude that TCP socket options do indeed affect TCP window size in Android kernel. Furthermore, the data agreed with previous conclusion about optimal buffer size. With 65536-byte buffer, file download took 15 seconds rather than more than 60 seconds with 1024-byte buffer.

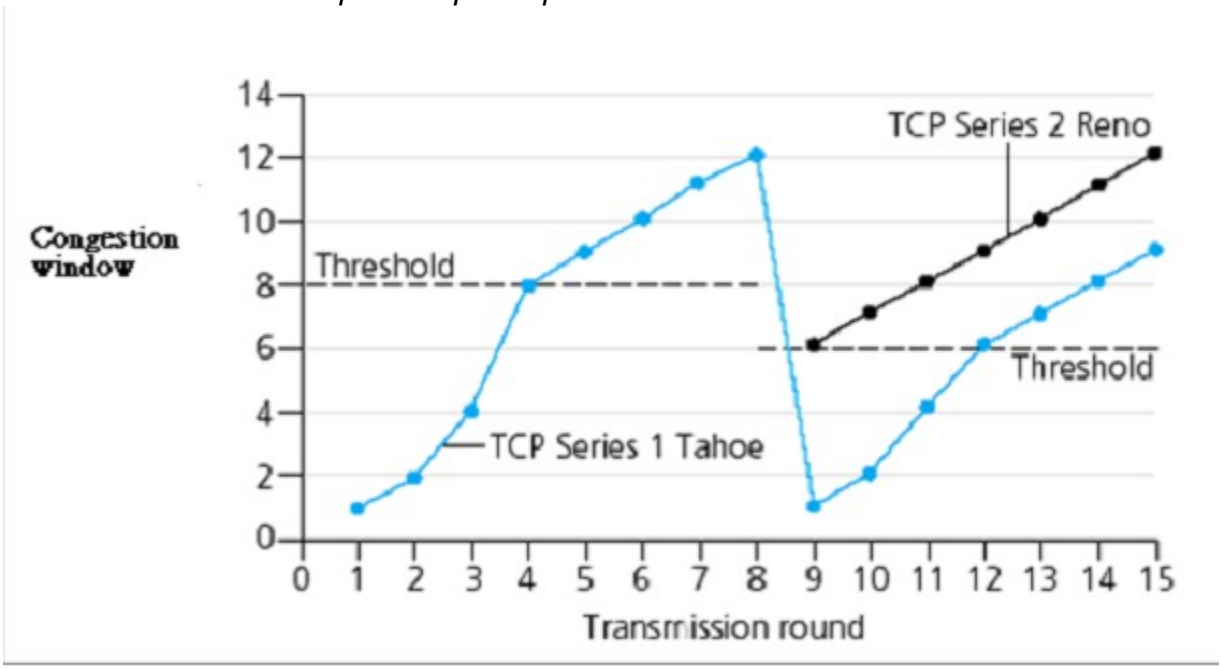
CSMA/CA and TCP Congestion Control in 802.11 Wireless Networks

Carrier sense multiple access with congestion avoidance algorithm in 802.11 networks senses the wireless channel, and transmits TCP packets only when the channel is idle. If the algorithm senses channel busy, it picks the random back-off value and starts countdown when the channel is idle again. When the countdown reaches 0, the station transmits wireless frame and waits for acknowledgement. This algorithm could be the reason for the abnormal delay between packets seen in 10mb file transfer tests.

Furthermore, as observed in test 4c, sender could get ahead of the receiver and send packets with sequence number greater than what the receiver expects, resulting in events such as server sending three duplicate ACKs (graph below). In a case of such event, TCP congestion control reduces congestion window by half as shown on the graph below after transmission round 8, and throughput is reduced. The loss of TCP packets could be caused by them never reaching server due to wireless frame collisions. In such cases, consecutive frame re-transmissions are required. In common situations, when streaming media over 802.11 network with much channel noise, both collision avoidance delays and frame collisions could cause the video to appear choppy and frozen at times.



Graph 9: Triple-duplicate ACK event in test 4c



Graph 10: evolution of TCP congestion window
(source: Computer Networking, a Top-Down Approach, Kurose and Ross)

SO_TIMEOUT

This is not a TCP socket option, but rather java.net.Socket option that throws SocketTimeoutException after specified millisecond timeout. This option could be used to timeout and close an idle connection.

TCP_NODELAY

This option is used to disable Nagle's algorithm. The algorithm (generally) improves the efficiency of TCP/IP networks by buffering the packets which received no acknowledgement from peer and, thus, delaying the send. However, multiple file uploads with this option turned on and off yielded no meaningful results. In theory, this option should be used in very specific cases, such as when encountering performance issues using Nagle's algorithm in combination with TCP delayed acknowledgement algorithm, or in environment where many small packets are passed between peers.