

Tuning your MSS

Noah Davids
Support Engineer
Stratus Technologies

The Maximum Segment Size (MSS) is the maximum number of bytes that TCP will place in a TCP segment (AKA an IP packet with TCP data). By default STCP will use an MSS of 536 for all connections which are not local, that is the host at the other end of the connection (the remote peer) is not on the same subnet as the local end of the connection. It does this even if the remote peer advertises an MSS larger than 536 bytes (see below for a discussion on MSS advertisement). The TCP RFC only requires that no more than the advertised MSS number of bytes be sent in a segment.

Just what is the effect of using 536 bytes instead of the maximum (for Ethernet) of 1460 bytes?

Since every TCP segment includes at least 58 bytes of overhead (20 bytes for the TCP header, 20 bytes for the IP header and 18 bytes from the Ethernet header and trailer) the more data that can be packed into a TCP segment the smaller the percentage of overhead and the more efficient the transfer of data. How efficient? Table 1 compares two FTP transfers between a PC workstation running Microsoft® Windows® XP and a system running VOS 14.6 with STCP. The PC client is doing an ASCII get of a 1 MEG stream file. The packet and byte counts and time include the acknowledgment packets that the PC must send back to the VOS system. As you can see, increasing the MSS can make a significant difference in a file transfer.

	MSS = 536	MSS = 1460	Improvement
IP Packets sent	3,050	1,204	60.52% fewer packets
IP Bytes sent	1,236,066	1,121,312	9.28% fewer bytes
Time	2.711 seconds	1.966 seconds	27.48% faster

Table 1 – FTP get comparison

Why does STCP do this?

It all goes back to the early days of TCP when hosts and routers had memory restrictions.

RFC 791 (September 1981) – Internet Protocol states that

“All hosts must be prepared to accept datagrams of up to 576 octets (whether they arrive whole or in fragments). It is recommended that hosts only send datagrams larger than 576 octets if they have assurance that the destination is prepared to accept the larger datagrams.”

RFC 879 (November 1983) – TCP maximum segment size and related topics states it more strongly

“HOSTS MUST NOT SEND DATAGRAMS LARGER THAN 576 OCTETS UNLESS THEY HAVE SPECIFIC KNOWLEDGE THAT THE DESTINATION HOST IS PREPARED TO ACCEPT LARGER DATAGRAMS.”

The RFCs are talking about IP packets which include both the 20 byte IP header and the 20 byte TCP header, while the MSS value deals with the number of data bytes carried in the TCP segment. Subtracting the 40 bytes of IP and TCP header from 576 gives the 536 MSS value that STCP uses.

The above excerpts from the RFCs do not explicitly state that if a host has specific knowledge that the destination host (remote peer) can accept larger datagrams it should send them. This is because the destination host has no knowledge of what the network between the two can support. So STCP sends only 536 bytes in any TCP segment going to a non-local remote peer.

How does a host know what the remote peer’s MSS is?

The MSS is advertised in the packets that set up the connection, the packets that contain the SYN flag. Each side sends its own MSS value. There is no requirement that both sides use the same value, but each side cannot send more data than is allowed by the remote peer’s advertisement. The MSS value is sent in the “Maximum Segment Size (MSS)” TCP header option. Most protocol analyzers will interpret the header, identify the option, and display its value. Packet_monitor, however, does not interpret TCP options; but if you know what to look for the MSS option is easy to identify. Figure 1 shows a packet_monitor trace of the first two packets of a connection. Packet 1 is from the remote peer and contains the **Syn** flag. Packet 2 is STCP’s acknowledgement and contains the **Syn** and **Ack** flags. TCP options have the format <option number> <option len> <option value>. The MSS option number is **2**. It will always have a length of **4** and the next two bytes are the MSS value – in hex of course. In packet 1, that is **5b4**, which is 1460 in decimal. In packet 2, it is **218**, which is 536. Packet 1 also includes **some other options**. STCP just ignores these.

```

>system>stcp>command_library>packet_monitor -numeric -time_stamp -verbose -hex_d
+ump -filter -host 164.152.79.200
dir
icmp type
+ tcp
hh:mm:ss.ttt dir len proto source destination src port ds
+t port type
19:20:16.784 Rcvd IP Ver/HL 45, ToS 0, Len 34, ID 1449, Flg/Frg 4000, TTL 3
+e, Prtl 6
Cksum 41b7, Src a4984fc8, Dst a4984dcb
TCP from 164.152.79.200.2315 to 164.152.77.203.telnet
seq 800916ce, ack n.a., window ebc0, 12. data bytes, flags Syn.
X/Off 08, Flags 02, Cksum fcf1, Urg-> 0000
offset 0 . . . 4 . . . 8 . . . C . . . 0...4... 8...C...
0 2 4 5 b4 1 3 3 2 1 1 4 2 <<<Z<<<< <<<<
19:20:16.809 Xmit IP Ver/HL 45, ToS 0, Len 2c, ID 68db, Flg/Frg 0, TTL 3
+c, Prtl 6
Cksum 2f2d, Src a4984dcb, Dst a4984fc8
TCP from 164.152.77.203.telnet to 164.152.79.200.2315
seq 557e6074, ack 800916cf, window 2000, 4. data bytes, flags Syn Ack.
X/Off 06, Flags 12, Cksum 3f01, Urg-> 0000
offset 0 . . . 4 . . . 8 . . . C . . . 0...4... 8...C...
0 2 4 2 18 <<<<

```

Figure 1 – packet_monitor trace showing the MSS option

How can you adjust STCP’s default MSS value?

STCP includes a variable named default_min_mtu. The value of this variable defaults to 556 (it includes the 20 byte TCP header). Setting this value to 1480 will allow the largest possible TCP segment on an Ethernet (1460 bytes) to be created with 1 exception. The exception happens if the remote peer advertises an MSS value less than the 1460. In that case the advertised value is used.

In releases earlier than VOS 14.7, you can display the value of default_min_mtu with the display_analyze_system request and set the value with the set_longword request.

```

as: display default_min_mtu
FEAC38F8 0 0000022C |..., |

as: set_longword default_min_mtu 5C8
addr from to
FEAC38F8 0000022C 000005C8

```

Figure 2 – Displaying and changing the default MSS value prior to 14.7

Starting in release 14.7, there is an analyze_system request that you can execute to display and set the value in decimal.

```

as: list_stcp_params min_mss
maximum tcp segment size [500-1480] (min_mss) 556

```

```
as: set_stcp_param min_mss 1480

Changing maximum tcp segment size (min_mss)
from 556 to 1480
```

Figure 3 – Displaying and changing the default MSS value in 14.7 and later

Changing the value this way must be done after each boot of the system and after STCP has been started. Adding the following command to the end of the start_stcp.cm command macro would take care of it for you.

```
analyze_system -request_line 'set_longword default_min_mtu 5c8' -quit
```

Or in VOS 14.7 and later,

```
analyze_system -request_line 'set_stcp_param min_mss 1480' -quit
```

You can also use the get_external_variable and set_external_variable commands to display and set the value. The advantage of using set_external_variable command is that it only has to be done once per VOS release instead of in every boot since it changes the executable file.

```
get_external_variable default_min_mtu -in ip.cp.pm -type integer
556
ready 15:19:23

set_external_variable default_min_mtu -in ip.cp.pm -type integer -to 1480
ready 16:00:12
```

Figure 4 – Using get_external_variable and set_external_variable

The disadvantage is that it only has to be done once per VOS release. The problem here is that people tend to forget it was done and notice a change after a new version of VOS is installed. A lot of time is then spent trying to figure out why. In general I prefer putting the change in start_stcp.cm so it is always done and it is obvious that it is being done. However, some sites don't like to run analyze_system, so this is an alternative. If you use it, I suggest that you place a comment in start_stcp.cm with perhaps a display_line message to get it into the module_start_up.out file. Also, change the version information on the file.

```
>system>maint_library>get_pm_version ip.cp.pm
ip.cp.pm 'Release 14.6.0ak'
ready 16:04:21

>system>maint_library>set_pm_version ip.cp.pm 'rel 14.6.0ak, MSS to 1460 NSD'
ready 16:05:07

>system>maint_library>get_pm_version ip.cp.pm
ip.cp.pm 'rel 14.6.0ak, MSS to 1460 NSD'
ready 16:05:22
```

Figure 5 – Using get_pm_version and set_pm_version to flag a changed PM file

Another disadvantage of this approach is that it has to be done before STCP is started. Once STCP is started and the IP module is loaded the file cannot be changed.

What is so bad about sending a frame that is too large?

Several things. First, if an intermediate router is required to fragment the packet so that it can be transmitted over the next hop, the router will take a lot longer to forward the frame. Second, fragmentation requires more resources from the router so a router running at the edge of its performance capabilities may experience congestion problems and start dropping packets (not necessarily the packet being fragmented). Third, a router fragments each packet individually so a 1460 byte packet is fragmented into two 536 byte fragment packets and one 388 byte fragment packet. A sequence of four 1460 byte packets fragmented into 536 byte frames requires twelve fragment packets but the sending TCP stack can pack the data into eleven packets, ten 536 byte packets and one 480 byte packet. The extra packet not only causes data transfer efficiency to go down, but also increases the chances that part of the packet will be lost. When that happens the entire 1460 bytes must be retransmitted. So, all in all, it is not a good thing if an intermediate router is required to fragment a packet.

How can you tell if fragmentation is happening?

One way is to watch the packets with an analyzer. However, packet_monitor will NOT show fragmented packets. By the time incoming packets get to packet_monitor they have been reassembled.

Another way is with the IP statistics displayed by netstat. Figure 6a shows a portion of the netstat statistics displayed by STCP. Note that the ipReasmReqds and ipReasmOKs have both gone up. ipReasmReqds is a count of all the fragmented packets and ipReasmOKs is a count of the packets once they are reassembled. If there were any errors that ipReasmFails would have gone up.

```
netstat -statistics -protocol ip
ip:
. . .
          92  ipReasmReqds
          46  ipReasmOKs
           0  ipReasmFails
. . .

netstat -statistics -protocol ip
ip:
. . .
          96  ipReasmReqds
          48  ipReasmOKs
           0  ipReasmFails
. . .
```

Figure 6a – STCP netstat statistics showing evidence of incoming fragmented packets

Every TCP stack is a little different. Figures 6b thru 6g show the relevant netstat output for the other operating systems that Stratus supports. You will note that like STCP, TCP_OS (figure 6b), Windows 2000 and 2003 (figure 6c) and Stratus ft Linux[®] operating systems (figure 6g) count both individual fragmented packets and the reassembled packets. FTX[®] 2.4 and 3.3 (figure 6d) count only the received fragments while FTX 3.4 (figure 6e) and HP-UX[®] 11 (figure 6f) count only the reassembled packets.

```
netstat -s
. . .
ip:
. . .
          20  fragments received
          10  datagrams reassembled
. . .
```

Figure 6b – TCP_OS netstat statistics showing evidence of incoming fragmented packets

```
C:\Documents and Settings\Administrator>netstat -s
```

```
IP Statistics
```

```
. . .  
Reassembly Required           = 5668  
Reassembly Successful         = 2834  
Reassembly Failures           = 0  
. . .
```

Figure 6c – Windows 2000/2003 netstat statistics showing evidence of incoming fragmented packets

```
# netstat -s | grep fragments  
32 fragments received  
0 fragments dropped (dup or out of space)  
0 fragments dropped after timeout  
#
```

Figure 6d – FTX 2.4 and 3.3 netstat statistics showing evidence of incoming fragmented packets

```
# netstat -s | grep fragment  
8 fragments received  
0 fragments dropped (dup or out of space)  
0 fragments dropped after timeout
```

Figure 6e – FTX 3.4 netstat statistics showing evidence of incoming fragmented packets

```
# netstat -s | grep fragment  
8 fragments received  
0 fragments dropped (dup or out of space)  
0 fragments dropped after timeout
```

Figure 6f – HP-UX 11.00 netstat statistics showing evidence of incoming fragmented packets

```
[root@localhost root]# netstat -s | grep reassemb  
15 reassemblies required  
5 packets reassembled ok
```

Figure 6g – ft Linux netstat statistics showing evidence of incoming fragmented packets

It is important to monitor all the hosts you are connected to. If you just monitor one or two, you cannot be sure that other hosts using different routes through the network are not getting fragmented packets. If there are multiple routes between two hosts then it is also possible that some routes will result in fragmented packets and some will not. Monitoring over time should be done. However, in general, today's networks will support 1460 byte TCP segments.

What about changing the default MSS value for the other Stratus operating systems?

Of all the other operating systems that Stratus supports, only TCP_OS and FTX 2.4 use a default value smaller than 1460 for non-local connections. The external variable

tcpos_default_mss\$ can be used to change the TCP_OS behavior. Like STCP's default_min_mtu you can change the value with set_longword (figure 7) after the TCP_OS driver has been loaded or use set_external_variable to change permanently when the driver is not loaded (figure 8). Prior to 14.7 and the fix for otp-868, the default value of tcpos_default_mss\$ is 512. The otp-868 bug was fixed in VOS 14.7 to set the value to 536. Note that with this variable, the 20 bytes of TCP header are not counted so you set the value to 1460, not 1480.

```
as: set_longword tcpos_default_mss$ 5b4
addr      from      to
FEBC09A4  00000200  000005B4
```

Figure 7 - Changing the default MSS value for TCP_OS

```
get_external_variable tcpos_default_mss$ -in tcp.pm -type integer
512
ready 20:47:53

set_external_variable tcpos_default_mss$ -in tcp.pm -type integer -to
1460
ready 20:48:44
```

Figure 8 - Using get_external_variable and set_external_variable on tcp.pm for TCP_OS

HP-UX, Windows 2000/20003, and ft Linux operating systems all start off with and advertise an MSS of 1460, and do something called PMTU discovery, which basically determines the optimum MSS value for each connection so there is no need for you to manually change anything.

What is PMTU discovery?

PMTU (path maximum transmission unit) discovery is the process where a sending host determines the largest packet that can be sent to a remote host. The MTU and MSS are linked via a simple formula. MTU – Ethernet header bytes – IP header bytes – TCP header bytes – Ethernet trailer bytes = MSS. When a host is doing PMTU discovery it sets the “Do Not Fragment” bit in the IP header and sends its data. If it gets back an ICMP “fragmentation needed but don't fragment bit set” error, it knows to reduce the size of the MTU for that path, which in turn will reduce the size of the MSS. The ICMP error message should include the MTU size that should be set. Complete details can be found in RFC 1191 (see <http://ftp.rfc-editor.org/in-notes/rfc1191.txt>).

Microsoft and Windows are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries. HP-UX is a registered trademark of Hewlett-Packard Company. FTX is a registered trademark of Ascend Communications, Inc. (except in Japan where it is a trademark). The registered trademark Linux® is used pursuant to a sublicense from the Linux Mark Institute, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

All other trademarks and registered trademarks are the property of their respective holders.

