==========
Use of shift count :
==========
To inform other end of TCP connection how much of RCV.BUF it currently has for receiving data or
how much it's application can consume at the current.

==========
Rational to increase shift count :
==========
To be able to get higher bandwidth

==========
Limiting factors for shf.cnt > 14 :
==========
Dependencies limiting the need for greater than 14 shf.cnt on system:
01. Memory: interface hardware will support link speed of 100 Gbps or more but they can be limited
            by RCV.BUF memory.

02. Interface Hardware: there may not be a necessity for having such large shift count as it is
            limited by link speed.

03. Software: it may also be limited by application which consumes segments in its receive buffer.

There may be systems where interface speed is less but more memory is available or there may be
systems where interface speed and memory are sufficient to support shft.cnt of 15 or more but are
limited by application to consume such huge data within practical limits.

==========
Legacy use of shf.cnt == 14 and exploring beyond :
==========
We know that 2^32 sequence counter (sequence counter is the the sequence field in TCP header) has
valid sequence numbers from 0 to 4294967295. To determine if a data segment is "old" or "new" it
sets the window size to floor((2^32-1)/2) + 1 since sender's and receiver's window can be out of
phase by at-most 1 window. Therefore sequence space is divided into 2 parts.
1st part has valid range of sequence numbers from = 0 to 2147483647 and since its contained inside
size of of 2^32 therefore we get another valid range of sequence numbers from 2147483648 to
4294967295. So if the sender has sent all its packets within the window from 0 to 2147483647 and
receiver has accepted all the packets and sent cumulative acks as well for all of these then it
advances RCV.NXT to start of next maximum available range of sequence numbers. Upon receiving all
cumulative acks the sender also advance to next set of available sequence numbers. Otherwise, if
considering the scenario when spurious retransmission occurs or another scenario where all
cumulative acks are lost and retransmission timeout or another scenario when cumulative ack
carrying widow update information is lost, then in such cases the sender will retransmit segment
with sequence number that will fail receiver's segment acceptability test.

If we have to have a window of size > 2^31 then we need sequence numbers that can represent that
much of data within a window distinguishly. As an example we have window size of 2^36 then we
need sequence number of size 2^37. So we have 2 windows in hand of size 2^36 as a fail safe just
in case TCP sender and receiver gets out of phase by at-most 1 window.

Q. Can PAWS help support window size > 2^31 without increasing sequence number size ?
A. PAWS is used for discarding old duplicates. The basic idea is that a segment can be discarded as
   an old duplicate if it is received with a timestamp SEG.TSval less than some timestamps recently
   received on this connection.

   Timestamp can be used in a way such that 1 timestamp value can be used for marking 2^36 sequence
   numbers. Timestamp size at current is 2^31-1, therefore the size of timestamp is ample.

   The representation of window, timestamp and sequence space is as follows:

            0--------2^30-1   30 bit window size
            0----------------2^31-1   31 bit timestamp size
            0--------------------------------2^32-1   32 bit sequence size

    The catch: the sequence must be big enough that it doesn't wrap for representing a window.
    If it does then there will be 2 or more packets of same sequence number and timestamp.
    Also similarly timestamp must not wrap in which case it will give rise to same problem.
    Since we have just mentioned that timestamp will not be a problem.

Q. What can be done ?
A. Timestamp alone can gurantee discarding old duplicates or sender-receiver out of phase problem
   if the sequence number space is large enough to accomodate the window size used.

   Since all TCP stacks may not use PAWS therefore the window size that is permissible is 2^31
   and not 2^32. If 2 communicating TCP stacks use PAWS (TCP Extensions for High Performance)
   then in that case it may use window size of 2^32.

Q. So what is limiting Dynamic shift count exchange for window size > 2^32 other than legacy
   interoperability aspect ?
A. Its the sequence number space.
   Consider the following representaion :

            0---2^36-1   36 bit window size
            0---2^37-1   37 bit sequence size

   Therefore for idea to be implemented the other changes we suggest is to make sequence number
   size greater than current 32 bits to 64 bits.

Q. Then who all can use Dynamic Shift Count Feature ?
A. Who has following things in check:
    - sequence number space is 64 bits
    - use PAWS for extention. (may not be needed if we can provide 2 windows for window
                               size used for sender-receiver out-of-phase scenario if
                               bew size of sequence number i.e. 64 bits always allows)
    - both end of TCP connection have new TCP stack with Dynamic Shift Count exchange feature.

==========
Growing and Shrinking of window with shf.cnt > 14 :
==========
When there is need to have a shf.cnt of > 14 we can query following checks from the system assuming
its using new TCP stack with Dynamic Shift Count exchange feature.

1. max speed of interface link on which TCP connection is established along with current Eff.snd.MSS
   to know the actual Tx rate. If Eff.snd.MSS changes then this may trigger change in shift count.
2. check to see if memory is available for max speed interface link possible.
3. check to see if application supports consumption of such high rate of data.

Checks 1 and 2 present a dynamic nature to changing shf.cnt. We know that since RETRACTED WINDOW due
to shft.cnt granularity was already in place as in RFC-7323 and also since "shrinking of window" and
"growing of window" was already there in RFC-793 and RFC-1122 for zero shf.cnt hence we don't have
to worry and implement on that logic when we introduce shf.cnt of 15 or more. Regrading Check 3, an
application too can in-directly shrink a window if it doesn't consume segments in RCV.BUF in
sufficient rate. In this case TCP can shrink the window on its behalf to communicate this to the
sending TCP. This behavior is known in second point of section "2.4. Addressing Window Retraction"
RFC-7323 p.11.

Hence by induction the above logic will also be there for shf.cnt > 14.

==========
Migration Notes :
==========
It takes 49.6 days for a 32 bit timestamp number to wrap again when timestamp tick is of interval
1ms. If we were to increase sequence numbers space to 64 bits, then we have to change timestamp size
as well if need is realized, otherwise, I feel 64 bit sequence space is enough to discard old
duplicates.

When sequence number size is 32 bits:
timestamp tick = 1ms
intf speed   = 1TbE/s
window size   = 1GB (2^30) can be transmitted in 1ms

If we want to retain PAWS timestamp for 64 bit sequence number, then we need to have a timestamp
size of 64 bits. Therefore it takes 213503982334.6 days (or 584942417 years) for a 64 bit timestamp
space to wrap again when timestamp tick is of interval 1ms. Certainly this is surplus amount of
interval and we need to normalize assuming 49.6 days of wrap for timestamp tick of 1ms which meets
our needs. The calculation is as follows:

213503982334.6 * x = 49.6 days
x = 0.0000000002323141679
x = 0.2 ns (normalized to 49.6 days wrap interval)

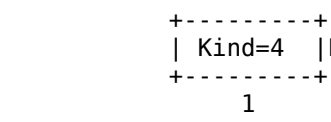
Therefore timestamp tick reaches to 0.2ns (i.e. 200us)

Example:
When sequence number and timestamp size is 64 bits:
timestamp tick = 200 us
interface speed = 64 TbE/s
1000000us   = 64000 GB
for 1us tick   = 64MB of window
for 200us   = 12.8 GB (~2^34) of window (WSopt/shf.cnt of 18 is required)

Summary:
We need sequence space to increase from 32 bit to 64 bit. If we want to retain PAWS then we need to
increase timstamp space as well to 64 bit. PAWS may be rendered obsolete with such a large sequence
space.

==========

Algorithm :
============
1. Connection establishment phase: When SYN or SYN-ACK flag is enabled in a packet then we simply
   exchange WSopt as is done. It will also contain new TCP option EXP.



2. On seeing EXP=1 it sets SND_DS_CAP=TRUE. SND_DS_CAP will be checked in data transfer phase
   and if TRUE then it signifies that other end TCP stack has DSEP feature and will be enabled
   to send request.

3. We have introduced 1 new TCP option 5 variable flags for implementing Dynamic Shift Count
   exchange protocol. Following are their definitions:

   EXP   signifies that the other end TCP stack supports dynamic shift count.
              EXP is for 'Exponent' whose presence in received packet's TCP option field tells
              that the other end TCP stack supports dynamic shift count.
              Otherwise its absence implies that other end TCP stack is legacy and doesn't support
              dynamic shift count feature. (It may support static shf.cnt = 15. The work is
              in progress by Yoshi !)

            = 1 signifies a request and also its capability to support dynamic shift count.
              Will only be set by a Host TCP stack supporting dynamic shift count. On
              receiving EXP=1 from other end, local TCP stack will set SND_DS_CAP = TRUE.

            = 0 If received in TCP connection establishment phase (in SYN or SYN-ACK packets) then
              it means that though dynamic shift feature is supported at other end TCP stack but it
              is currently not ready to change it dynamically in data transfer phase for this
              TCP connection. (Its same as its absence of EXP option for old TCP stacks and dynamic
              shift count change won't be done)
              If received in data transfer phase then 'WSopt' is considered as an response to
              request of shift count change.

   Following are variable flags and represents other end TCP stack's DS protocol process state
   maintained locally:

SND_DS_CAP Sending Dynamic Shift Count Capability. Is set in TCP connection establishment phase
            (in SYN or SYN-ACK packets) depending on the EXP value received. If EXP = 1 then it
            is set to true otherwise EXP=0 or its absence make it set to FALSE. Its checked in
            during Data Transfer Phase when a host needs to send a DS change request but only be
            sent if its set to TRUE.

            = TRUE, then sender can send request change during data transfer phase. Its NOT
            checked in TCP connection establishment phase when sending SYN or SYN-ACK packets.
            Its initial value before setting in TCP connection establishment phase is NULL.

            = FALSE then sender is not allowed to send 'EXP' or 'WSopt' in data transfer phase.
              Its initially set to this value TCP connection establishment phase and doesn't
              change if other end is old TCP stack

SND_SHF_REQ Sending Shift Request. Its used in Data transfer phase of TCP connection and is set
            to requested shf.cnt value from other end on receiving REQ packet, when its NULL,
            otherwise it cannot accept a request and it be dropped. It means that already a
            request of shift count is received at this end and change is in progress. Hence its
            used as a variable flag. It is used to set locally maintained oEnd::shf.cnt when WIN
            packet is received. It holds either a requested shift count from other end TCP stack
            or a NULL value.

            = requested shift count value in REQ packet received from other end TCP stack

            = NULL. Initial set to NULL before TCP connection startup phase starts.
            Its reset to this value after its use of setting locally maintained
            oEnd::shf.cnt when local TCP receives WIN packet. Its also reset to NULL when it has
            sent RSP_NOK packet. Its compulsory to reset it to NULL after use since its used to
            prohibit an outstanding request from other end when already one is in progress.

Following are variable flags concerning local DS protocol process state:

RCV_SHF_REQ Receiving Shift Request. Its used in Data transfer phase of TCP connection and
            signifies the current findings of the system to support shf.cnt which should be sent
            as request in REQ packet to other end. Its set when its current value is NULL and REQ
            packet is sent successfully otherwise if its not NULL then it means that other
            request is already in progress and it cannot send. Hence its used as a variable flag.
            Its used to set local own::shf.cnt when RSP_OK packet is received. It holds either
            shift count requested to other end TCP stack or a NULL value.

            = requested shift count, sent in REQ packet to other end TCP stack.

            = NULL. Initial set to NULL before TCP connection startup phase starts.
            Its reset to this value after its use of setting local own::shf.cnt when local
            TCP receives RSP_OK packet. Its also reset to NULL when it receives RSP_NOK packet.
            Its compulsory to reset it to NULL after use since its used to prohibit a
            request to be sent by this end when already one is in progress.

RSP_SEQ   Response sequence number. Its used in Data transfer phase of TCP connection and holds
            the sequence number of response (RSP_OK) sent acknowledging request packet (REQ) from
            other end TCP stack. This value will is used to match and check when WIN packet is
            received later. Its set when its current value is NULL and a RSP_OK packet needs
            to be sent. Its not set when sending RSP_NOK packet as an acknowledgment to REQ
            packet. Its use as a variable flag is to provide sanity check when parsing packets.

            = sequence number of RSP_OK packet sent in acknowledgment to REQ packet.

            = NULL. Initial set to NULL before TCP connection startup phase starts.
            Its reset to this value after its use of matching and checking received WIN packet.

REQ_SEQ   Request sequence number. Its used in Data transfer phase of TCP connection and holds
            the sequence number of request packet (REQ) being sent. This value will be used to
            match and check RSP_OK or RSP_NOK packet received later as response to REQ being
            sent. Its set when its current value is NULL and a REQ packet needs to be sent.
            Its use as a variable flag is to provide sanity check when parsing packets.

            = sequence number of REQ packet being sent.

            = NULL. Initial set to NULL before TCP connection startup phase starts.
            Its reset to this value after its use of matching and checking received RSP_OK or
            RSP_NOK packet.

These variables will be updated according to state changes.

 Changes to these variables will cause change in following existing TCP connection variables:
 introduced in previous RFCs:
 namespace own
     {
     shf.cnt   /* represents shift count applied when sending rWnd as window update
                  * This is update when we receive RSP_OK packet from other end.
                  */
     rWnd   /* got from ops rBufMem>>shf.cnt & published in TCP::Window field */
     RCV.WND   /* got from ops rWnd<<shf.cnt & is offset from RCV.NXT for use in
                    segment acceptability test */
     SND.WND   /* got from ops oEnd::rWnd<<oEnd::shf.cnt & is offset from SND.UNA */
     SND.WL1   /* segment sequence number used for last 'window update' and is updated
                  * on receiving of WIN packet which passes 'window update' acceptability test
                  */
     SND.WL2   /* segment acknowledgment number used for last 'window update' and is updated
                  * on receiving of WIN packet which passes 'window update' acceptability test
                  */
     }

 namespace oEnd
     {
     rWnd   /* received in TCP::Window field */
     shf.cnt   /* represents shift count applied when receiving rWnd as window update
                  * from other end TCP stack. This is updated when WIN packet is received
                  */
     }

 Note: namespace 'own' represents local TCP connection variables whereas namespace 'oEnd'
       represents remote end's TCP connection variables maintained locally

 Example usage: own::rWnd represent local recieve window to be send in TCP window field whereas
                oEnd::rWnd represents recieve window received in TCP window field from other end

4. If Host A is an old TCP it neglects processing of the 'EXP' option in TCP connection
   establishment phase. Host B which is a new stack observing absence of 'EXP' option will
   understand that Host A doesn't support dynamic changing of shift count.

   Note:
   Remember when we mention about dynamic shift count use-case then we implicitly mean that it

supports both i.e shf.cnt==15 and dynamic shift count > 14. When we mention new TCP stack then we mean that it has support for Dynamic Shift Count Exchange Protocol (DSEP).

5. State Machine:
Convention to followed is that for reading state machine is :

Single Event is represented as "PacketType1_Received"
Combined Event is represented as "PacketType1_Received-PacketType2_Received"

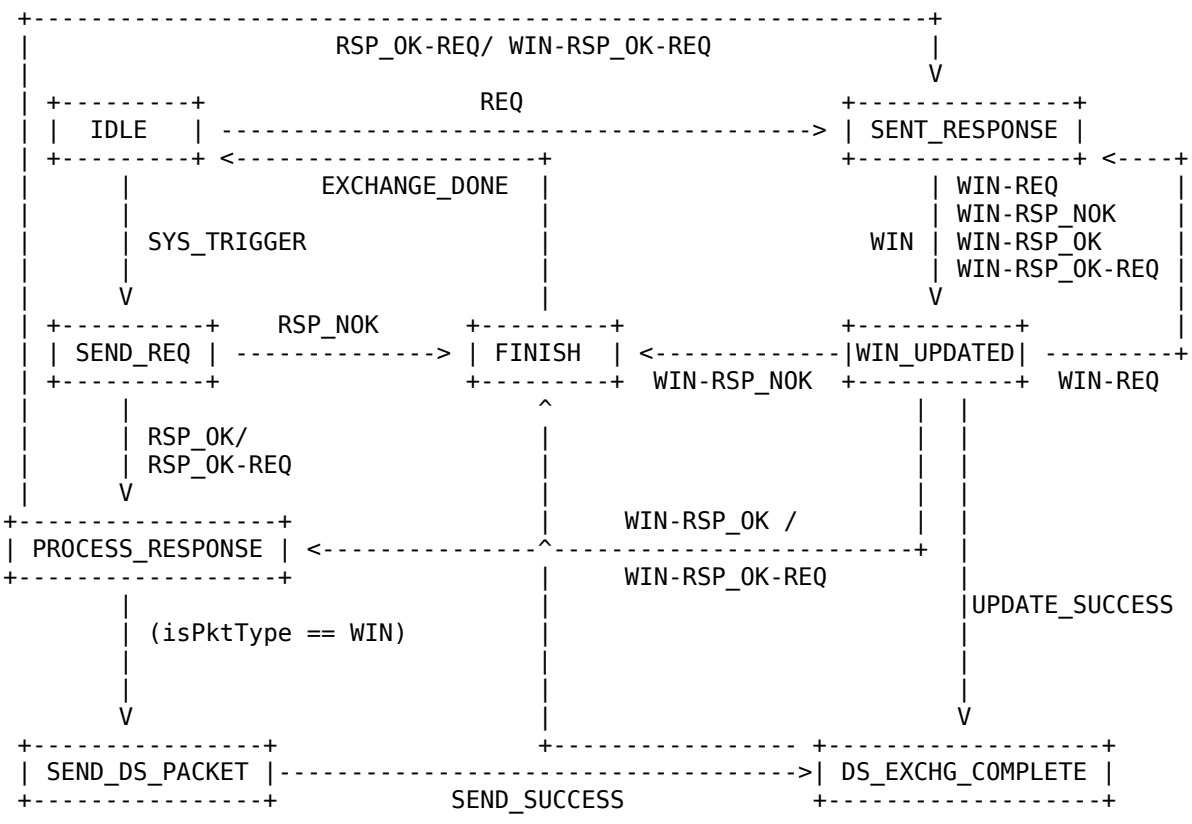
The events in below state machine are 'DSPktType' packet type received and named after them, These events are classified into 3 classes directly related to packet types:

a. External Single Event : signifies a single packet type that is dequeued from incoming interface queue for processing. REQ, RSP_OK, RSP_NOK, WIN

b. External Combined Event : signifies a combined packet types (due to piggybacking) as a single packet that is dequeued from incoming inerface queue for processing. The combined event follows following order for processing:
WIN > RSP_OK(or RSP_NOK) > remote REQ > local REQ to be sent

RSP_OK-REQ, WIN-REQ, WIN-RSP_NOK, WIN-RSP_OK, WIN-RSP_OK-REQ

c. Local events : These are internal DSEP protocol process events generated when it reaches some meaningful outcome.
SYS_TRIGGER, SEND_SUCCESS, UPDATE_SUCCESS, EXCHANGE_DONE



We send combined or single packet here !

6. Describing 'DSPktType' or External Single or Combined Events:

REQ : packet is used to carry dynamic shift count change request to other end TCP stack
RSP_NOK : packet carries NOK acknowledgment from other end which declined the request because of system constraints (like its current CPU load, etc.)
RSP_OK : packet carries OK acknowledgment indicating the acceptance of other end TCP stack of the requested shift count
WIN : packet carries window update calculated by requester using new shift count approved by other end TCP stack. This is the final message of a dynamic shift count exchange process.
RSP_OK-REQ : packet is same as RSP_NOK but caries REQ from other end combined or piggybacked along with it.
WIN-REQ : packet is same as WIN but caries REQ from requester end combined or piggybacked along with it
WIN-RSP_NOK : packet is same as WIN but caries NOK acknowledgement (i.e. combined with RSP_NOK) to the request of other end TCP stack
WIN-RSP_OK : packet is same as WIN but caries OK acknowledgement (i.e. combined with RSP_OK) to the request of other end TCP stack
WIN-RSP_OK-REQ : packet is same as WIN but caries OK acknowledgement (i.e. combined with RSP_OK) to the request of other end TCP stack and also a new dynamic shift count change request.

7. Describing state of DSEP process :

==========
IDLE state :
==========
This is the default state when TCP connection is being established and it will remain in this state until Data Transfer phase starts. In this state 'SND_DS_CAP' will be set to either TRUE or FALSE depending on EXP value of 1 or 0. If EXP=0 then this state will be persistent in Data Transfer Phase and no change is shift count request is accepted.
Otherwise if EXP=1 then exchange can be done.

```
InGuard : isPktType(aPkt)==REQ else "Drop Packet"
Event   : REQ
Action  : ProcessREQ()
OutGuard: if (SYS_TRIGGER event received)
            {
            if(canSendREQ(reqShf))
                {
                CanCombineRSP-REQ(combine);
                }
            Send(combine);
            }
Trans to: SENT_RESPONSE


InGuard : isEventType(aEvt)==SYS_TRIGGER else "Drop Packet"
Event   : SYS_TRIGGER
Action  : canSendREQ(reqShf)
OutGuard: if(isREQBackoffTimerRunning)
            {
            if(reqShf > backoffShf)
                {
                CancelREQBackoffTimer();
                Send(combine);
                }
            else
                {
                /* wait for timer to expire
                 */
                WaitREQBackoffTimer();
                }
              }
            else
              {
              Send(combine);
              }
    Trans to: SEND_REQ
```

Note: Backoff Timer find its requirement in scenario when both end of the TCP connection send REQ simultaneously in which case when packets reach either ends they will fail sanity check and will be dropped and replied by RSP_NOK packet. When receiving RSP_NOK the sending TCP must back off before sending the same request, but if a new request is locally generated i.e. of higher shift count then it will reset the ongoing timer.

One peril is how to
distinguish the intent of RSP_NOK, whether it was sent due to conflict or b'cause system doesn't currently support. If we don't use Backoff timer then it will hit performance when we instead choose to drop valid data packets.

```
// ========================================================================//
//                                                                        //
// TODO: Provide explaination for rest of the states in state machine  //
//                                                                        //
// ========================================================================//
```

8. General Notes :
[1] A TCP connection may have asymmetric Tx rate even when both ends have support of > 1Gb window. Traffic from A->B will support more Tx rate as facilitated by B than B->A which is constrained by A. We have assumed that CPU speed is not a limiting factor.

[2] TCP A realizes that it can have more RCV.BUF in which case it must communicate this info
    to B using shft.cnt in WSopt field. Past behvior was that these are only exchanged during
    connection ESTBLISMENT state and not after. All acceptability tests like:
    1. segment
    2. acknowledgment
    3. window update
    4. retransmission
    will seamlessly handle increase in window size. We plan to use same TCP WSopt during data
    transfer phase along with new TCP options EXP. The condition is that these shf.cnt update
    will be transmitted only reliably on data segment or on non-empty ack segment (NOT using
    empty ack segment or pure ack)

[3] Though the protocol could have been made simpler but combining/ piggybacking different
    packet types together was done purposefully for high performance. Without combining packet
    types the normal flow of state changes are:
    a. sender side are :
        IDLE--->REQ--->PROCESS_RESPONSE--->SEND_DS_PACKET--->DS_EXCHG_COMPLETE--->FINISH--->IDLE
    b. receiver side are:
         IDLE--->SENT_RESPONSE--->WIN_UPDATED--->DS_EXCHG_COMPLETE--->FINISH--->IDLE

[4] If WIN packet gets reordered at receive end within a RTT then it will have no effect at
    receiver end. Receiver will keep sending packets SND.WND calulated from now old shift
    count (since WIN packet hasn't reached it yet). The only effect that it will have is that
    the receiver MAY turn up sending less packets than can be accepted at sender end which
    uses new shift count and 'window update'. This is the delay induced. There will be no
    packets dropped since current DSEP algorithm uses increasing shift count in exchange
    than previous one (ref. point [5] for use-cases supported).

    Ref RFC 0793 Section pg. 72
    Ref RFC 7323 Section "Addressing Window Retraction" pg. 10
    Ref RFC 1122 Section 4.2.3.3 "When to Send a Window Update"

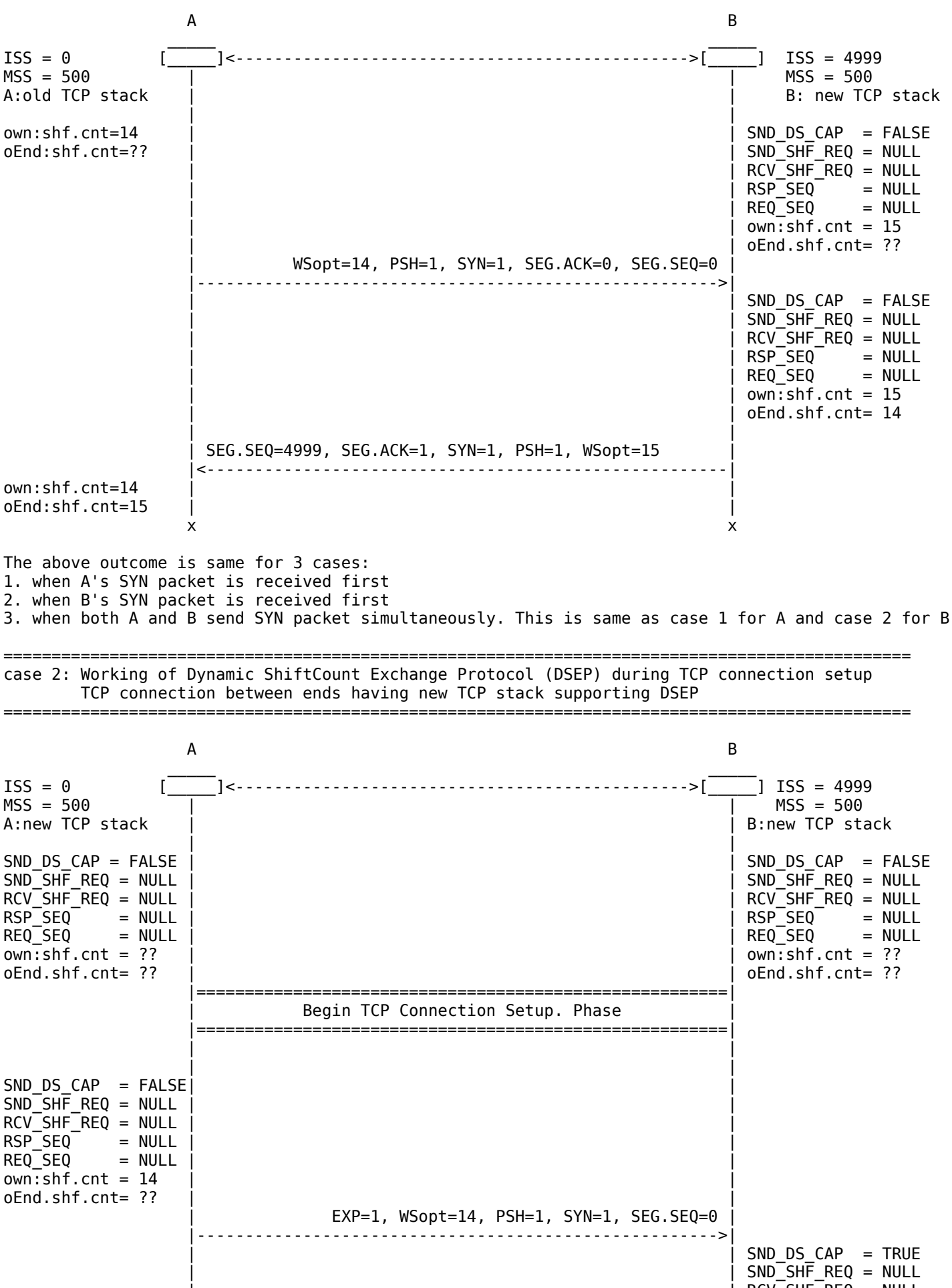
    Also the algorithm bestows faith and hence depends on existing implementation of TCP stack
    that they have following :
    a. segment acceptability test
    b. acknowledgment acceptability test
    c. window update acceptability test

    Also it has dependency on RFC 7323 for (TCP Extensions for High Performance) for following:
        d. Addressing Window Retraction
        e. PAWS (if its retained even when we move to sequence number of 64 bits size)

[5] In above implementation we have RSP_OK combined with "inc shf.cnt REQ"
    In above implementation we don't have RSP_NOK combined with "inc shf.cnt REQ", may in future
    In above implementation we don't have RSP_NOK combined with "dec shf.cnt REQ", may in future
    In above implementation we don't have RSP_OK combined with "dec shf.cnt REQ", may in future

==
Interoperability and Algorithm Use-cases :
==

==
case 1: Interoperability between old TCP stack and new TCP stack
        WSopt getting exchanged only in SYN and SYN-ACK segments i.e. TCP connection setup
==

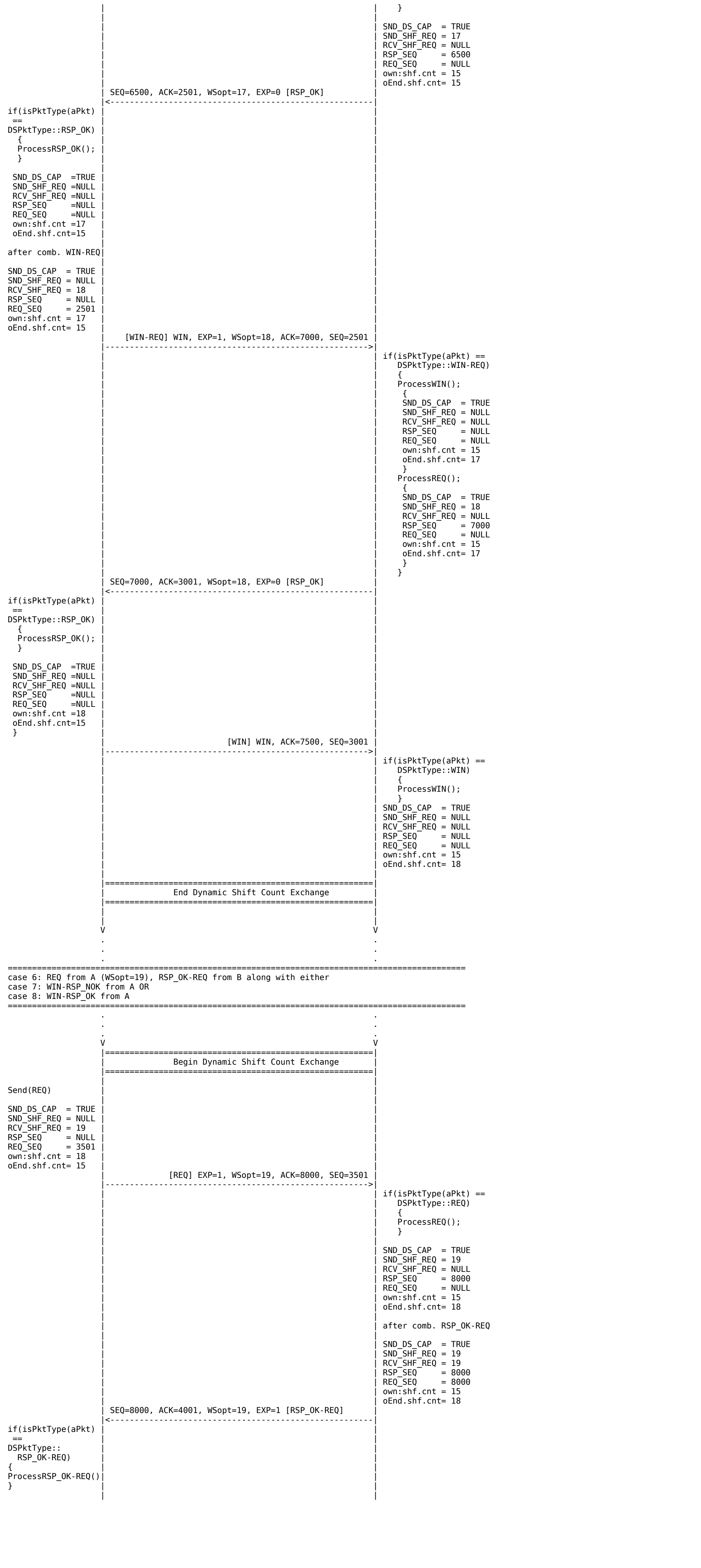


The above outcome is same for 3 cases:
1. when A's SYN packet is received first
2. when B's SYN packet is received first
3. when both A and B send SYN packet simultaneously. This is same as case 1 for A and case 2 for B.

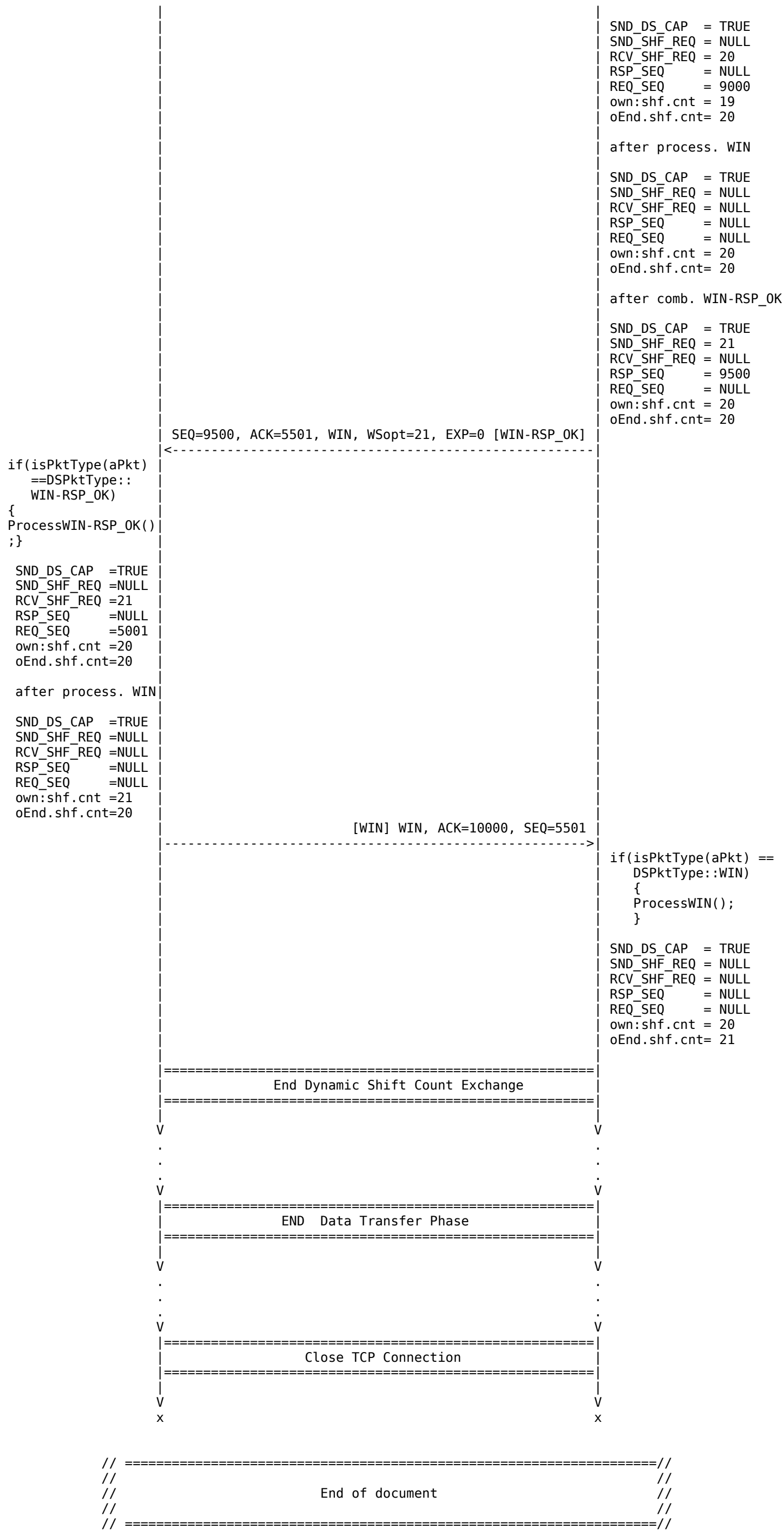
==
case 2: Working of Dynamic ShiftCount Exchange Protocol (DSEP) during TCP connection setup
        TCP connection between ends having new TCP stack supporting DSEP
==

==
case 3: REQ from A for WSopt=15 and RSP_OK from B and WIN by A :
==

Begin Dynamic Shift Count Exchange
Send(REQ)
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = 15
RSP_SEQ = NULL
REQ_SEQ = 501
own:shf.cnt = 14
oEnd.shf.cnt= 15
[REQ] EXP=1, WSopt=15, ACK=5000, SEQ=501
if(isPktType(aPkt) == DSPktType::REQ) { ProcessREQ(); }
SND_DS_CAP = TRUE
SND_SHF_REQ = 15
RCV_SHF_REQ = NULL
RSP_SEQ = 5000
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 14
SEQ=5000, ACK=1001, WSopt=15, EXP=0 [RSP]
if(isPktType(aPkt) == DSPktType::RSP_OK) { ProcessRSP_OK(); }
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = NULL
RSP_SEQ = NULL
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 15
[WIN] WIN, ACK=5500, SEQ=1001
if(isPktType() == DSPktType:WIN) { ProcessWIN(); }
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = NULL
RSP_SEQ = NULL
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 15
End Dynamic Shift Count Exchange
case 4: REQ from A for WSopt=16 and RSP_NOK from B :
Begin Dynamic Shift Count Exchange
Send(REQ)
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = 16
RSP_SEQ = NULL
REQ_SEQ = 1501
own:shf.cnt = 15
oEnd.shf.cnt= 15
[REQ] EXP=1, WSopt=16, ACK=6000, SEQ=1501
if(isPktType(aPkt) == DSPktType::REQ) { ProcessREQ(); }
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = NULL
RSP_SEQ = 5000
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 15
SEQ=6000, ACK=2001, WSopt=15, EXP=0 [RSP_NOK]
if(isPktType(aPkt) == DSPktType::RSP_NOK) { ProcessRSP_NOK(); }
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = NULL
RSP_SEQ = NULL
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 15
End Dynamic Shift Count Exchange
case 5: REQ from A (WSopt=17), RSP_OK from B & WIN-REQ from A :
Begin Dynamic Shift Count Exchange
Send(REQ)
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = 17
RSP_SEQ = NULL
REQ_SEQ = 1501
own:shf.cnt = 15
oEnd.shf.cnt= 15
[REQ] EXP=1, WSopt=17, ACK=6500, SEQ=2001
if(isPktType(aPkt) == DSPktType::REQ) { ProcessREQ();


}
SND_DS_CAP = TRUE
SND_SHF_REQ = 17
RCV_SHF_REQ = NULL
RSP_SEQ = 6500
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 15
SEQ=6500, ACK=2501, WSopt=17, EXP=0 [RSP_OK]
if(isPktType(aPkt) == DSPktType::RSP_OK) { ProcessRSP_OK(); }
SND_DS_CAP =TRUE
SND_SHF_REQ =NULL
RCV_SHF_REQ =NULL
RSP_SEQ =NULL
REQ_SEQ =NULL
own:shf.cnt =17
oEnd.shf.cnt=15
after comb. WIN-REQ
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = 18
RSP_SEQ = NULL
REQ_SEQ = 2501
own:shf.cnt = 17
oEnd.shf.cnt= 15
[WIN-REQ] WIN, EXP=1, WSopt=18, ACK=7000, SEQ=2501
if(isPktType(aPkt) == DSPktType::WIN-REQ)
{
ProcessWIN();
{
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = NULL
RSP_SEQ = NULL
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 17
}
ProcessREQ();
{
SND_DS_CAP = TRUE
SND_SHF_REQ = 18
RCV_SHF_REQ = NULL
RSP_SEQ = 7000
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 17
}
}
SEQ=7000, ACK=3001, WSopt=18, EXP=0 [RSP_OK]
if(isPktType(aPkt) == DSPktType::RSP_OK) { ProcessRSP_OK(); }
SND_DS_CAP =TRUE
SND_SHF_REQ =NULL
RCV_SHF_REQ =NULL
RSP_SEQ =NULL
REQ_SEQ =NULL
own:shf.cnt =18
oEnd.shf.cnt=15
[WIN] WIN, ACK=7500, SEQ=3001
if(isPktType(aPkt) == DSPktType::WIN)
{
ProcessWIN();
}
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = NULL
RSP_SEQ = NULL
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 18
End Dynamic Shift Count Exchange
case 6: REQ from A (WSopt=19), RSP_OK-REQ from B along with either
case 7: WIN-RSP_NOK from A OR
case 8: WIN-RSP_OK from A
Begin Dynamic Shift Count Exchange
Send(REQ)
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = 19
RSP_SEQ = NULL
REQ_SEQ = 3501
own:shf.cnt = 18
oEnd.shf.cnt= 15
[REQ] EXP=1, WSopt=19, ACK=8000, SEQ=3501
if(isPktType(aPkt) == DSPktType::REQ)
{
ProcessREQ();
}
SND_DS_CAP = TRUE
SND_SHF_REQ = 19
RCV_SHF_REQ = NULL
RSP_SEQ = 8000
REQ_SEQ = NULL
own:shf.cnt = 15
oEnd.shf.cnt= 18
after comb. RSP_OK-REQ
SND_DS_CAP = TRUE
SND_SHF_REQ = 19
RCV_SHF_REQ = 19
RSP_SEQ = 8000
REQ_SEQ = 8000
own:shf.cnt = 15
oEnd.shf.cnt= 18
SEQ=8000, ACK=4001, WSopt=19, EXP=1 [RSP_OK-REQ]
if(isPktType(aPkt) == DSPktType:: RSP_OK-REQ) { ProcessRSP_OK-REQ() }

SND_DS_CAP =TRUE
SND_SHF_REQ =NULL
RCV_SHF_REQ =NULL
RSP_SEQ =NULL
REQ_SEQ =NULL
own:shf.cnt =19
oEnd.shf.cnt=15
after comb.
WIN-RSP_OK
SND_DS_CAP =TRUE
SND_SHF_REQ =19
RCV_SHF_REQ =NULL
RSP_SEQ =4501
REQ_SEQ =NULL
own:shf.cnt =19
oEnd.shf.cnt=15
/* Here it may choose to
* send RSP_NOK for case 7
*/
[WIN-RSP_OK] EXP=0, WSopt=19, WIN, ACK=8500, SEQ=4001
if(isPktType(aPkt) == DSPktType::WIN-RSP_OK)
{
ProcessWIN-RSP_OK();
}
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = 19
RSP_SEQ = NULL
REQ_SEQ = 8000
own:shf.cnt = 15
oEnd.shf.cnt= 19
after process. WIN
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = NULL
RSP_SEQ = NULL
REQ_SEQ = NULL
own:shf.cnt = 19
oEnd.shf.cnt= 19
SEQ=8500, ACK=4501, WIN [WIN]
if(isPktType(aPkt) == DSPktType::WIN)
{
ProcessWIN();
}
SND_DS_CAP =TRUE
SND_SHF_REQ =NULL
RCV_SHF_REQ =NULL
RSP_SEQ =NULL
REQ_SEQ =NULL
own:shf.cnt =19
oEnd.shf.cnt=19
End Dynamic Shift Count Exchange
case 9: REQ from A (WSopt=20), RSP_OK-REQ from B and WIN-RSP_OK-REQ from A
Begin Dynamic Shift Count Exchange
Send(REQ)
SND_DS_CAP = TRUE
SND_SHF_REQ = NULL
RCV_SHF_REQ = 20
RSP_SEQ = NULL
REQ_SEQ = 4501
own:shf.cnt = 19
oEnd.shf.cnt= 19
[REQ] EXP=1 WSopt=20, ACK=9000, SEQ=4501
if(isPktType(aPkt) == DSPktType::REQ)
{
ProcessREQ();
}
SND_DS_CAP = TRUE
SND_SHF_REQ = 20
RCV_SHF_REQ = NULL
RSP_SEQ = 9000
REQ_SEQ = NULL
own:shf.cnt = 19
oEnd.shf.cnt= 19
after comb. RSP_OK-REQ
SND_SHF_REQ = 20
RCV_SHF_REQ = 20
RSP_SEQ = 9000
REQ_SEQ = 9000
own:shf.cnt = 19
oEnd.shf.cnt= 19
SEQ=9000, ACK=5001, WSopt=20, EXP=1 [RSP_OK-REQ]
if(isPktType(aPkt) == DSPktType:: RSP_OK-REQ)
{
ProcessRSP_OK-REQ();
}
SND_DS_CAP =TRUE
SND_SHF_REQ =NULL
RCV_SHF_REQ =NULL
RSP_SEQ =NULL
REQ_SEQ =NULL
own:shf.cnt =20
oEnd.shf.cnt=19
after comb.
WIN-RSP_OK
SND_DS_CAP =TRUE
SND_SHF_REQ =20
RCV_SHF_REQ =NULL
RSP_SEQ =5001
REQ_SEQ =NULL
own:shf.cnt =20
oEnd.shf.cnt=19
after comb.
WIN-RSP_OK-REQ
SND_DS_CAP =TRUE
SND_SHF_REQ =20
RCV_SHF_REQ =21
RSP_SEQ =5001
REQ_SEQ =5001
own:shf.cnt =20
oEnd.shf.cnt=19
[WIN-RSP_OK-REQ] EXP=1,WSopt=21,WIN,ACK=9500,SEQ=5001
if(isPktType(aPkt) == DSPktType:: WIN-RSP_OK-REQ)
{
ProcessWIN-RSP_OK-REQ();
}



// ========== //
// End of document //
// ========== //