ENSC 835-3: HIGH PERFORMANCE NETWORKS

DYNAMIC RIGHT-SIZING A TCP FLOW-CONTROL INVESTIGATION IN OPNET

Spring 2006

FINAL PROJECT

CAMILLE JAGGERNAUTH

www.sfu.ca/~jaggerna

jaggerna@sfu.ca

April 30, 2006

This page is intentionally left blank.

Table of Contents

Abstract 5
Introduction 6
Design and Implementation... 9
Discussion 14
Conclusion and Future Work.. 17
References 18
Appendix - Code Listing - tcp_conn_v3.pr.c 19

List Of Figures

Figure 1: TCP Segment Definition Layout [3] 6
Figure 2: Receive Window Layout 8
Figure 3: OPNET Setup 9
Figure 4: OPNET Parameter Layout 9
Figure 5: Node Model for Client / Server 10
Figure 6: TCP Manager Process Model 11
Figure 7: TCP Connection Process Model 12
Figure 8: Expected Results for Data Transfers [1] 14
Figure 9: Expected Results for Window Sizes [1] 15
Figure 10: Actual Results 16

Abstract

“Grid and networking researchers continue the practice of manually optimizing TCP buffer sizes to keep the network pipe full, and thus achieve acceptable performance over the wide-area network. Not only is this process cumbersome, but the result oft tuning window sizes for a particular pair of hosts is sub-par performance for connections with larger delay-bandwidth products and the misappropriation of scarce resources to connections with smaller delay-bandwidth products” [1]. “Dynamic Right sizing lets the receiver estimate the sender’s congestion window size and uses that estimate to dynamically change the size of the receiver’s window advertisements. As a result, the sender will be congestion-window-limited rather than flow-control-window-limited” [1].

This project will investigate a simple OPNET implementation of Dynamic Rightsizing.

Introduction

Generally, TCP buffer sizes are manually optimized. Optimization is key because it keeps the network pipe full and improves network performances over a wide-area network.

However, manual optimization is cumbersome and may not adequately serve the needs of network products with varying delay bandwidths.

Dynamic Rightsizing was proposed by Mike Fisk and Wu-chun Feng as a solution to this issue. Originally implemented in **ns** as part of a 3 year project, dynamic rightsizing dynamically adjusts the receiver's window size to match the congestion window size. This results in the sender being congestion window controlled limited instead of flow control window limited.

The receiver window can be seen in Figure 1 below showing the TCP segment.

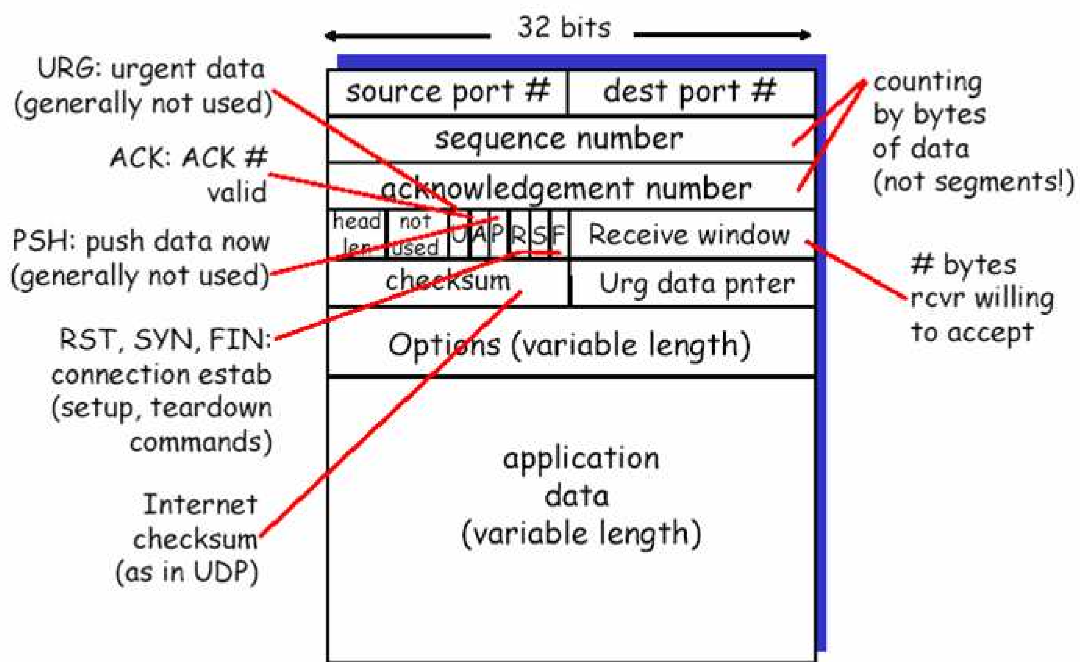

Figure 1: TCP Segment Definition [3]

Introduction

The OPNET definition of the TCP segment is to be found in tcp_seg_support.h and is shown below as well.

```
/* Data structure for fields in the   */
/* tcp segment.                                  */
typedef struct
   {
        int                          src_port;
        int                          dest_port;
        unsigned int                 seq_num;
        unsigned int                 ack_num;
        unsigned int                 rcv_win;
        int                          urgent_pointer;
        int                          data_len;

        /* The following represents bytes 13 and 14 of the TCP header.                 */
        /*
                                 */
        /*   0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15               */
        /*  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+         */
        /*  |           |           | C | E | U | A | P | R | S | F |     */
        /*  | Header Length |   Reserved   | W | C | R | C | S | S | Y | I |         */
        /*  |           |           | R | E | G | K | H | T | N | N |     */
        /*  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+         */
        /*
                                 */
        TcpT_Flag            flags;          /* Information from bits 8 through 15.                 */

        /* The following two fields are not part of */
        /* the standard TCP header, but are used in     */
        /* the OPNET TCP model for enhancing the        */
        /* simulation performance. Each TCP process     */
        /* assigns itself a unique "key" which can      */
        /* be used to perform fast lookup when ever     */
        /* is a need to match a connection ID.          */
        OmsT_Dt_Key                  local_key;
        OmsT_Dt_Key                  remote_key;
   } TcpT_Seg_Fields;
```

Introduction - Flow Control in TCP

Dynamic Rightsizing is a flow control mechanism. Flow control ensures that the sender does not overflow the receiver's buffer by transmitting too much, too fast. This is done by advertising any spare room in the receiver window field of the TCP segment. The flow control mechanism is also pictorially represented below in Figure 2.

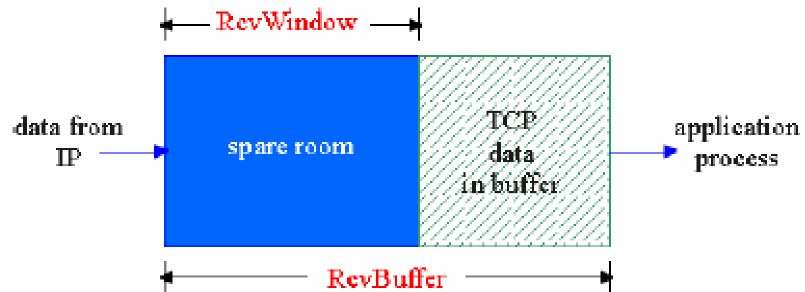

Figure 2: Receive Window Layout

Design and Implementation - OPNET Setup

The OPNET setup is shown below in Figure 3 - a simple FTP client server model. The client, server attributes used were the default OPNET attributes.

Figure 4 details the FTP application attributes. These are also the default FTP OPNET attributes - a file size of 1.6M using a constant distribution.

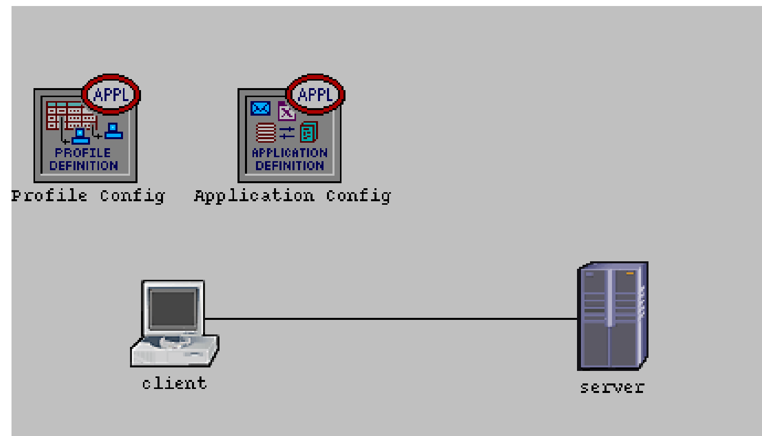

Figure 3: OPNET Project Setup

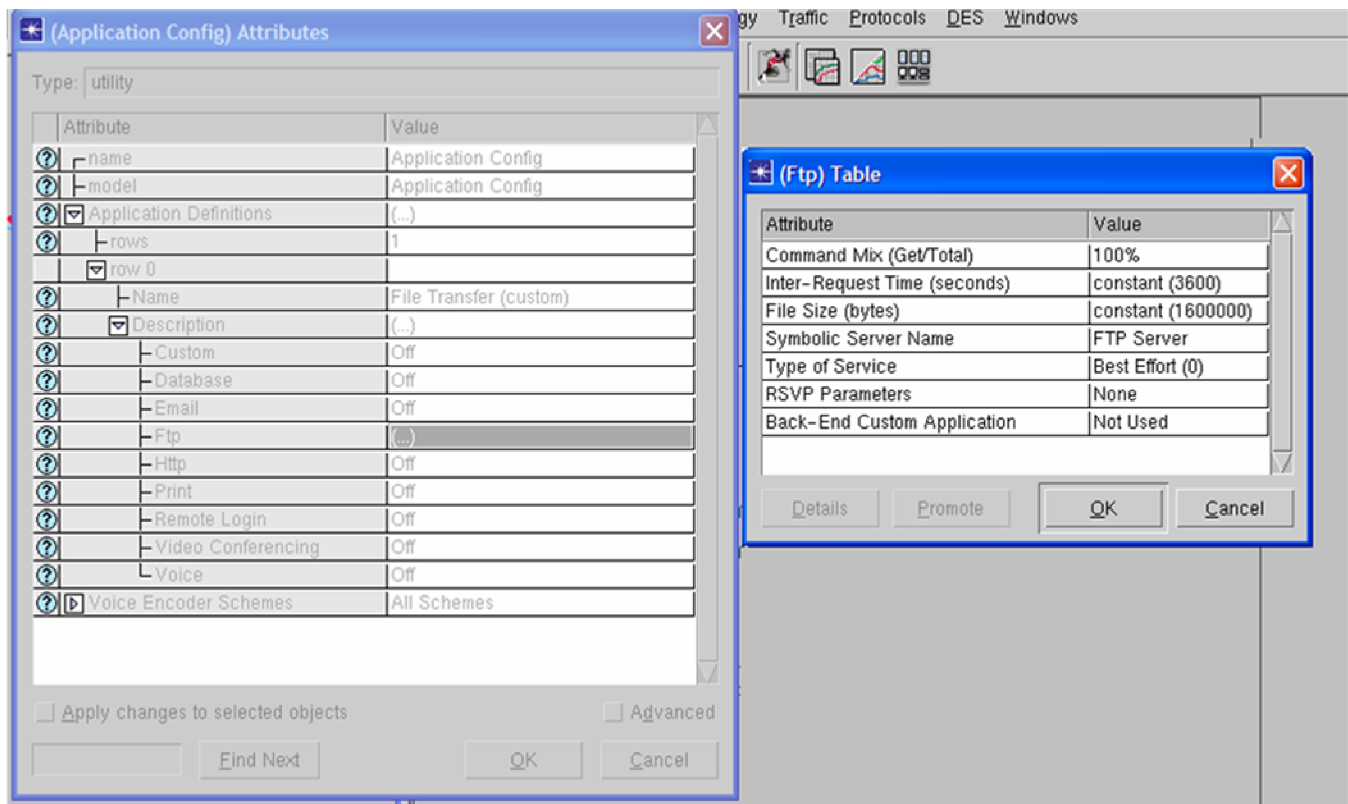

Figure 4: OPNET Project Parameters

Design and Implementation - TCP Node Model

Figure 5 below shows the client/server OPNET node model showing the interaction between the different network layers.

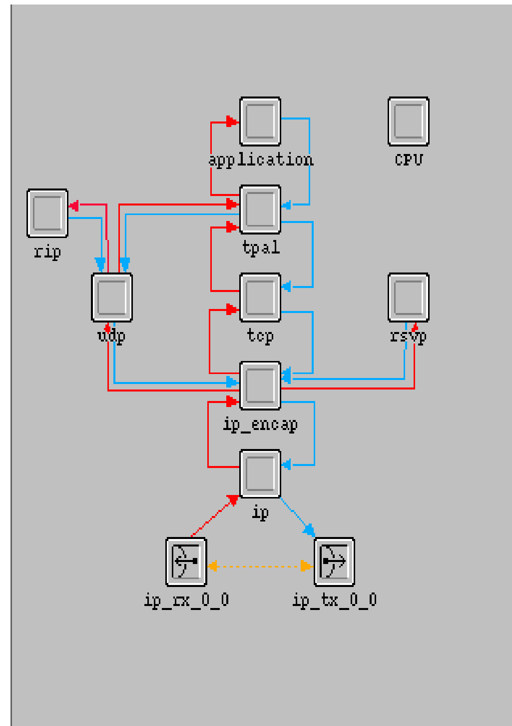

Figure 5: Node Model for Client / Server

Design and Implementation - TCP Process Manager Model

Figure 6 below shows the TCP Manager Process Model. The TCP Manager Process Model represents the root process of the “tcp” module. It manages a set of TCP connections by invoking the appropriate api processes. When this process model is compiled it yields the tcp_manager_v3.c file.

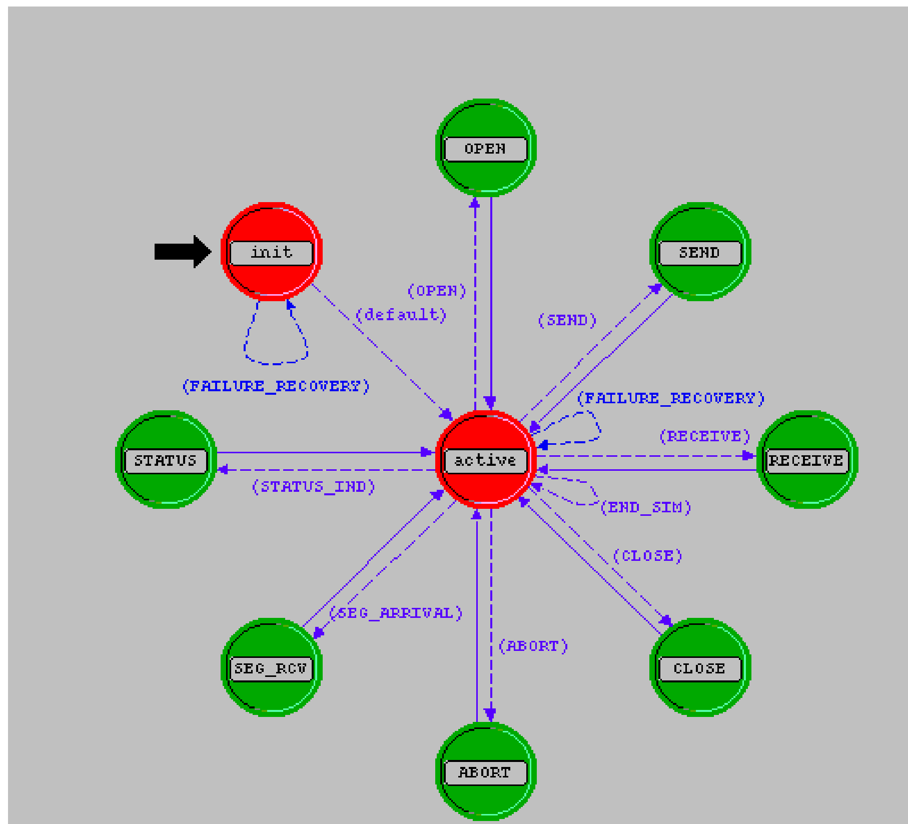

Figure 6: TCP Manager Process Model

Design and Implementation - TCP Connection Model

Figure 7 below shows the TCP Connection Process Model. The TCP Connection Process Model represents the TCP connection (i.e., “socket”) process, which manages one end of a single connection. It is spawned by the tcp_manager_v3 process to manage activities related to an individual connection. When this process model is compiled it yields the tcp_conn_v3.c file.

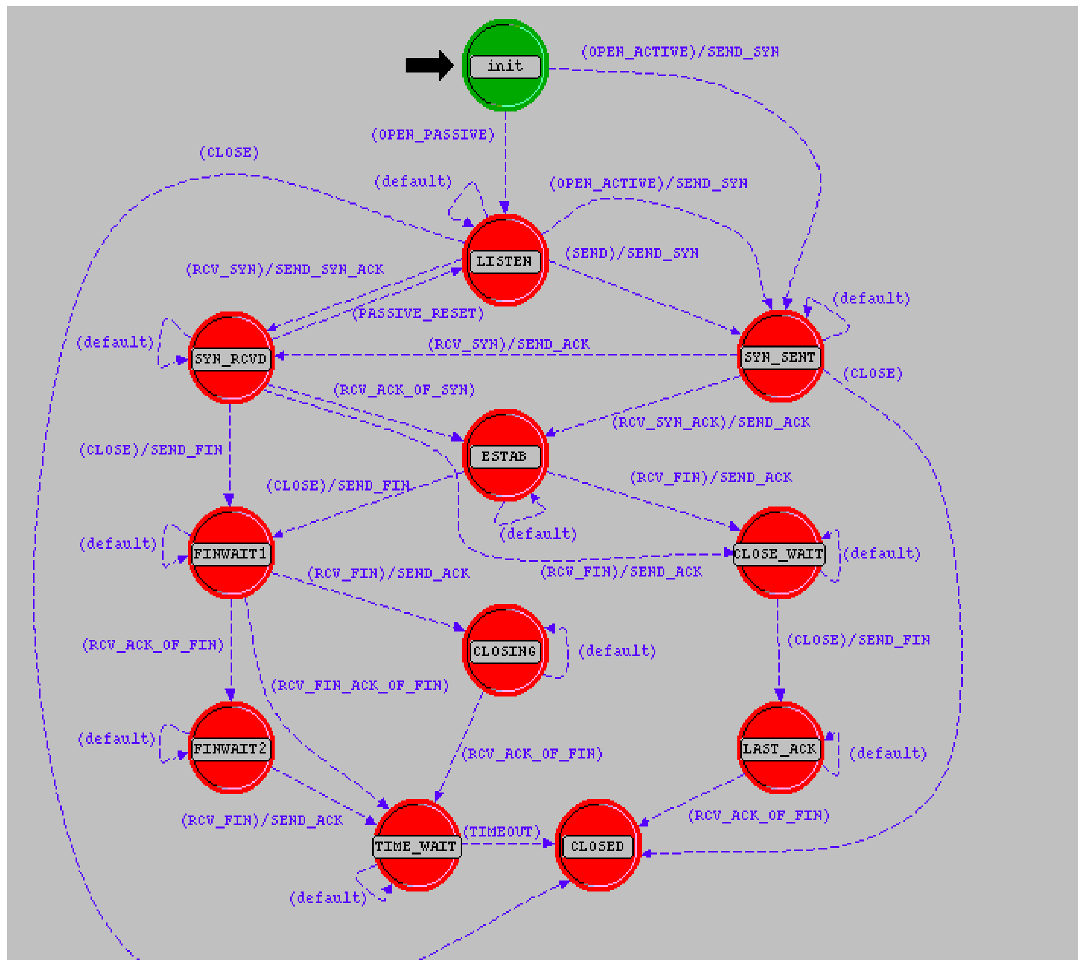

Figure 7: TCP Connection Process Model

Design and Implementation

The project implementation required a thorough understanding of the two TCP process models - TCP manager and TCP connection. This was necessary in order to do the dynamic right sizing implementation as well as not to break any of the existing functionality.

The implementation required a scaled down TCP model where the only option that was permitted was to enable window scaling. This was necessary to allow the receive window to grow beyond 64K standard operating restriction.

The receive window is adjusted in the **LISTEN, SYN_RCVD, SYN_SENT** and **ESTAB** states of the tcp connection process model.

The actual code changes required a simple assignment of the congestion window to the receive window as show in below code excerpt.

```
/* Set the advertized window. If the usage threshold is set      */
/* to zero, then the complete receive buffer is advertized.      */
if (rcv_buf_usage_thresh == 0.0)
        {
                rcv_wnd = cwnd;
        }
```

The changes were specifically made to the **tcp_seg_send, tcp_rcv_buf_process**, and **tcp_seg_receive** routines.

With these changes in place, the simulation runs were executed with the default OPNET operational parameters.

Discussion

OPNET was setup to collect the below statistics:

- Data Transfer
- Congestion Window Size
- Receive Window Size
- Flight Size

The expected results shown in Figures 8 and 9 were then compared against the actual results shown in Figure 10.

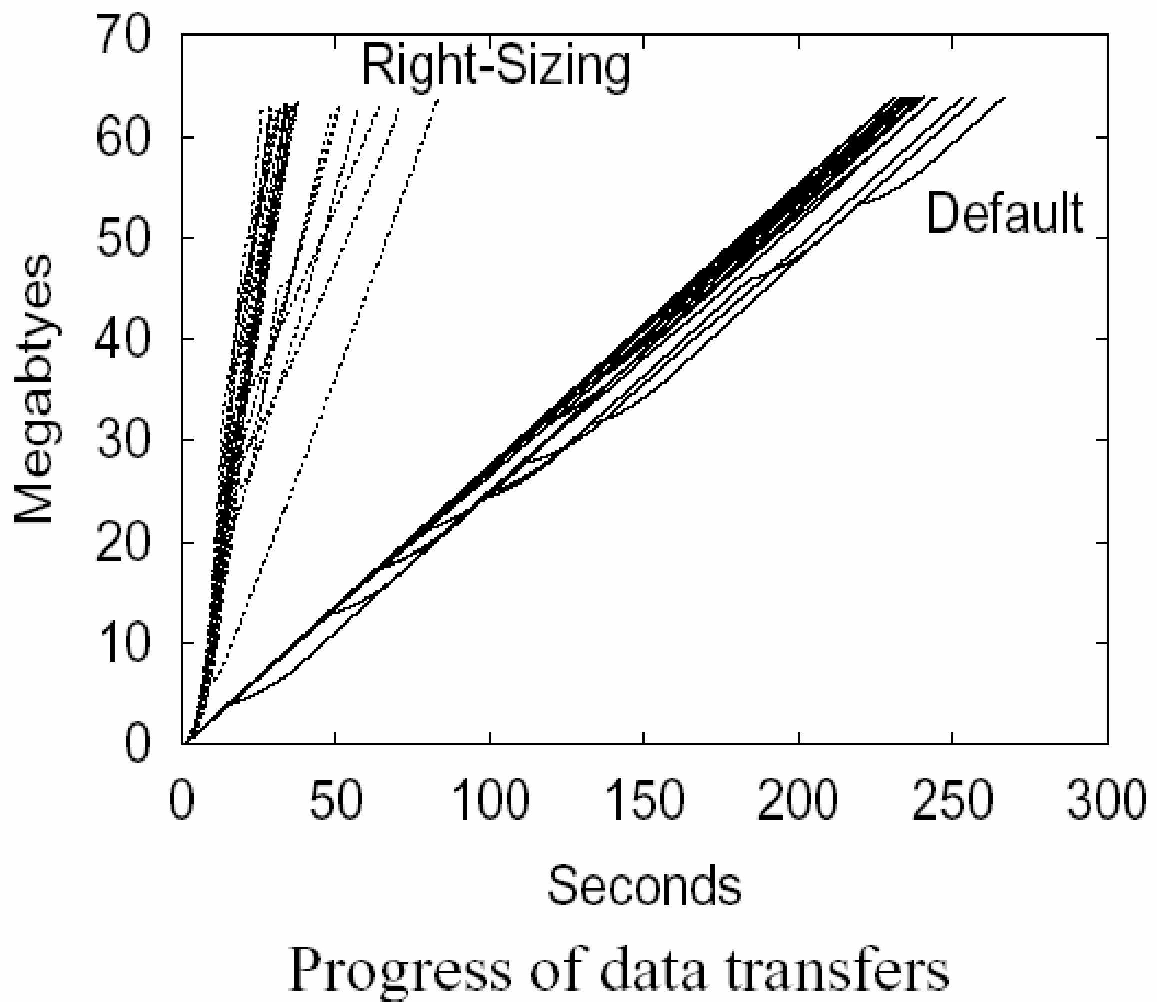

Figure 8: Expected Results for Data Transfers [1]

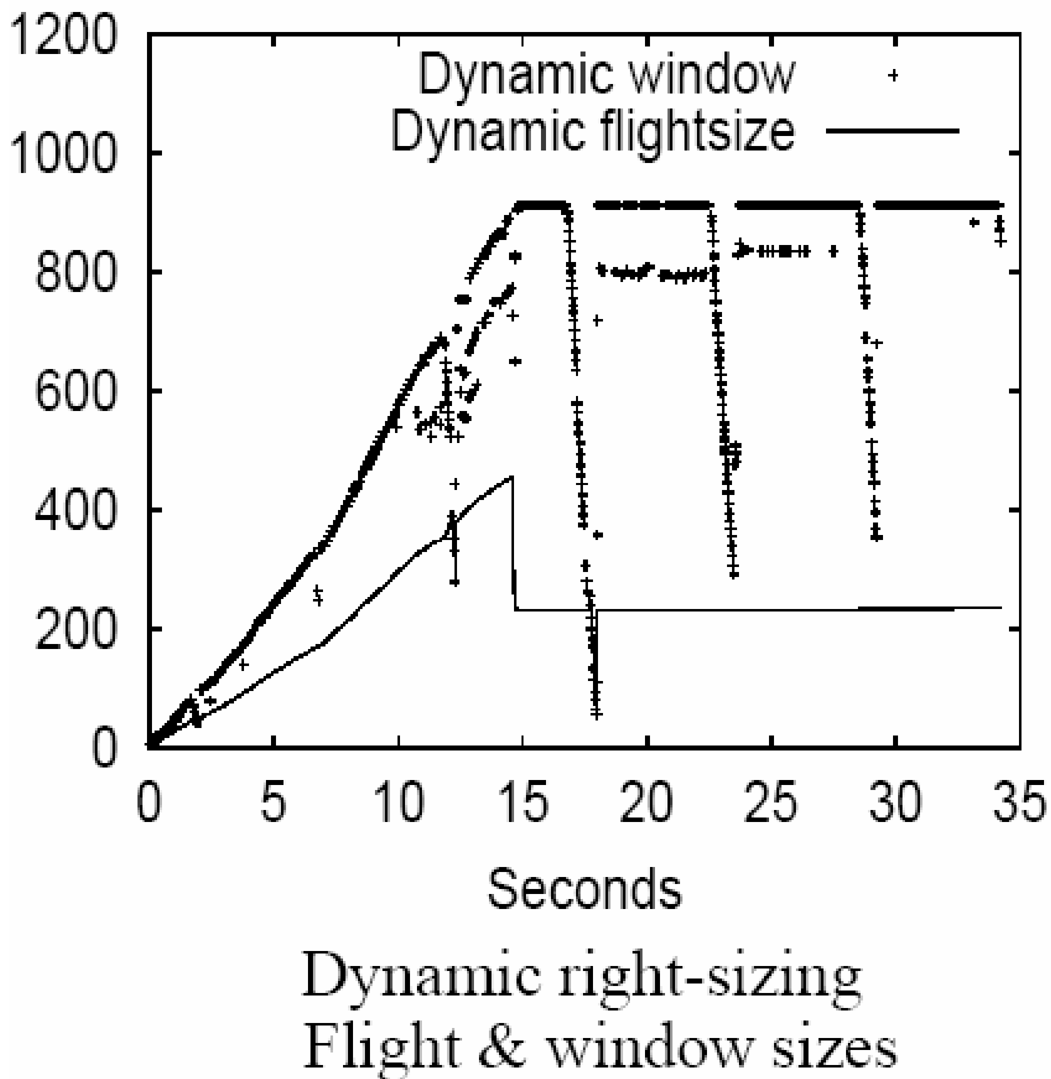

Figure 9: Expected Results for Window Sizes [1]

Figure 8 shows the marked improvement in data transfer rates when dynamic right sizing is implemented. Figure 9 shows the dynamic linear increase of the receive window sizes and the flight sizes - these would have increased proportionally to the congestion window.

Figure 10 shows the results achieved in OPNET. Graph 1 shows that the data transfer successfully occurs with the dynamic right sizing changes in place. Graphs 2 and 3 show the **remote** receive window varying proportionally to the congestion window on the client side and Graphs 4 and 5 show the **remote** receive window varying proportionally to the congestion window on the server side. Without dynamic right sizing in place the receive window is usually a flat line - it's value unchanged as the simulation progresses.

The simulations were unable to show any improvement in data transfer rate because of dynamic right sizing or proportional changes in the flight window size.

Discussion

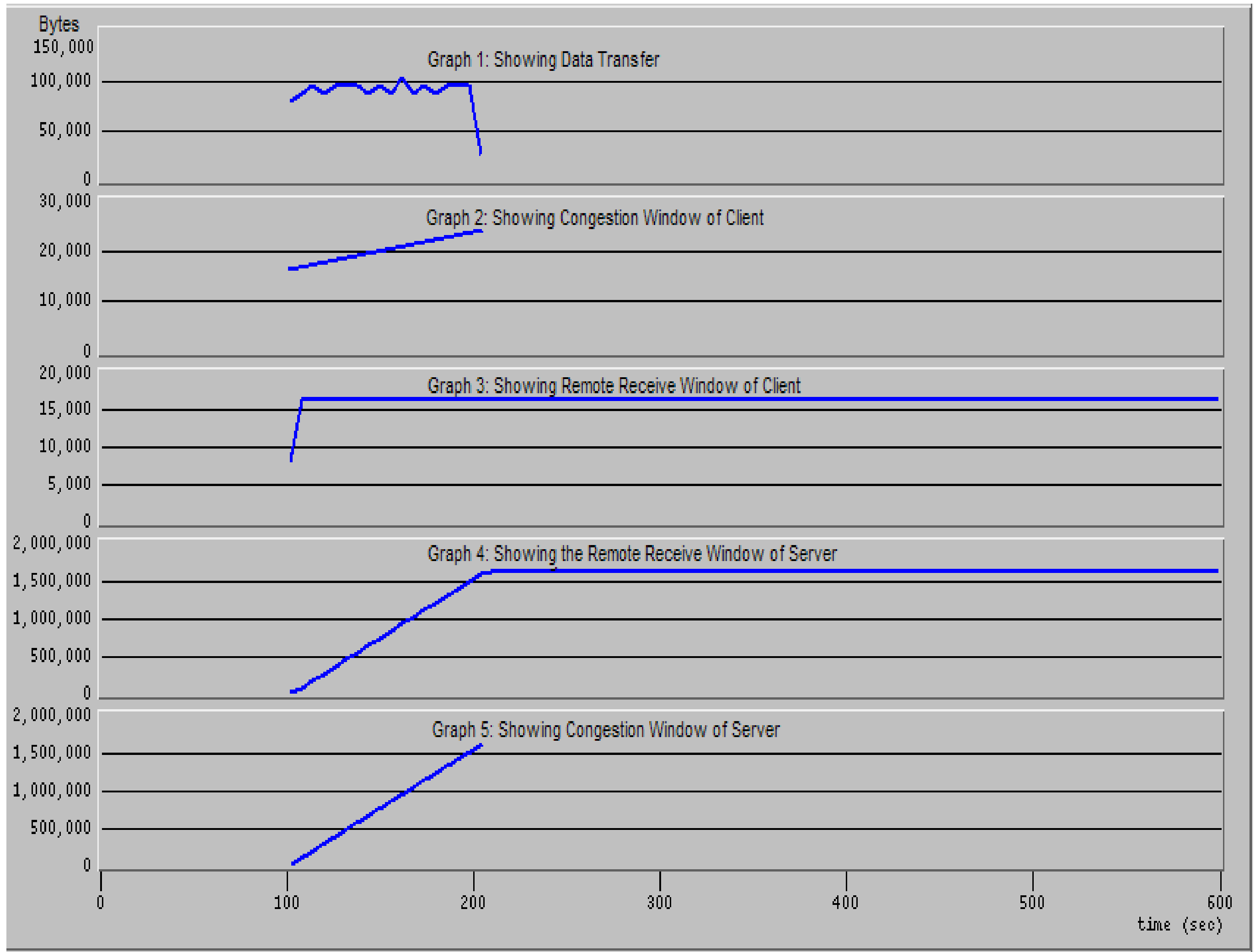

Figure 10: Actual Results

Conclusion and Future Work

The simulations were able to demonstrate the dynamic right sizing concept i.e. the receive window varying proportionally to the congestion window.

However, the simulations were unable to show any improvement in data transfer rate because of dynamic right sizing or proportional changes in the flight window size.

Future work would address these 2 missing results.

Possible reasons for these missing results include the complexity of the TCP protocol and the thoroughness that is required to implement changes to such a protocol. As well, demonstrating data transfer in OPNET may be non-trivial because of the way that the discrete time simulations are performed.

References

REFERENCES

Mike Fisk and Wu-chun Feng, Dynamic Right-Sizing: TCP Flow-ControlAdaptation, in Proceedings of the 14th Annual ACM/IEEE SC2001 Conference,November 2001. http://public.lanl.gov/radiant/pubs/drs/sc2001-poster.pdf

RFC1323 – TCP Window Scale Option

Kurose and Ross – Chapter 3 presentation slides pg 72, 73 Computer Networking: A Top Down Approach Featuring the Internet, 3rd edition. Jim Kurose, Keith Ross Addison-Wesley, July 2004.

Van Jacobson, Congestion Avoidance and Control, in Proceedings,SIGCOMM 1988 Workshop. ACM SIGCOMM, Aug. 1988, pp. 314-329, ACM Press,Stanford, CA.

Jeff Semke, Jamshid Mahdavi, and Matt Mathis., Automatic TCP buffertuning, Computer Communications Review, vol. 28, no. 4, pp. 315-323, Oct.1998.

Jian Lui and Jim Ferguson,"Automatic TCP socket buffer tuning,"inSupercomputing 2000 Research Gems, Nov. 2000, Awarded Best Research Gem ofthe Conference

OPNET Online help, Standard Models User Guide, TCP Model User Guide

Appendix - Code Listing - tcp_conn_v3.pr.c

```
/* Process model C form file: tcp_conn_v3.pr.c */
/* Portions of this file copyright 1992-2004 by OPNET Technologies, Inc. */



/* This variable carries the header into the object file */
const char tcp_conn_v3_pr_c [] = "MIL_3_Tfile_Hdr_ 110A 30A opnet 7 44A287B0 44A287B0 1 asb9884blade1 jaggerna 0 0 none
none 0 0 none 0 0 0 0 0 0 0 0 b56 1
";
#include <string.h>



/* OPNET system definitions */
#include <opnet.h>



/* Header Block */

#include <math.h>
#include <string.h>
#include <ip_addr_v4.h>
#include <oms_dt.h>
#include <tcp_api_v3.h>
#include <tcp_v3.h>
#include <llm_support.h>
#include <oms_pr.h>
#include <ip_qos_support.h>
#include <tcp_seg_sup.h>
#include <tcp_timer.h>
#include <oms_tan.h>
#include <ip_notif_log_support.h>
#include <ip_rte_support.h>

/* Duration of TIME-WAIT timer (2MSL -- i.e., twice maximum */
/* segment lifetime). Most TCP implementations set it to 240.0 */
/* seconds, however for simulation purposes (in order to free */
/* up memory help by the TCP process), we set it to 10 seconds. */
#define TCPC_2MSL 10.0

/* The TCP header uses a 16 bit field to report the receive */
/* window size to the sender. Therefore, the largest window */
/* that can be used is 2**16 -1 = 65K bytes. */
#define TCPC_MAX_WND_SIZE 65535

/* The window scale extension expands the definition of the TCP */
/* window to 32 bits and then uses a scale factor to carry this */
/* 32-bit value in the 16-bit Window field of the TCP header. */
/* The scale factor is carried in a new TCP option, "Window */
/* Scale". This option is sent only in a SYN segment, hence */
/* the window scale is fixed in each direction upon connection */
/* instantiation. */
#define TCPC_WS_MAX_WND_SHIFT 14

/* The maximum window size with the window scaling option ON */
/* The value is calculated as TCPC_MAX_WIN_SIZE * WND_SHIFT */
#define TCPC_MAX_WND_WWS_SIZE 1073725440
```

Appendix - Code Listing -

```
/* The selective acknowledgment (SACK) extension uses the form         */
/* of two new TCP options:                                                                                  */
/*    1. An enabling option, "SACK-permitted", that may be sent        */
/*       in a SYN segment to indicate that the the SACK option  */
/*       may be used once the connection is established.               */
/*    2. The SACK option itself -- sent over an established            */
/*       connection once permission has been given by                  */
/*       SACK-permitted.                                                                     */
/* SACK option is to be used to convey extended acknowledgment         */
/* information over an established connection.  Specifically,    */
/* it is sent by a data receiver to inform the data transmitter      */
/* of non-contiguous blocks of data that have been received and        */
/* queued. Define a case if SACK-permitted option has been             */
/* both sent and received.                                                                    */
#define SACK_PERMITTED                    (sack_permit_sent && sack_permit_rcvd)

/* Specifies whether the system is in recovery phase for SACK.          */
#define SACK_RECOVERY                     (SACK_PERMITTED && tcp_seq_lt (snd_una, scoreboard_ptr->recovery_end))

/* Byte size which each SACK block consumes in the TCP header.        */
#define TCPC_SACK_BLOCK_SIZE                                                8

/* Maximum number of SACK blocks which can be put in the header     */
/* (Note that if Timestamps are implemented, this should become       */
/* a variable.)                                                                                           */
#define TCPC_SACK_MAX_BLOCKS                                                4

/* Interrupt code indicating that a reset has to be sent to the      */
/* the other side when the limit for retransmissions is reached  */
#define TCPC_MAX_RETRANS_REACHED                                            10

/* Retransmissions can be limited in two ways. 1. By placing a*/
/* maximum limit on the number of retransmissions attempted for       */
/* single segment and 2. By placing a max limit on the time for         */
/* which retransmission will be attempted.                                          */
typedef enum
            {
            TcpC_Max_Retrans_Limit_By_Attempts = 0,
            TcpC_Max_Retrans_Limit_By_Interval = 1
            } TcpT_Max_Retrans_Limit_Mode;

/* A record type used by a SACK-receiver to store selectively  */
/* acknowledged blocks in its "scoreboard".                                         */
typedef struct
            {
            TcpT_Seq                    start;                      /* Start of a SACK'd block.
            */
            TcpT_Seq                    end;                        /* End of the SACK'd block (byte after)                 */
            TcpT_Seq                    retran_end;                 /* End of the retransmitted data preceeding  */
                                                                                 /* this SACK'ed block (byte after)
            */
            } TcpT_Scoreboard_Entry;

/* A record type used by the SACK-receiver to hold a list of      */
/* all blocks selectively acknowledged by the data receiver.      */
/* (Blocks are stored in increasing sequence number order.)            */
typedef struct
            {
```

```
	TcpT_Seq		recovery_end;		/* "snd_nxt" at the time at which last	*/
								/* retransmission occured.
		*/
	TcpT_Seq		last_retran_end;	/* End of last block retransmitted.		*/
	Boolean			more_retran;		/* Is more data to be retransmitted? */
	List*			entries;		/* List of selectively ACKed blocks (of	*/
								/* type -- TcpT_Scoreboard_Entry.)
		*/
	} TcpT_Scoreboard;

/* A record type used by the SACK-sender to store SACK'd blocks	*/
/* in its SACK list.								*/
typedef struct
	{
	TcpT_Seq		start;		/* Start of a SACK'ed block.	*/
	TcpT_Seq		end;		/* End of a SACK'ed block.		*/
	} TcpT_Sackblock;

/* A record type used by the SACK-sender to hold a list of all	*/
/* blocks which it will send in the SACK option. The blocks are	*/
/* stored with most recently reported blocks first; very first	*/
/* entry is the segment triggering the current SACK (RFC 2018).	*/
typedef struct
	{
	List*			entries;	/* List of SACK'ed blocks.		*/
	} TcpT_Sacklist;

/* A record type which the SACK-sender places in the SACK		*/
/* option field  in the TCP header to transmit a portion of		*/
/* its SACK list.								*/
typedef struct
	{
	int				size;		/* Number of SACK blocks being transmitted.	*/
	TcpT_Seq*		blocks;		/* Pointer to an array of SACK blocks.		*/
	} TcpT_Sackoption;

/* TcpT_Seg_Record: Record of a received segment. Used to store	*/
/* segments that have arrived out of order until the preceding	*/
/* data has arrived.								*/
typedef struct
	{
	TcpT_Seq		seq;
	Packet*			data_ptr;
	int				push_flag;
	} TcpT_Seg_Record;

/* An event record is used to hold event information in cases	*/
/* (such as self interrupts) where no event record pointer is	*/
/* handed down from the manager. It is reused every time record	*/
/* space is needed.								*/
TcpT_Event		event_record;

/* Global statistics declaration.					*/
extern double		tcp_global_ete_delay;

/* MAXimum and MINimum macros; note that one argument	*/
/* is evaluated twice.							*/
#define MAX(a, b)			((a) > (b) ? (a) : (b))
#define MIN(a, b)			((a) < (b) ? (a) : (b))
```

Appendix - Code Listing -

```
/* Transition condition macros. */
#define OPEN_PASSIVE            ev_ptr->event == TCPC_EV_OPEN_PASSIVE
#define OPEN_ACTIVE             ev_ptr->event == TCPC_EV_OPEN_ACTIVE
#define CLOSE                   ev_ptr->event == TCPC_EV_CLOSE
#define SEND                    ev_ptr->event == TCPC_EV_SEND
#define PASSIVE_RESET           ev_ptr->event == TCPC_EV_PASSIVE_RESET
#define RCV_SYN                 ev_ptr->event == TCPC_EV_RCV_SYN
#define RCV_ACK_OF_SYN          ev_ptr->event == TCPC_EV_RCV_ACK_OF_SYN
#define RCV_SYN_ACK             ev_ptr->event == TCPC_EV_RCV_SYN_ACK
#define RCV_FIN                 ev_ptr->event == TCPC_EV_RCV_FIN
#define RCV_ACK_OF_FIN          ev_ptr->event == TCPC_EV_RCV_ACK_OF_FIN
#define RCV_FIN_ACK_OF_FIN      ev_ptr->event == TCPC_EV_RCV_FIN_ACK_OF_FIN
#define TIMEOUT                 ev_ptr->event == TCPC_EV_TIMEOUT_TIME_WAIT


/* Transition executive macros. */
#define SEND_SYN                tcp_syn_send (TCPC_FLAG_NONE)
#define SEND_ACK                tcp_ack_schedule ()
#define SEND_SYN_ACK            tcp_syn_send (TCPC_FLAG_ACK)
#define SEND_FIN                tcp_fin_schedule ()

/* Special constants used to represent values or absence of values   */
/* in segment manipulation functions.  This makes the function call  */
/* more legible by clarifying the meaning of each argument.          */
#define TCPC_DATA_NONE          (Packet*) OPC_NIL

/* Declarations of FB functions. */
static void             tcp_conn_sv_init ();

static int              tcp_seq_lt (TcpT_Seq n1, TcpT_Seq n2);
static int              tcp_seq_le (TcpT_Seq n1, TcpT_Seq n2);
static int              tcp_seq_gt (TcpT_Seq n1, TcpT_Seq n2);
static int              tcp_seq_ge (TcpT_Seq n1, TcpT_Seq n2);

static int              tcp_seq_check (void);
static int              tcp_rst_check (void);
static int              tcp_syn_check (void);
static int              tcp_ack_check (void);
static int              tcp_fin_check (void);

static TcpT_Event*      tcp_ev_analyze (const char* state_name);

static void             tcp_timeout_retrans (void* PRG_ARG_UNUSED (input_ptr), int
PRG_ARG_UNUSED (code));
static void             tcp_timeout_delay_ack (void* PRG_ARG_UNUSED (input_ptr), int
PRG_ARG_UNUSED (code));
static void             tcp_timeout_persist (void* PRG_ARG_UNUSED (input_ptr), int
PRG_ARG_UNUSED (code));
static void             tcp_retrans_timer_reset (void);
static void             tcp_new_reno_retransmit (void);

static void             tcp_command_send (Packet* pk_ptr, TcpT_Flag flags);
static void             tcp_command_receive (int num_pks);

static void             tcp_cwnd_update (TcpT_Size acked_bytes);
static void             tcp_cwnd_stat_update (void);
static void             tcp_seg_send_delay_stat_record ();
```

Appendix - Code Listing -

```
static void                    tcp_rtt_measurements_update (double measurement_start_time);
static void                    tcp_acked_bytes_processing (void);

static void                    tcp_ecn_processing (void);
static void                    tcp_ecn_request_to_ip (int set_ect_codepoint);

static void                    tcp_seg_send (Packet* data_pk_ptr, TcpT_Seq seq, TcpT_Flag flags);
static void                    tcp_seg_receive (Packet* seg_ptr, TcpT_Flag flags);

static void                    tcp_snd_data_process (void);
static void                    tcp_una_buf_process (Boolean unacked_data_exist);
static void                    tcp_snd_buf_process (Boolean unacked_data_exist);
static TcpT_Size       tcp_snd_data_size (Boolean unacked_data);
static TcpT_Size       tcp_snd_total_data_size (Boolean unacked_data_exist);
static void                    tcp_send_window_update (TcpT_Seq seq_num, TcpT_Seq ack_num, TcpT_Seq
new_snd_wnd);

static void                    tcp_rcv_buf_process (void);
static void                    tcp_ack_schedule (void);
static void                    tcp_seg_and_timer_based_ack_schedule (void);
static void                    tcp_clock_based_ack_schedule (void);

static void                    tcp_syn_send (TcpT_Flag flags);
static void                    tcp_fin_schedule (void);
static void                    tcp_fin_send (void);
static void                    tcp_ev_error (int ev_type, const char* state_name);

static void                    tcp_conn_abort (void);
static void                    tcp_conn_app_notify_open (const char* state);
static void                    tcp_conn_app_notify_conn_close (const char* state, int status);
static void                    tcp_conn_app_notify_received_close (const char* state, int status);
static void                    tcp_conn_app_indication (const char* state, int status);

static void                    tcp_conn_error (const char* msg0, const char* msg1, const char* msg2);
static void                    tcp_conn_warn (const char* msg0, const char* msg1, const char* msg2);

static void                    tcp_lan_handle_get (void);

static void                    tcp_frfr_processing (void);
static void                    tcp_fast_retrans (void);

static void                    tcp_window_scale_option_process (Packet* pkptr);
static void                    tcp_window_scaling_initialize (void);

static void                    tcp_sack_processing (Packet* pkptr);
static void                    tcp_sack_initialize (void);
static void                    tcp_sack_memory_free (void);
static void                    tcp_sack_retransmission (void);
static TcpT_Sackoption*   tcp_sackoption_get (Packet* pkptr);
static void                    tcp_sackoption_set (Packet* pkptr);

static void                    tcp_sacklist_update_newack (TcpT_Seq ack_seq);
static void                    tcp_sacklist_update_block (TcpT_Seq start, TcpT_Seq end);
static void                    tcp_sacklist_clear (void);
static void                    tcp_sacklist_print (void);

static void                    tcp_scoreboard_clear (void);
static void                    tcp_scoreboard_update_newack (TcpT_Seq ack_seq, TcpT_Seq old_snd_una);
static void                    tcp_scoreboard_update_sack (TcpT_Sackoption* new_sackoption_ptr);
```

```
static TcpT_Sackblock*    tcp_scoreboard_find_retransmission (int max_bytes);
static void               tcp_scoreboard_print (void);
static TcpT_Size          tcp_sack_number_sacked_bytes_find ();

static void               tcp_connection_statistics_register (Boolean active_session);
static void               tcp_connection_on_max_retrans_reset (void* input_ptr, int code);
static void               tcp_conn_retrans_timeout_schedule (double);
static void               tcp_retrans_timeout_check_and_schedule ();

extern void               tcp_max_retransmit_limit_reached_log_write (char* log_message);

static Boolean            tcp_conn_app_open_indicate (void);

static void               tcp_ts_info_process (Packet* pk_ptr);
static void               tcp_conn_timestamp_init (Packet* seg_ptr, Boolean update_rtt);
static void               tcp_timeout_ev_cancel (void);
static void               tcp_conn_attr_set_passive_as_disabled ();
static void               tcp_conn_frfr_reset ();

static void               tcp_seg_rcvd_stat_write (void);

static int                tcp_conn_option_size_get (void);
static void               tcp_conn_retrans_stat_write (void);
static void               tcp_conn_rcv_buff_adjust (void);

static TcpT_Size          tcp_conn_mss_auto_assign (InetT_Address rem_addr);

static void               tcp_conn_process_destroy (void);


/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef   BIN
#undef   BOUT
#define  BIN              FIN_LOCAL_FIELD(_op_last_line_passed) = __LINE__ - _op_block_origin;
#define  BOUT   BIN
#define  BINIT  FIN_LOCAL_FIELD(_op_last_line_passed) = 0; _op_block_origin = __LINE__;
#else
#define  BINIT
#endif /* #if !defined (VOSD_NO_FIN) */



/* State variable definitions */
typedef struct
	{
	/* Internal state tracking for FSM */
	FSM_SYS_STATE
	/* State Variables */
	TcpT_Tcb*                 tcb_ptr               ;   /* Represents TCB -- transmission control block.
Shares    */
	                                                        /* information about the connection with tcp_manager.
	   */
	TcpT_Conn_Parameters*             tcp_parameter_ptr             ;       /* Parameter structure passed
from the tcp_manager containing the attributes from the tcp connections */
	TcpT_Event*               ev_ptr                ;   /* Record of current TCP event.      */
	double                    conn_start_time           ;       /* Time at which this connection starts. */
```

```
Evhandle                    ack_evh                 ;  /* Delayed ACK timer.
 */
Evhandle                    retrans_evh             ; /* Retransmission timer.          */
Evhandle                    persist_evh             ; /* Persistence timer.
 */
Evhandle                    time_wait_evh           ;      /* Time-wait timer.
                  */
Sbhandle                    snd_buf                 ;  /* Buffer for data that has not yet been
sent.             */
Sbhandle                    una_buf                 ;  /* Buffer for sent data that has not been
acknowledged.  */
Sbhandle                    rcv_buf                 ;  /* Buffer for received data to be stored
before            */
                                        /* transferring to the socket buffer.
                */
List*                  rcv_rec_list                 ; /* List of records of out-of-order received segments.
 */
TcpT_Size              snd_mss                      ; /* Maximum Segment Size for outgoing segments.
                  */
TcpT_Size              initial_window_size          ;     /* Initial Window Size used during slow-
start.            */
double                 rcv_buf_usage_thresh         ;     /* Threshold used to determine the limit on
the usage      */
                                        /* of receive buffer before transferring segments out   */
                                        /* to the socket buffer.
                              */
int                    sess_svr_id                  ;
Boolean                tcp_conn_info_reg            ;
LlmT_Lan_Handle             my_lanhandle            ;      /* LAN node handle, if this proc-
ess resides within a LAN node.     */
int                    sess_wkstn_id                ;
double                 retrans_rtt                  ;  /* Mean round-trip time.
                    */
double                 retrans_rtt_dev              ;     /* Mean deviation of round-trip time.
                    */
double                 rtt_gain                     ;  /* Gain factor for effect of each error on RTT.
           */
double                 rtt_dev_gain                 ;/* Gain factor for effect of each error on deviat.
 */
double                 rtt_dev_coef                 ;/* Coefficient of deviation in RTO calculation.
           */
int                    rtt_active                   ;  /* Nonzero if RTT measurements are being made.
           */
double                 rtt_base_time                ;     /* Time that the segment being timed was
sent.      */
TcpT_Seq                    rtt_seq                 ;     /* Sequence number of the segment being
timed.     */
double                 rto_min                      ;  /* Minimum duration of retransmission timeout.
           */
double                 rto_max                      ;  /* Maximum duration of retransmission timeout.
                                  */
double                 retrans_rto                  ;  /* Calculated of retransmission timeout (based on
 */
                                        /* received segments.)
                             */
double                 retrans_backoff              ;     /* Exponential backoff factor for retrans-
mission.  */
double                 current_rto                  ; /* Duration of retransmission timeout used for
           */
```

```
                                                    /* scheduling retransmission timeout events.          */
        TcpT_Seq                        snd_una                         ; /* Sequence number of first unacknow-
ledged segment.           */
                                                    /* (i.e., oldest unacknowledged sequence number.)
          */
        TcpT_Seq                        snd_max                         ; /* Maximum send sequence number (equal
to "snd_nxt"              */
                                                    /* when no retransmissions take place.)
                              */
        TcpT_Seq                        snd_nxt                         ; /* Sequence number of next segment to be
sent.                     */
        TcpT_Size               snd_wnd                         ; /* Size of the remote receive window (as advertized
                */
                                                    /* by the receiver.)
                                         */
        TcpT_Seq                        snd_up                          ; /* Sequence number of last byte of urgent
data.                     */
        TcpT_Seq                        snd_wl1                         ; /* Sequence number used for last window
update.                   */
        TcpT_Seq                        snd_wl2                         ; /* Acknowledgement number used for last
window update.    */
        TcpT_Seq                        iss                             ;   /* Initial send sequence number.
                                */
        TcpT_Seq                        snd_fin_seq                     ;/* Sequence number of outgoing FIN seg-
ment.     */
        int                     snd_fin_valid                   ;       /* Nonzero if the CLOSE command has
been issued.      */
        int                     snd_up_valid                    ;       /* Nonzero if SND.UP actually marks ur-
gent data.        */
        TcpT_Seq                        rcv_nxt                         ; /* Sequence number of next expected seg-
ment.     */
        TcpT_Seq                        rcv_up                          ; /* Sequence number of last byte of urgent
data.     */
        TcpT_Seq                        irs                             ;   /* Initial receive sequence number.  */
        TcpT_Seq                        rcv_fin_seq                     ;/* Sequence number of the received FIN
control.  */
        int                     rcv_fin_valid                   ;       /* Nonzero if a FIN control has been re-
ceived.   */
        int                     rcv_up_valid                    ;       /* Nonzero if RCV.UP actually marks ur-
gent data.        */
        TcpT_Seq                        rcv_buf_seq                     ;       /* Sequence number of next seg-
ment to forward to appl.  */
        TcpT_Seq                        seg_seq                         ; /* Sequence number of "current" segment.
          */
        TcpT_Seq                        seg_ack                         ; /* Acknowledgement number of "current"
segment. */
        TcpT_Size               seg_len                         ; /* Width in sequence space of "current" segment.
          */
        TcpT_Size               rcv_buff                        ; /* Total size of receive buffer.        */
        TcpT_Size               rcv_wnd                         ; /* Advertised size of the avaliable receive buffer.
          */
        TcpT_Size               cwnd                            ; /* Current slow-start/congestion window size.
          */
        TcpT_Size               ssthresh                        ; /* Slow-start/congestion window size threshold.
          */
        int                     tcp_trace_active                        ;
        int                     tcp_retransmission_trace_active                 ;
        int                     tcp_extns_trace_active                  ;   /* If enabled in ADDITION to tcp_trace,
will print out                */
```

```
                                        /* information related to Window Scaling, Fast Retransmit-
     */
                                        /* Fast Recovery and Selective Acknowledgement exten-
sions.   */
     TcpT_Nagle                 nagle_support                   ;     /* Nonzero if Nagle SWS avoidance
should be used.   */
     double                     max_ack_delay                   ;     /* Max time to wait before sending a
dataless ACK.   */
     double                     timer_gran                      ; /* Timer Granularity -- timer ticks at which timer-
based    */
                                        /* events like retransmissions occur.
                           */
     double                     persist_timeout                 ;     /* Duration of the persistence timeout.
     */
     Boolean                    karns_algo_enabled              ;     /* Store information whether Karn's algo-
rithm is enabled or not.                                        */
     TcpT_Seq                        max_retrans_seq            ;     /* Store information about the
maximum retransmitted sequence number. This information is  */
                                        /* used for implementing Karn's algorithm used to calculate
retransmission timeout.          */
     int                        syn_rcvd                        ; /* Nonzero if a SYN segment has been received.
     */
     int                        conn_estab                      ;/* Nonzero if the connection is fully established.
     */
     int                        num_pks_req                     ;     /* Number of packets that have been re-
quested by appl.   */
     double                     last_snd_time                   ;     /* Last time that a segment was sent by this
socket.   */
     Boolean                    dup_una_buf_init                ;     /* Segment retransmission related informa-
tion. */
     Sbhandle                        dup_una_buf                ;     /* Buffer used to contain copy of
packets from the   */
                                        /* unacknowledged data budder (una_buf)
                  */
     int                        passive                         ;  /* Retained information about the original connec-
tion specification.  */
                                        /* This information is used to reset the connection parame-
ters before        */
                                        /* returning to LISTEN in the case of an RST from the SYN-
RCVD state.        */
     InetT_Address              passive_rem_addr                ;
     int                        passive_rem_port                ;
     Boolean                    tcp_app_notified_for_conn_closed     ;     /* Declare a flag used to keep
track of sending the            */
                                        /* "CLOSE" indication to the application layer.
      */
     Stathandle                 packet_thru_handle              ;     /* Statistic handle declarations.
                                */
     Stathandle                 byte_thru_handle                ;
     Stathandle                 packet_sec_thru_handle          ;
     Stathandle                 byte_sec_thru_handle            ;
     Stathandle                 tcp_seg_delay_handle            ;
     Stathandle                 tcp_seg_global_delay_handle     ;
     Stathandle                 tcp_delay_handle                ;
     Stathandle                 tcp_global_delay_handle         ;
     Stathandle                 tcp_global_retrans_count_handle ;
     Stathandle                 retrans_count_handle            ;
     TcpT_Delayed_Ack_Scheme         tcp_del_ack_scheme         ;     /* Variables to model the
dataless acknowledgment scheduling.              */
```

```
        int                     tcp_segments_rcvd_without_sending_ack       ;   /* Count of how many segments
have been received without sending */
                                                  /* any acknowledgement.
                                                           */
        Boolean                 wnd_scale_sent                  ;       /* Has a window scaling option been sent
in the SYN? */
        Boolean                 wnd_scale_rcvd                  ;       /* Was the window scaling option set in
the received SYN? */
        int                     snd_scale                     ; /* The factor by which the snd_wnd gets scaled; 0 if
window scaling disabled */
        int                     rcv_scale                     ; /* The factor by which rcv_wnd gets scaled; 0 if
window scaling disabled */
        int                     requested_snd_scale             ;       /* The window scaling factor which the
other host has requested */
        int                     requested_rcv_scale             ;       /* The window scaling factor requested
from the other host */
        int                     window_scaling_enabled            ;     /* Is this process using window scaling
option?   */
        TcpT_Flavor             tcp_flavor                    ; /* Attribute indicates the TCP flavor being used.
The supported flavors are "TCP Tahoe" (Only Fast Retransmit) & "TCP Reno" (Supports both Fast Retransmit and Fast Recovery)
*/
        int                     dup_ack_cnt                   ;         /* Counts the number of completely dupli-
cate acknowledgements      */
                                                  /* received in a row, i.e., no intermediate acks of any sort
        */
                                                  /* have been received, and these ACKS contain no new data,
                 */
                                                  /* window info, etc.
                                                         */
        int                     sack_enabled                  ;         /* Is this process using selective acknowl-
edgment option?   */
        Boolean                 sack_permit_rcvd                ;       /* Is the remote host also using SACK?
        */
        Boolean                 sack_permit_sent                ;       /* Indicates if SACK-permitted flag was
set in the sent SY.          */
        TcpT_Scoreboard*                scoreboard_ptr                  ;       /* Stores information related to
SACK blocks received.      */
        TcpT_Sacklist*          sacklist_ptr                  ; /* Stores information related to SACK blocks sent.
        */
        TcpT_Size               pipe                        ;   /* A variable used in SACK retransmission. Keeps
count */
                                                  /* of approximately how much data is outstanding.     */
        Boolean                 fast_retransmit_occurring         ;     /* A flag which will be true if TCP is cur-
rently retransmitting */
                                                  /* a packet due to Fast Retransmit.                   */
        Pmohandle               scoreboard_entry_pmh              ;     /* A handle for the memory category used
for allocating entries */
                                                  /* in the scoreboard.                          */
        Pmohandle               sackblock_pmh                   ;       /* A handle for the category of memory
used for allocating */
                                                  /* entries in the sacklist.                   */
        Boolean                 tcp_scoreboard_and_sacklist_ptr_valid       ;   /* Flag to determine whether the
scoreboard and the sacklist_ptr  */
                                                  /* been initialized after receiving the ACK of the SEQ or the
ACK */
                                                  /* ACK_SEQ. This prevents the access of the scoreboard_ptr
or the */
                                                  /* sacklist_ptr when have not been allocated                  */
        char                    tcp_conn_id_str [32]            ;       /* Used when tracing on a particular TCP
```

```
connection.                     */
        int                     max_retrans_mode                    ;   /* The retransmission of a segment will be
attempted only for a */
                                            /* limited time. This "time" span can be specified in terms of
*/
                                            /* "attempts" or in terms of "minutes".  This attribute is used
*/
                                            /* to specify if maximum time for retransmission is in terms
of */
                                            /* attempts or minutes.                    */
        int                     max_connect_retrans_attempts        ; /* Specifies the Maximum number of re-
transmission attempts that */
                                            /* will be made for a CONNECT message (SYN).
*/
        double                  max_connect_retrans_interval        ;  /* Specifies the Maximum duration for
which retransmission will */
                                            /* be attempted for a CONNECT message (SYN).
*/
        int                     max_data_retrans_attempts           ;   /* Specifies the Maximum number of re-
transmission attempts that */
                                            /* will be made for User Data.                    */
        double                  max_data_retrans_interval           ;   /* Specifies the Maximum duration for
which retransmission will */
                                            /* be attempted for User data.                    */
        int                     max_retrans_attempts                ;   /* Holds the limit on the retransmission in
number of attempts */
        double                  max_retrans_interval                ;   /* Holds the limit on the retransmission in
duration.        */
        Evhandle                        max_retrans_evh                     ;     /* Holds the handle for the proce-
dure call event that is scheduled to   */
                                            /* close the connection if the limit on the retransmissions of
a single */
                                            /* segment is reached.                         */
        int                     num_retrans_attempts                ;   /* Specifies the total number of retransmis-
sion attempts that have */
                                            /* been made this far.                         */
        double                  transmission_start_time             ;   /* Holds the transmission start time of the
segment at the head           */
                                            /* of the unacknowledged buffer. This value will be reset
each  */
                                            /* time an ackowledgement is received.
                          */
        Boolean                 close_indicated_to_app              ;   /* Used to maintain state on whether a
close confirm has been */
                                            /* sent to the application or not. If a close confirm has not */
                                            /* been sent already, one will be sent from the Termination
*/
                                            /* Block, i.e., when this process is destroyed.          */
        OmsT_Dt_Key             local_dt_key                ;           /* Key used for efficint TCB lookup. Used
by this process */
                                            /* when it is time to send packets or when destroying     */
                                            /* itself.                          */
        Boolean                 fin_segment_sent                    ;     /* Flag used to indicate if a FIN segment
has been sent. */
        TcpT_Seq                        push_seq                            ; /* Sequence number of the last packet to
be pushed in         */
                                            /* the socket buffer (valid only if complete_pkt_rcvd) */
        double                  nagle_limit_time                    ;     /* The last time the TCP process started
refraining from      */
```

```
                                        /* sending data due to limitations given by Nagle's
        */
                                        /* algorithm limitations.
                                    */
                                        /* The variable has a non-zero value only if the data has
      */
                                        /* not been sent, otherwise it is 0.
        */
    double                  snd_wnd_limit_time          ;   /* The last time the TCP process started
refraining from    */
                                        /* sending data due to limitations given by the remote
            */
                                        /* receive window.
                                              */
                                        /* The variable has a non-zero value only if the data has
      */
                                        /* not been sent, otherwise it is 0.
        */
    double                  cwnd_limit_time             ;   /* The last time the TCP process started
refraining from    */
                                        /* sending data due to limitations given by the congestion
      */
                                        /* window.
                                                     */
                                        /* The variable has a non-zero value only if the data has
      */
                                        /* not been sent, otherwise it is 0.
        */
    Boolean                 tcp_app_notified_for_close_rcvd   ;/* Flag to indicate the application layer has
been notified    */
                                        /* of a CLOSE being received from the other side.
              */
    TcpT_Seq                    snd_recover                 ;   /* The sequence number denoting
the end of fast recovery for New     */
                                        /* Reno TCP flavor. This is the maximum sequence number
snd_max at      */
                                        /* the time when fast recovery process started for new Reno.
                    */
    TcpT_Option_TS          timestamp_info              ;   /* TCP Time stamp option information. If
time stamp is not          */
                                        /* supported for teh connection, it is set to
                      */
                                        /* TCP_TIME_STAMP_NOT_USED (OPC_NIL),
                                      */
                                        /* otherwise it contains latest received timestamp value and
      */
                                        /* sequence number of the next expected    segment.
                                                                   */
    int                     conn_supports_ts            ;   /* Specifies whether this connection sup-
ports timestamps. */
    TcpT_Size               max_recvd_wnd               ;   /* Maximum received window for the con-
nection.  */
    TcpT_Ip_Encap_Req_Ici_Info          ip_encap_ici_info           ;   /* Structure for storing
the ip_encap_ind_v6 ici used to send */
                                        /* packets to ip_encap.
                                             */
    } tcp_conn_v3_state;

#define pr_state_ptr                ((tcp_conn_v3_state*) (OP_SIM_CONTEXT_PTR->_op_mod_state_ptr))
```

Appendix - Code Listing -

```
#define tcb_ptr                        pr_state_ptr->tcb_ptr
#define tcp_parameter_ptr                      pr_state_ptr->tcp_parameter_ptr
#define ev_ptr                         pr_state_ptr->ev_ptr
#define conn_start_time                        pr_state_ptr->conn_start_time
#define ack_evh                                pr_state_ptr->ack_evh
#define retrans_evh                            pr_state_ptr->retrans_evh
#define persist_evh                            pr_state_ptr->persist_evh
#define time_wait_evh                          pr_state_ptr->time_wait_evh
#define snd_buf                        pr_state_ptr->snd_buf
#define una_buf                        pr_state_ptr->una_buf
#define rcv_buf                        pr_state_ptr->rcv_buf
#define rcv_rec_list                           pr_state_ptr->rcv_rec_list
#define snd_mss                                pr_state_ptr->snd_mss
#define initial_window_size                    pr_state_ptr->initial_window_size
#define rcv_buf_usage_thresh                   pr_state_ptr->rcv_buf_usage_thresh
#define sess_svr_id                            pr_state_ptr->sess_svr_id
#define tcp_conn_info_reg                      pr_state_ptr->tcp_conn_info_reg
#define my_lanhandle                           pr_state_ptr->my_lanhandle
#define sess_wkstn_id                          pr_state_ptr->sess_wkstn_id
#define retrans_rtt                    pr_state_ptr->retrans_rtt
#define retrans_rtt_dev                        pr_state_ptr->retrans_rtt_dev
#define rtt_gain                       pr_state_ptr->rtt_gain
#define rtt_dev_gain                           pr_state_ptr->rtt_dev_gain
#define rtt_dev_coef                           pr_state_ptr->rtt_dev_coef
#define rtt_active                     pr_state_ptr->rtt_active
#define rtt_base_time                          pr_state_ptr->rtt_base_time
#define rtt_seq                        pr_state_ptr->rtt_seq
#define rto_min                        pr_state_ptr->rto_min
#define rto_max                                pr_state_ptr->rto_max
#define retrans_rto                            pr_state_ptr->retrans_rto
#define retrans_backoff                        pr_state_ptr->retrans_backoff
#define current_rto                            pr_state_ptr->current_rto
#define snd_una                                pr_state_ptr->snd_una
#define snd_max                                pr_state_ptr->snd_max
#define snd_nxt                        pr_state_ptr->snd_nxt
#define snd_wnd                                pr_state_ptr->snd_wnd
#define snd_up                         pr_state_ptr->snd_up
#define snd_wl1                                pr_state_ptr->snd_wl1
#define snd_wl2                                pr_state_ptr->snd_wl2
#define iss                            pr_state_ptr->iss
#define snd_fin_seq                            pr_state_ptr->snd_fin_seq
#define snd_fin_valid                          pr_state_ptr->snd_fin_valid
#define snd_up_valid                           pr_state_ptr->snd_up_valid
#define rcv_nxt                        pr_state_ptr->rcv_nxt
#define rcv_up                         pr_state_ptr->rcv_up
#define irs                            pr_state_ptr->irs
#define rcv_fin_seq                            pr_state_ptr->rcv_fin_seq
#define rcv_fin_valid                          pr_state_ptr->rcv_fin_valid
#define rcv_up_valid                           pr_state_ptr->rcv_up_valid
#define rcv_buf_seq                            pr_state_ptr->rcv_buf_seq
#define seg_seq                        pr_state_ptr->seg_seq
#define seg_ack                        pr_state_ptr->seg_ack
#define seg_len                        pr_state_ptr->seg_len
#define rcv_buff                       pr_state_ptr->rcv_buff
#define rcv_wnd                                pr_state_ptr->rcv_wnd
#define cwnd                           pr_state_ptr->cwnd
#define ssthresh                       pr_state_ptr->ssthresh
#define tcp_trace_active                       pr_state_ptr->tcp_trace_active
#define tcp_retransmission_trace_active                pr_state_ptr->tcp_retransmission_trace_active
```

Appendix - Code Listing -

```
#define tcp_extns_trace_active            pr_state_ptr->tcp_extns_trace_active
#define nagle_support                     pr_state_ptr->nagle_support
#define max_ack_delay                     pr_state_ptr->max_ack_delay
#define timer_gran                        pr_state_ptr->timer_gran
#define persist_timeout                   pr_state_ptr->persist_timeout
#define karns_algo_enabled                pr_state_ptr->karns_algo_enabled
#define max_retrans_seq                   pr_state_ptr->max_retrans_seq
#define syn_rcvd                          pr_state_ptr->syn_rcvd
#define conn_estab                        pr_state_ptr->conn_estab
#define num_pks_req                       pr_state_ptr->num_pks_req
#define last_snd_time                     pr_state_ptr->last_snd_time
#define dup_una_buf_init                  pr_state_ptr->dup_una_buf_init
#define dup_una_buf                       pr_state_ptr->dup_una_buf
#define passive                           pr_state_ptr->passive
#define passive_rem_addr                  pr_state_ptr->passive_rem_addr
#define passive_rem_port                  pr_state_ptr->passive_rem_port
#define tcp_app_notified_for_conn_closed  pr_state_ptr->tcp_app_notified_for_conn_closed
#define packet_thru_handle                pr_state_ptr->packet_thru_handle
#define byte_thru_handle                  pr_state_ptr->byte_thru_handle
#define packet_sec_thru_handle            pr_state_ptr->packet_sec_thru_handle
#define byte_sec_thru_handle              pr_state_ptr->byte_sec_thru_handle
#define tcp_seg_delay_handle              pr_state_ptr->tcp_seg_delay_handle
#define tcp_seg_global_delay_handle       pr_state_ptr->tcp_seg_global_delay_handle
#define tcp_delay_handle                  pr_state_ptr->tcp_delay_handle
#define tcp_global_delay_handle           pr_state_ptr->tcp_global_delay_handle
#define tcp_global_retrans_count_handle   pr_state_ptr->tcp_global_retrans_count_handle
#define retrans_count_handle              pr_state_ptr->retrans_count_handle
#define tcp_del_ack_scheme                pr_state_ptr->tcp_del_ack_scheme
#define tcp_segments_rcvd_without_sending_ack  pr_state_ptr->tcp_segments_rcvd_without_sending_ack
#define wnd_scale_sent                    pr_state_ptr->wnd_scale_sent
#define wnd_scale_rcvd                    pr_state_ptr->wnd_scale_rcvd
#define snd_scale                         pr_state_ptr->snd_scale
#define rcv_scale                         pr_state_ptr->rcv_scale
#define requested_snd_scale               pr_state_ptr->requested_snd_scale
#define requested_rcv_scale               pr_state_ptr->requested_rcv_scale
#define window_scaling_enabled            pr_state_ptr->window_scaling_enabled
#define tcp_flavor                        pr_state_ptr->tcp_flavor
#define dup_ack_cnt                       pr_state_ptr->dup_ack_cnt
#define sack_enabled                      pr_state_ptr->sack_enabled
#define sack_permit_rcvd                  pr_state_ptr->sack_permit_rcvd
#define sack_permit_sent                  pr_state_ptr->sack_permit_sent
#define scoreboard_ptr                    pr_state_ptr->scoreboard_ptr
#define sacklist_ptr                      pr_state_ptr->sacklist_ptr
#define pipe                              pr_state_ptr->pipe
#define fast_retransmit_occurring         pr_state_ptr->fast_retransmit_occurring
#define scoreboard_entry_pmh              pr_state_ptr->scoreboard_entry_pmh
#define sackblock_pmh                     pr_state_ptr->sackblock_pmh
#define tcp_scoreboard_and_sacklist_ptr_valid  pr_state_ptr->tcp_scoreboard_and_sacklist_ptr_valid
#define tcp_conn_id_str                   pr_state_ptr->tcp_conn_id_str
#define max_retrans_mode                  pr_state_ptr->max_retrans_mode
#define max_connect_retrans_attempts      pr_state_ptr->max_connect_retrans_attempts
#define max_connect_retrans_interval      pr_state_ptr->max_connect_retrans_interval
#define max_data_retrans_attempts         pr_state_ptr->max_data_retrans_attempts
#define max_data_retrans_interval         pr_state_ptr->max_data_retrans_interval
#define max_retrans_attempts              pr_state_ptr->max_retrans_attempts
#define max_retrans_interval              pr_state_ptr->max_retrans_interval
#define max_retrans_evh                   pr_state_ptr->max_retrans_evh
#define num_retrans_attempts              pr_state_ptr->num_retrans_attempts
#define transmission_start_time           pr_state_ptr->transmission_start_time
```

Appendix - Code Listing -

```
#define close_indicated_to_app          pr_state_ptr->close_indicated_to_app
#define local_dt_key                    pr_state_ptr->local_dt_key
#define fin_segment_sent                pr_state_ptr->fin_segment_sent
#define push_seq                        pr_state_ptr->push_seq
#define nagle_limit_time                pr_state_ptr->nagle_limit_time
#define snd_wnd_limit_time              pr_state_ptr->snd_wnd_limit_time
#define cwnd_limit_time                 pr_state_ptr->cwnd_limit_time
#define tcp_app_notified_for_close_rcvd         pr_state_ptr->tcp_app_notified_for_close_rcvd
#define snd_recover                     pr_state_ptr->snd_recover
#define timestamp_info                  pr_state_ptr->timestamp_info
#define conn_supports_ts                pr_state_ptr->conn_supports_ts
#define max_recvd_wnd                   pr_state_ptr->max_recvd_wnd
#define ip_encap_ici_info               pr_state_ptr->ip_encap_ici_info

/* These macro definitions will define a local variable called   */
/* "op_sv_ptr" in each function containing a FIN statement.      */
/* This variable points to the state variable data structure,    */
/* and can be used from a C debugger to display their values.    */
#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE
#  define FIN_PREAMBLE_DEC   tcp_conn_v3_state *op_sv_ptr;
#if defined (OPD_PARALLEL)
#  define FIN_PREAMBLE_CODE             \
                op_sv_ptr = ((tcp_conn_v3_state *)(sim_context_ptr->_op_mod_state_ptr));
#else
#  define FIN_PREAMBLE_CODE             op_sv_ptr = pr_state_ptr;
#endif


/* Function Block */


#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ };
#endif
/***** Sequence number comparisons. *****/
/*** Sequence number comparisons are done using a subtly different        **/
/*** mechanisms than the normal "less than" or "greater than" for         **/
/*** either signed or unsigned integers.  The definition we want for   **/
/*** inequalities is: (n2 > n1) if the "distance" from n1 to n2 is     **/
/*** smaller in the positive direction than the negative direction.       **/
/*** Another way of stating this is that we want (n - n0) < (n + n0)   **/
/*** for all n and "small" positive values of n0.                                          **/

#define MAX_SEQ_DIFF ((unsigned int) 1 << 31)          /* 2**31 */

static int
tcp_seq_lt (TcpT_Seq n1, TcpT_Seq n2)
        {
        /** (n1 < n2) if (0 < (n2 - n1) < 2**31) **/
        FIN (tcp_seq_lt (n1, n2));
        FRET (0 < n2 - n1 && n2 - n1 < MAX_SEQ_DIFF);
        }

static int
tcp_seq_le (TcpT_Seq n1, TcpT_Seq n2)
        {
        /** (n1 <= n2) if (0 <= (n2 - n1) < 2**31) **/
        FIN (tcp_seq_le (n1, n2));
```

```
	FRET (n2 - n1 < MAX_SEQ_DIFF && 0 <= n2 - n1);
	}

static int
tcp_seq_gt (TcpT_Seq n1, TcpT_Seq n2)
	{
	/** (n1 > n2) if (0 < (n1 - n2) < 2**31) **/
	FIN (tcp_seq_gt (n1, n2));
	FRET (0 < n1 - n2 && n1 - n2 < MAX_SEQ_DIFF);
	}

static int
tcp_seq_ge (TcpT_Seq n1, TcpT_Seq n2)
	{
	/** (n1 >= n2) if (0 <= (n1 - n2) < 2**31) **/
	FIN (tcp_seq_ge (n1, n2));
	FRET (0 <= n1 - n2 && n1 - n2 < MAX_SEQ_DIFF);
	}

/***** TCP State Variable Initialization *****/
static void
tcp_conn_sv_init ()
	{

	/** Initializes all the state variables based on the 	**/
	/** tcp_parameter_info structure obtained from the 		**/
	/** tcp_manager through parent to child shared mem. 		**/
	FIN (tcp_conn_sv_init ());

	/* Initailize all the state variables of the tcp_conn_v3 */
	/* process from the structure received from the tcp_manager 	*/
	/* through the parent to the child memory. 				*/
	snd_mss 						= 	tcp_parameter_ptr->max_seg_size;
	rcv_buf_usage_thresh 			= 	tcp_parameter_ptr->rcv_buff_thresh;
	tcp_del_ack_scheme 				= 	tcp_parameter_ptr->delayed_ack_scheme;
	max_ack_delay 					= 	tcp_parameter_ptr->maximum_ack_delay;
	window_scaling_enabled 			= 	tcp_parameter_ptr->window_scaling_flag;
	sack_enabled 					= 	tcp_parameter_ptr->sack_options_flag;
	nagle_support 					= 	tcp_parameter_ptr->nagle_flag;
	karns_algo_enabled 				= 	tcp_parameter_ptr->karns_algorithm_flag;
	max_retrans_mode 				= 	tcp_parameter_ptr->mode;
	max_connect_retrans_attempts= 	tcp_parameter_ptr->max_conn_attempts;
	max_data_retrans_attempts 		= 	tcp_parameter_ptr->max_data_attempts;
	max_connect_retrans_interval= 	tcp_parameter_ptr->max_conn_interval;
	max_data_retrans_interval= 		tcp_parameter_ptr->max_data_interval;
	retrans_rto 					= 	tcp_parameter_ptr->initial_rto;
	rto_min 						= 	tcp_parameter_ptr->min_rto;
	rto_max 						= 	tcp_parameter_ptr->max_rto;
	rtt_gain 						= 	tcp_parameter_ptr->gain_in_rtt;
	rtt_dev_gain 					= 	tcp_parameter_ptr->dev_gain;
	rtt_dev_coef 					= 	tcp_parameter_ptr->rtt_dev_coeff;
	timer_gran 						= 	tcp_parameter_ptr->timer_granularity;
	persist_timeout 				= 	tcp_parameter_ptr->persistence_timeout;
	conn_supports_ts 				= tcp_parameter_ptr->timestamp_flag;
	scoreboard_ptr 					= 	OPC_NIL;

	/* Set the receive window size -- it is the amount of 	*/
	/* receive data (in bytes) that can be buffered at one 	*/
	/* time on a connection. The sending host can send only 	*/
```

```
	/* that amount of data before waiting for an ACK and*/
	/* window update from the receiving host. Check if the		*/
	/* receive buffer is set to be computed dynalically.	*/
	if (tcp_parameter_ptr->rcv_buff_size == -1)
		{
		/* When set to "Default", this parameter is set to at	*/
		/* least four times the negotiated MSS, with a maximum		*/
		/* size of 64 KB.								*/
		rcv_buff = (4 * snd_mss <= TCPC_MAX_WND_SIZE) ? 4 * snd_mss : TCPC_MAX_WND_SIZE;
		}
	else
		{
		rcv_buff = tcp_parameter_ptr->rcv_buff_size;
		}

	/* Initialize the flag used to indicate if a FIN has been*/
	/* sent to initiate a connection close sequence.			*/
	fin_segment_sent = OPC_FALSE;

	/* Determine the size of the initial window used during slow	*/
	/* start. Slow-start occurs in as many as three different ways:	*/
	/*  1. start a new connection								*/
	/*  2. restart connection after a long idle period				*/
	/*  3. restart after an idle timeout						*/
	/* The initial window size is only used for cases 1. and 2.		*/
	if (tcp_parameter_ptr->slow_start_initial_count == -1)
		{
		/* Set the initial window wise should be set as defined in		*/
		/* RFC-2414.
*/
		initial_window_size = MIN(4*snd_mss, MAX(2*snd_mss, 4380));
		}
	else
		{
		/* Set the initial window size as a multiple of the MSS.		*/
		initial_window_size = snd_mss * tcp_parameter_ptr->slow_start_initial_count;
		}

	/* Initialize variables used to determine send delay.	*/
	nagle_limit_time = 0.0;
	snd_wnd_limit_time		= 0.0;
	cwnd_limit_time = 0.0;

	/* Flags to monitor indications sent to higher layers.	*/
	tcp_app_notified_for_close_rcvd  = OPC_FALSE;
	tcp_app_notified_for_conn_closed = OPC_FALSE;

	/* Initialize the end of recovery process for New Reno and ECN.		*/
	snd_recover = iss;

	/* Cache the time at which this connection starts.	*/
	conn_start_time = op_sim_time ();

	FOUT;
	}

/***** Segment acceptability tests. *****/
static int
tcp_seq_check (void)
```

```
{
TcpT_Seq                                rcv_wnd_nxt, seg_end;
char                                    msg [128];
TcpT_Seg_Fields*            fd_ptr;

/** Check the sequence number of the arrived segment.           **/
/** This check is used in the following states:                          **/
/** SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2,   **/
/** CLOSE-WAIT, CLOSING, LAST-ACK, and TIME-WAIT.                          **/
/** The tests are defined in RFC 793, p. 69.                      **/
/** Returns nonzero if acceptable, zero otherwise.                **/
FIN (tcp_seq_check ());

/*          Access fields structure from the packet for                */
/*          obtaining sequence number and data length.                       */
op_pk_nfd_access (ev_ptr->pk_ptr, "fields", &fd_ptr);
seg_seq = fd_ptr->seq_num;

/* Record statistics. */
tcp_seg_rcvd_stat_write ();

seg_len = fd_ptr->data_len;

/* Initialize variables used upon retransmissions.        */
fast_retransmit_occurring = OPC_FALSE;

/* rcv_wnd_nxt is the first byte past the valid receive window. */
rcv_wnd_nxt = rcv_nxt + rcv_wnd;

/* Any segment is acceptable if it begins in the receive window. */
if (tcp_seq_le (rcv_nxt, seg_seq) && tcp_seq_lt (seg_seq, rcv_wnd_nxt))
            {
            FRET (1);
            }

if (seg_len == 0)
            {
            /* If the receive window is closed, segments with zero sequence            */
            /* width are acceptable if and only if SEG.SEQ == RCV.NXT.                           */
            if (rcv_wnd == 0 && seg_seq == rcv_nxt)
                        {
                        FRET (1);
                        }
            }
else
            {
            /* Segments with nonzero sequence width are acceptable if       */
            /* any part of the segment overlaps the receive window.                    */

            /* seg_end is the last byte of the segment.                                 */
            seg_end = seg_seq + seg_len - 1;

            if (tcp_seq_le (rcv_nxt, seg_end) && tcp_seq_lt (seg_end, rcv_wnd_nxt))
                        {
                        FRET (1);
                        }

            /* Segments with 1-byte length (persistence check segments)   */
            /* are acceptable if and only if RCV.NXT == SEG.SEQ.                     */
            if ((seg_len == 1 && rcv_nxt == seg_seq) ||
                        (snd_fin_valid && (rcv_nxt == seg_seq + seg_len)))
                        {
                        FRET (1);
                        }
            }
```

```
	/* The packet is acceptable if it is an ACK for a FIN segment.  */
	if (snd_fin_valid && (rcv_nxt == seg_seq + 1))
		{
		FRET (1);
		}

	/* The segment is not acceptable.  Send a reply ACK and drop the segment. */
	tcp_ack_schedule ();

	if (rcv_wnd == 0 && seg_seq == rcv_nxt)
		{
		/* If the receive window is closed, even though no segments will		*/
		/* be acceptable, we still need to be prepared to handle control */
		/* segments (RST and SYN) with valid sequence numbers.  We should */
		/* also handle ACK's with valid sequence numbers, even if the data	*/
		/* in the received segment must be discarded.					*/
		if (tcp_rst_check ())
			if (tcp_syn_check ())
				tcp_ack_check ();
		}

	if (tcp_trace_active)
		{
		sprintf (msg, "SEG.SEQ = %u, SEG.LEN = %u; RCV.NXT = %u, RCV.WND = %u",
			seg_seq, seg_len, rcv_nxt, rcv_wnd);
		op_prg_odb_print_major ("TCP rejecting out-of-window segment:", msg, OPC_NIL);
		}

	FRET (0);
	}

static int
tcp_rst_check (void)
	{
	/** Check the RST bit of the received segment.				**/
	/** This check is used in the following states:				**/
	/** ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT,		**/
	/** CLOSING, LAST-ACK, and TIME-WAIT.  In all of these		**/
	/** states, the response to a received RST is to			**/
	/** immediately abort without sending a response.			**/
	/** Returns 0 if RST is set, 1 otherwise.					**/
	FIN (tcp_rst_check ());

	if (ev_ptr->flags & TCPC_FLAG_RST)
		{
		if (tcp_trace_active)
			op_prg_odb_print_major ("TCP received RST segment; aborting connection.", OPC_NIL);

		ev_ptr->event = TCPC_EV_ABORT_NO_RST;

		/* Update the statistics for the number of reset messages.		*/
		op_stat_write (tcp_parameter_ptr->num_conn_rst_rcvd_stathandle, 1.0);

		FRET (0);
		}
	else
		{
		FRET (1);
		}
	}

static int
tcp_syn_check (void)
	{
```

```
	/** Check the SYN bit of the received segment.                              **/
	/** This check is used in the following states:                             **/
	/** ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT,        **/
	/** CLOSING, LAST-ACK, and TIME-WAIT.  In all of these **/
	/** states, an in-sequence SYN is an invalid packet.    **/
	/** The connection should immediately be aborted.                  **/
	/** Returns 1 if acceptable (no SYN), 0 otherwise.                 **/
	FIN (tcp_syn_check ());

	if (ev_ptr->flags & TCPC_FLAG_SYN)
		{
		if (tcp_trace_active)
			op_prg_odb_print_major ("TCP received unexpected SYN segment; aborting connection.", OPC_NIL);

		ev_ptr->event = TCPC_EV_ABORT_NO_RST;

		FRET (0);
		}
	else
		{
		FRET (1);
		}
	}

static int
tcp_fin_check (void)
	{
	/** Check the FIN bit of the received segment.                              **/
	/** This check is used in the following states:                             **/
	/** ESTABLISHED, SYN_RCVD, FIN-WAIT-2. In all of these           **/
	/** states, an in-sequence FIN is valid           packet.                  **/
	FIN (tcp_fin_check (void));
	if (ev_ptr->flags & TCPC_FLAG_FIN)
		{
		FRET (1);
		}
	else
		{
		FRET (0);
		}
	}

static int
tcp_ack_check (void)
	{
	TcpT_Seg_Fields*			fd_ptr;
	char					str0 [128];
	TcpT_Seq				old_snd_una;
	TcpT_Size				acked_bytes;
	double					current_time;
	double					next_timeout_time;

	/** Check the ACK bit and ACK sequence number of the received segment.       **/
	/** Use the segment information to update congestion window, remote          **/
	/** receive window and to flush the acknowledged data from the retrans         **/
	/** buffer. This check is used in the following states:                                  **/
	/**	ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING,                     **/
	/**		LAST-ACK, and TIME-WAIT.
	**/
	/** Returns 1 if ACK is acceptable, 0 otherwise.                                             **/
	FIN (tcp_ack_check ());

	/* Obtain the current time. */
	current_time = op_sim_time ();
```

```
/* First, make sure that the ACK bit is set.                    */
if (!(ev_ptr->flags & TCPC_FLAG_ACK))
        {
        if (tcp_trace_active)
                op_prg_odb_print_major ("TCP received non-ACK segment; aborting connection.", OPC_NIL);

        FRET (0);
        }

/* The fields structure in the segment contains TCP's header information.           */
op_pk_nfd_access (ev_ptr->pk_ptr, "fields", &fd_ptr);
if (fd_ptr == OPC_NIL)
   tcp_conn_error ("Unable to get the header fields from the received packet.", OPC_NIL, OPC_NIL);

/* Cache the ACK number of the current segment.      */
seg_ack = fd_ptr->ack_num;

/* Record statistics.           */
if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
        op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->rcv_seg_ack_no_stathandle) == OPC_TRUE)
        {
        op_stat_write (tcb_ptr->tcp_conn_stat_ptr->rcv_seg_ack_no_stathandle, seg_ack);
        }

/* Store the value of the current snd_una.      */
old_snd_una = snd_una;

/* Process any ECN-related information.       */
tcp_ecn_processing ();

/* Check for a duplicate acknowledgment. Duplicate ACKs are those which                    */
/* repeat an ACK sequence number already seen in a previous ACK. Hence,       */
/* snd_una has already been advanced up to or past the seg_ack in the pkt           */
/* just received. There are several situations which can cause dup-ACKs:           */
/*  1) the ACK-sender could be in error or the packet might have been              */
/*    delayed and thus received out of order.                                                    */
/*  2) a TCP might repeat an ACK sequence when transmitting new data or a       */
/*    new send window if no new data had been received between the time         */
/*    the new packet was sent and the time the previous ACK was sent.              */
/*  3) another possibility is that TCP is duplicating ACKs because it is   */
/*    receiving packets but it is missing a packet prior to those being       */
/*    received. Thus, it must still send ACKs because new data has arrived          */
/*    but the cumulative ACK cannot be advanced.  This might indicate             */
/*    packet loss, or it might simply indicate packet reordering                      */
/*    somewhere in the network.                                                                                  */
if (tcp_seq_lt (seg_ack, snd_una))
        {
        /* This segment duplicates an ACK older than the most recently received           */
        /* ACK. Count only consecutive receptions of the most recent ACK reset           */
        /* counter, as long as Fast Retransmit has not occurred.                                */
        if (dup_ack_cnt < tcp_parameter_ptr->fr_dup_ack_thresh)
                dup_ack_cnt = 0;

        if (tcp_trace_active || tcp_extns_trace_active)
                op_prg_odb_print_major ("TCP received an old duplicate ACK; ignoring.", OPC_NIL);

        /* Check if the incoming segment contains data.        */
        if (seg_len > 0)
                {
                /* Even though this segment is not in order, accept its*/
                /* data; however, dont process the other details.                */
                FRET (1);
                }
        else
                {
```

```
				FRET (0);
				}
		}

	/* Check if this segment duplicates the most recently received ACK.			*/
	else if (seg_ack == snd_una)
			{
			if ((seg_len != 0) && (conn_supports_ts == TCPC_OPTION_STATUS_ENABLED))
				{
				/* Time stamp is supported and this is not a duplicate ACK.	*/

				/* Process timestamp information carried in the packet.		*/
				tcp_ts_info_process (ev_ptr->pk_ptr);
				}

			/* Does this duplicate ACK contain any new data or a window update?		*/
			if ((seg_len != 0) || (fd_ptr->rcv_win << snd_scale != snd_wnd))
				{
				if ((tcp_trace_active || tcp_extns_trace_active) && dup_ack_cnt != 0)
					{
					op_prg_odb_print_major ("TCP received a duplicate ACK containing new data or a window update.", 
OPC_NIL);
					}

				/* Process SACK-data contained in this packet, if any.		*/
				tcp_sack_processing (ev_ptr->pk_ptr);

				/* Reset the duplicate count, as long as Fast Retransmit has not		*/
				/* already occured.
				*/
				if (dup_ack_cnt < tcp_parameter_ptr->fr_dup_ack_thresh)
					{
					dup_ack_cnt = 0;
					}
				}
			else
				{
			  /* This segment is a true duplicate, i.e., no new data/window			*/
				/* update. Thus, it must indicate packet drop. Now there is				*/
				/* outstanding unacknowledged data which was may have been lost.	*/
				if (tcp_seq_gt (snd_max, snd_una))
					{
					/* Increment the count of "pure" duplicate ACK segment.				*/
					dup_ack_cnt++;

					if (tcp_trace_active || tcp_extns_trace_active)
						{
						sprintf (str0, "TCP received consecutive duplicate ACK number %d.", dup_ack_cnt);
						op_prg_odb_print_major (str0, OPC_NIL);
						}

					/* Process SACK-data contained in this packet, if any.		*/
					tcp_sack_processing (ev_ptr->pk_ptr);

					/* Perform fast-retransmission, if applicable. */
					tcp_frfr_processing ();

					/* Additional packets from snd/una buffers will be sent		*/
					/* if allowed by the congestion control/send window. */
					FRET (1);
					}
				else
					{
					/* Completely duplicate ACK, but there is no outstanding data so simply		*/
					/* discard the packet.
```

```
                                        */
                                if (tcp_trace_active || tcp_extns_trace_active)
                                        {
                                        op_prg_odb_print_major
                                                ("TCP received a duplicate ACK, but there is no outstanding data.", OPC_NIL);
                                        }

                                FRET (0);
                                }
                        }
                }

/* Check for acknowledgment of data not yet sent.      */
else if (tcp_seq_gt (seg_ack, MAX (snd_nxt, snd_max)))
        {
        if (tcp_trace_active)
                op_prg_odb_print_major ("TCP received ACK of data not yet sent; sending ACK.", OPC_NIL);

        /* Write a simulation log message.                              */
        tcp_ack_unsent_log_write ();

        tcp_ack_schedule ();
        FRET (0);
        }

/* Process the newly received acknowledgements.                 */
if (tcp_seq_gt (seg_ack, snd_una))
        {
        /* How many bytes got acked? */
        acked_bytes = seg_ack - snd_una;

        /* If this ACK acknowledges some data, clear them from          */
        /* the unacknowledged buffer.                                                   */
        tcp_acked_bytes_processing ();

        if ((tcp_flavor == TcpC_Flavor_New_Reno) && tcp_seq_lt (snd_una,snd_recover))
                {
                /** The process is in a fast recovery phase for New Reno.        **/

                /* Retransmit the next unacknowledged segment.       */
                tcp_new_reno_retransmit ();
                }

        /* Next, we look for the Timestamp option -- RTT processing  */
        /* is based on how this option is configured.                        */
        if (conn_supports_ts == TCPC_OPTION_STATUS_DISABLED)
                {
                /* Timestamp option is not used. Update the round trip          */
                /* time and RTO timer measurements.                                             */
                tcp_rtt_measurements_update (rtt_base_time);
                }
        else
                {
                /* Timestamp option is used. RTT timers will be updated         */
                /* from within this timestamp info process function.   */
                tcp_ts_info_process (ev_ptr->pk_ptr);
                }

        /* Process SACK-data contained in this packet, if any.          */
        tcp_sack_processing (ev_ptr->pk_ptr);

        /* Since this is an advancing ack, update the scoreboard if        */
        /* SACK option is being used.                                                   */
        if (SACK_PERMITTED && (op_prg_list_size (scoreboard_ptr->entries) != 0))
                {
```

```
				/* SACK is used and scoreboard contains some entries. Update it.		*/
				tcp_scoreboard_update_newack (seg_ack, old_snd_una);

				/* Update the number of selectively ACKed data.		*/
				if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
					op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->sacked_data_stathandle) == OPC_TRUE)
					{
					op_stat_write (tcb_ptr->tcp_conn_stat_ptr->sacked_data_stathandle, (double)
tcp_sack_number_sacked_bytes_find ());
					}
				}

			/* Update the congestion window.	*/
			tcp_cwnd_update (acked_bytes);

			/* This must be the first ACK for this ack_seq number since	*/
			/* it advanced snd_una.											*/
			dup_ack_cnt = 0;

			/* Reset the retransmission timers/handles.	*/
			tcp_retrans_timer_reset ();
			}

		/* Update the remote receive window if the received segment is current. */
		if (tcp_seq_lt (snd_wl1, seg_seq) || (snd_wl1 == seg_seq && tcp_seq_le (snd_wl2, seg_ack)))
			{
			tcp_send_window_update (seg_seq, seg_ack, fd_ptr->rcv_win);

			/* Record advertized receive window statistics. */
			if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
				op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->remote_rcv_win_stathandle) == OPC_TRUE)
				{
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->remote_rcv_win_stathandle, rcv_wnd);
				}

			/* Cancel the persistence timer; this may be set again below. */
			if (op_ev_valid (persist_evh) && op_ev_pending (persist_evh))
				if (op_ev_cancel (persist_evh) == OPC_COMPCODE_FAILURE)
					tcp_conn_warn ("Unable to cancel retransmission timeout.",
						"Spurious retransmission may take place.", OPC_NIL);
			}

		/* If there is now no outstanding unacknowledged data, but the*/
		/* remote receive window size is zero, set the persistence		*/
		/* timeout so we can poll the remote receive window size.		*/
		if (snd_una == snd_max && snd_wnd == 0)
			{
			if (!op_ev_valid (persist_evh) || !op_ev_pending (persist_evh))
				{
				/* Compute the next persistance expiration time.			*/
				next_timeout_time = Tcp_Slowtimo_Next_Timeout_Time_Obtain (current_time, persist_timeout, timer_gran);

				persist_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_persist, OPC_NIL);
				if (op_ev_valid (persist_evh) == OPC_FALSE)
					tcp_conn_warn ("Unable to schedule persistence timeout.",
						"Remote receive window size is closed but will not be polled.", OPC_NIL);
				}
			}

		FRET (1);
		}


static void
tcp_rtt_measurements_update (double measurement_start_time)
```

```
{
int                                     ignore_rto_update = 0;
double                        measured_rtt;
double                        rtt_err;

/** Updates the timers used by TCP -- e.g., RTO, RTT, etc.     **/
FIN (tcp_rtt_measurements_update (measurement_start_time));

/* If this segment acknowledges the sequence number being     */
/* monitored for round trip timing, stop the timer and        */
/* update the measurements.                                                  */
if (rtt_active && tcp_seq_lt (rtt_seq, seg_ack))
        {
        if ((karns_algo_enabled == OPC_TRUE) && (conn_supports_ts == TCPC_OPTION_STATUS_DISABLED))
                {
                /* Karn's algorithm specifies that the round-trip         */
                /* time measurement should not be updated when an  */
                /* acknowledgement arrives for a packet that has       */
                /* already been retransmitted at least once. In                 */
                /* addition, the previous value of RTO needs to be     */
                /* used for the next segment transmission.                      */

                /* Determine if the received acknowledgement is for  */
                /* a retransmitted packet or not. If the sequence        */
                /* number of the segment for which RTT is recorded  */
                /* is less than the maximum retransmission sequence  */
                /* number (i.e., the sequence number of the last        */
                /* retransmitted byte), then it is a case that the          */
                /* received acknowledgement is for an earlier                   */
                /* retransmitted packet. Hence, RTO should not be     */
                /* recalculated/updated.                                                 */
                ignore_rto_update = tcp_seq_le (rtt_seq, max_retrans_seq);
                }

        /* Reset the variable to indicate that a segment is        */
        /* being timed for round-trip time measurement.                */
        /* However, when the timestamp option is used, then */
        /* this is not required.                                                  */
        if (conn_supports_ts == TCPC_OPTION_STATUS_DISABLED)
                rtt_active = 0;

        /* Compute the "measured RTT" for the currently monitored segment.             */
        /* The RTT is computed based on the timer granularity usage. If timer   */
        /* is not used, then it is set to be the exact time difference from          */
        /* the current time to the time from which the segment was timed;           */
        /* otherwise, it is computed based on the timer ticks elapsed during        */
        /* this interval.
        */
        measured_rtt = Tcp_Slowtimo_Elapsed_Time_Obtain (op_sim_time (), measurement_start_time, timer_gran);

        /* Round-trip time measurements are made using the method   */
        /* described in [Jacobson 1988].  Compute the smoothed RTO.             */
        rtt_err = measured_rtt - retrans_rtt;
        retrans_rtt += rtt_gain * rtt_err;
        retrans_rtt_dev += rtt_dev_gain * (fabs (rtt_err) - retrans_rtt_dev);
        retrans_rto = retrans_rtt + rtt_dev_coef * retrans_rtt_dev;

        /* Restrict RTO to the specified limits.          */
        if (retrans_rto < rto_min)
                retrans_rto = rto_min;
        if (retrans_rto > rto_max)
                retrans_rto = rto_max;

        /* Ignore updating the current retransmission                    */
        /* timeout value, if Karn's algorithm determines         */
```

```
		/* it to be ingored. Otherwise, update its value to	*/
		/* the newly calculated retransmitted timeout value	*/
		if ((ignore_rto_update == 0) || (conn_supports_ts == TCPC_OPTION_STATUS_ENABLED))
			{
			/* Update the current retransmission timeout.		*/
			current_rto = retrans_rto;

			/* Record the RTO statistic.				*/
			if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
				op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->rto_stathandle) == OPC_TRUE)
				{
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->rto_stathandle, current_rto);
				}
			}

		/* Record the mean RTT and mean RTT deviation.		*/
		if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL)
			{
			if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->mean_seg_rtt_stathandle) == OPC_TRUE)
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->mean_seg_rtt_stathandle, retrans_rtt);

			if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->mean_seg_rtt_dev_stathandle) == OPC_TRUE)
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->mean_seg_rtt_dev_stathandle, retrans_rtt_dev);
			}
		}

	FOUT;
	}


static void
tcp_acked_bytes_processing (void)
	{
	TcpT_Size				acked_bytes;
	char					msg [128];

	/** This function clears data acknowledged by the current ACK.	*/
	FIN (tcp_acked_bytes_processing (void));

	/* The ACK sequence number is acceptable. Determine	*/
	/* the number of bytes which the segment ACKs.		*/
	acked_bytes = seg_ack - snd_una;

	/* Print a diagnostic message if tracing is on.			*/
	if (tcp_trace_active)
		{
		sprintf (msg, "Number of ACKed bytes = %d", acked_bytes);
		op_prg_odb_print_minor (msg, OPC_NIL);
		}

	/* Check for ACK of FIN. */
	if (fin_segment_sent && tcp_seq_gt (seg_ack, snd_fin_seq))
		{
		if (acked_bytes > 1)
			{
			/* The number of bytes to be flushed from the retransmission	*/
			/* buffer is the same as the number of acknowledged bytes.		*/
			if (op_sar_srcbuf_bits_flush (una_buf, 0.0, 8.0 * (acked_bytes - 1)) == OPC_COMPCODE_FAILURE)
				{
				tcp_conn_warn ("Unable to flush acknowledged data up to FIN.",
					"Data may be inappropriately retransmitted.", OPC_NIL);
				}
			}

		/* Set the largest unacknowledged sequence number to the		*/
```

```
		/* acknowledgment field.								*/
		snd_una = seg_ack;

		/* Note that this segment may actually be the FIN from the		*/
		/* remote side, but we can't check right now, since we can't	*/
		/* actually process the received FIN until we've made sure that*/
		/* all preceding segments have also been received.			*/
		ev_ptr->event = TCPC_EV_RCV_ACK_OF_FIN;
		}

	/* Flush acknowledged segments from the retransmission buffer. */
	if (tcp_seq_gt (seg_ack, snd_una))
		{
		/* Check if there is something in the retransmission buffer. It	*/
		/* may not contain any data if the connection is being CLOSEd		*/
		/* before the 3-way handshake is complete.					*/
		if (op_sar_buf_size (una_buf) > 0 &&
			op_sar_srcbuf_bits_flush (una_buf, 0.0, 8.0 * acked_bytes) == OPC_COMPCODE_FAILURE)
			{
			tcp_conn_warn ("Unable to flush acknowledged data from UNA buffer.",
				"Data may be inappropriately retransmitted.", OPC_NIL);
			}
		}

	/* Set the largest unacknowledged sequence number to the		*/
	/* acknowledgment field.								*/
	snd_una = seg_ack;

	/* Check for the case if we need to update the SND.NXT variable		*/
	/* due to reception of an ACK for a segment sent previously,	*/
	/* not yet retransmitted after detecting a retransmission.		*/
	if (tcp_seq_lt (snd_nxt, snd_una))
		{
		snd_nxt = snd_una;
		}

	/* Write the number of unacknowledged data.		*/
	if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
		op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle) == OPC_TRUE)
		{
		op_stat_write (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle, (double) snd_max - snd_una);
		}

	/* If the segment contained acknowledgment for all the data in */
	/* the unacknowledged data buffer, reset the sequence number		*/
	/* of next segment to be sent to be equal to the maximum send */
	/* sequence number.									*/
	if (tcp_seq_le (snd_nxt, snd_una) && (snd_max == snd_una))
		{
		snd_nxt = snd_max;

		/* Print a diagnostic message if tracing is on.			*/
		if (tcp_trace_active)
			{
			sprintf (msg, "Setting SND.NXT to = %u", snd_nxt);
			op_prg_odb_print_minor (msg, OPC_NIL);
			}
		}

	/* Check to see if all urgent outgoing data (if any) has been	*/
	/* acknowledged.									*/
	if (snd_up_valid)
		{
		if (tcp_seq_gt (seg_ack, snd_up))
			snd_up_valid = 0;
```

```
		}

	FOUT;
	}


static void
tcp_cwnd_update (TcpT_Size acked_bytes)
	{
	TcpT_Size					cwnd_old;

	/** Updates the congestion window maintained by this TCP connection	**/
	/** Note that while sending data, a TCP conenction has to honor the	**/
	/** the limit of sending data eaqual to MIN (cwnd, rcv_adv_win).	**/
	FIN (tcp_cwnd_update ());

	if (tcp_flavor == TcpC_Flavor_New_Reno)
		{
		/* The TCP flavor is New Reno.	*/
		if (tcp_seq_lt (snd_una, snd_recover))
			{
			/* The process is in a recovery phase for New Reno.	*/

			/* Deflate the congestion window by the number	*/
			/* of ACKed data and add back one MSS.			*/
			cwnd = cwnd - acked_bytes + snd_mss;

			/* Record the congestion window statistics.	*/
			tcp_cwnd_stat_update ();

			FOUT;
			}

		/* The second condition below if to make sure that we perform*/
		/* the recovery only once within a given window.			*/
		else if (tcp_seq_ge (snd_una, snd_recover) &&
			tcp_seq_lt ((snd_una - acked_bytes), snd_recover))
			{
			/* This is the first time the process got	*/
			/* out of recovery process for New Reno.	*/

			/* Reset the congestion window.	*/
			cwnd = MIN(ssthresh, snd_max - snd_una + snd_mss);
			}
		}

	/* If this function is called for the first ACK for new data after */
	/* Fast Retransmit and if Fast Recovery is also used then, end the	*/
	/* recovery phase, and set cwnd back to ssthresh				*/
	if (tcp_flavor == TcpC_Flavor_Reno &&
		dup_ack_cnt >= tcp_parameter_ptr->fr_dup_ack_thresh)
		{
		cwnd = ssthresh;
		}

	/* Check if the incoming segment has the ECE flag set.	*/
	if (ev_ptr->flags & TCPC_FLAG_ECE)
		{
		/* This is an ECN-Echo ACK packet. RFC-3168 states	*/
		/* that the sending TCP SHOULD NOT increase "cwnd"	*/
		/* in response to the receipt of an ECN-Echo ACK.	*/
		/* Do nothing.								*/
		}
	else
		{
```

```
		/* Update the congestion window (per desciption on page 18	*/
		/* of Jacobson [1988] or page 310 in TCP/IP Illustrated		*/
		/* [Vol. I] by W. Richard Stevens).				*/
		if (cwnd < ssthresh)
			{
			/* If we're still doing slow-start, then open the		*/
			/* window exponentially. Each time an ACK is received,	*/
			/* increase the congestion window by one. Note that		*/
			/* with this approach the growth in window size may not	*/
			/* be exactly exponential because the receiver may		*/
			/* delay sending ACKs (i.e., it may send a single ACK		*/
			/* for more than one data packet received. In other		*/
			/* words, the number of acknowledged bytes may be more	*/
			/* than one MSS.) Refer RFC-2001 for more details.		*/
			cwnd += snd_mss;

			/* Limit the value of the congestion window to the slow	*/
			/* start threshold value.						*/
			cwnd = (cwnd < ssthresh) ? cwnd : ssthresh;
			}
		else
			{
			/* Otherwise perform congestion avoidance increment		*/
			/* by-one. However, do not start congestion avoidance,	*/
			/* if we fast recovery has not been completed for SACK.	*/
			if (!((SACK_PERMITTED) && (tcp_seq_gt (scoreboard_ptr->recovery_end, seg_ack))))
				{
				/* Store the value of congestion window.	*/
				cwnd_old = cwnd;

				/* Perform increment based on the fact that cwnd	*/
				/* should increase by 1 MSS per round trip time.	*/
				cwnd += snd_mss * snd_mss / cwnd;

				/* Make sure that congestion window was increased. It might happen that,	*/
				/* if the congestion window is sufficiently large, and the integer		*/
				/* arithmetic is used, the above formula will cease to increase cwnd.	*/
				/* RFC 2581 (TCP Congestion Control) states that in that case the cwnd	*/
				/* be incremented by 1 byte. Thus if we did not increase the congestion	*/
				/* window using the equation above, increase it by 1 byte.			*/
				if (cwnd_old == cwnd)
					{
					cwnd += 1;
					}
				}
			}
		}

	/* Record the congestion window statistics.	*/
	tcp_cwnd_stat_update ();

	FOUT;
	}


static void
tcp_ecn_processing (void)
	{
	TcpT_Size				current_cwnd;

	/** Processes the arrival of an ECN segment -- segments with	**/
	/** ECE or CWR bit set. Section 6.1.2 of RFC-3168 states that		**/
	/** TCP should not react to congestion indications more than	**/
	/** once every window of data (or more loosely, more than once	**/
	/** every round-trip time). That is, a TCP sender's congestion	**/
```

```
/** window should be reduced only once in response to a series          **/
/** of dropped and/or ECE packets from a single window of data.          **/
FIN (tcp_ecn_processing ());

/* Check if the incoming segment has the ECE flag set.          */
if (ev_ptr->flags & TCPC_FLAG_ECE)
        {
        /* This is an ECN-Echo ACK packet.          */
        if (tcp_trace_active || op_prg_odb_ltrace_active ("ecn"))
                op_prg_odb_print_major ("TCP received a segment with ECE flag set.", OPC_NIL);

        /* RFC-3168 states that "TCP should not react to congestion     */
        /* indications more than once every window of data (or more     */
        /* loosely, more than once every round-trip time). If the       */
        /* process is not in a recovery phase, start it.                */
        if (tcp_seq_gt (snd_una, snd_recover))
                {
                /** The last recovery phase is over (or there has been none).    **/

                /* Print diagnostic message, if enabled.        */
                if (tcp_trace_active || op_prg_odb_ltrace_active ("ecn"))
                        {
                        op_prg_odb_print_minor (
                                "Reducing cwnd and ssthresh to react to this congestion indication.", OPC_NIL);
                        }

                /* This ECN recovery process will end once all unACKed data             */
                /* (up to snd_max) are ACKed. Store the current snd_max value.          */
                snd_recover = snd_max;

                /* The indication of congestion should be treated just as          */
                /* a congestion loss in non- ECN-Capable TCP. That is, the         */
                /* TCP source halves the congestion window "cwnd" and                       */
                /* reduces the slow start threshold "ssthresh".                             */
                current_cwnd = cwnd;
                cwnd = cwnd / 2;
                if (cwnd < snd_mss)
                        cwnd = snd_mss;

                /* Set the slow-start threshold to half the current window         */
                /* size (but at least two segments).                                                */
                ssthresh = (snd_max - snd_una) / 2;

                if (ssthresh <= 2.0*snd_mss)
                        ssthresh = 2.0*snd_mss;

                /* Since we have processed an ECE flag set by the peer TCP,   */
                /* set CWR flag (indicating that congestion window has been   */
                /* redcued) in the next outgoing segment.                                   */
                tcb_ptr->ecn_status |= TcpC_Ecn_Send_CWR;
                }
        else
                {
                if (tcp_trace_active || op_prg_odb_ltrace_active ("ecn"))
                        {
                        op_prg_odb_print_minor (
                                "Ignoring this flag as this TCP process has already",
                                "reacted to another congestion indication within the",
                                "current round-trip time.", OPC_NIL);
                        }
                }
        }

/* Check if the incoming segment has the CWR flag set.          */
if (ev_ptr->flags & TCPC_FLAG_CWR)
```

```
				{
				/* The remote side has responded to an ECE-echo flag set by	*/
				/* this TCP connection earlier. It can now stop setting		*/
				/* the ECN-Echo flag in all the ACK packets it sends.		*/
				tcb_ptr->ecn_status ^= TcpC_Ecn_Send_ECE;
				}

	FOUT;
	}


static void
tcp_frfr_processing (void)
	{
	TcpT_Seg_Fields*		fd_ptr;

	/** Performs Fast Recovery, Fast Retransmit as described in RFC 2001. **/
	FIN (tcp_frfr_processing (void));

	/* The fields structure in the segment contains TCP's header information.		*/
	op_pk_nfd_access (ev_ptr->pk_ptr, "fields", &fd_ptr);

	/* If the FR/FR threshold has been reached, retransmit the packet & adjust		*/
	/* congestion window statistics accordingly.  Since the threshold has been		*/
	/* reached, it is deemed safe to assume that a packet has been lost and not		*/
	/* just that packets are arriving to the ACK-sender out of order.				*/
	if ((tcp_flavor !=  TcpC_Flavor_Basic) && (dup_ack_cnt == tcp_parameter_ptr->fr_dup_ack_thresh))
		{
		/* If doing SACK, mark where the recovery/retransmission phase ends. */
		if (SACK_PERMITTED)
		   {
		   if (op_prg_list_size (scoreboard_ptr->entries) == 0)
				{
				if (tcp_trace_active)
					{
					op_prg_odb_print_major (
						"Performing Fast Retransmit, but no SACK Options have been received.",
						"Retransmission will proceed as if SACK were not enabled.", OPC_NIL);
					}
				}

			if (tcp_seq_le (scoreboard_ptr->recovery_end, seg_ack))
				{
				/* Previous (if any) selective fast retransmit has been completed.		*/
				/* Start a new one and initialize scoreboard variables.				*/

				if (fin_segment_sent == OPC_FALSE)
					{
					scoreboard_ptr->recovery_end = snd_max;
					}
				else
					{
					/* FIN has been sent. This meand that the last seguence		*/
					/* number that needs to be ACKed so that the system can		*/
					/* exit fast recovery is one byte less than the maximum */
					/* sequence number (since FIN consumes 1 byte).				*/
					scoreboard_ptr->recovery_end = snd_max - 1;
					}
				scoreboard_ptr->last_retran_end = snd_una;

				/* Perform "fast" retransmission.	*/
				fast_retransmit_occurring = OPC_TRUE;

				tcp_fast_retrans ();
				}
```

```
				else
					{
					/* Previous selective acknowledgemnet has not finished. Do not		*/
					/* re-start SACK, but rather continue in the previous one. Any */
					/* missing data will be re-tranmsitted using previously started  */
					/* selective retransmission.
	*/
					fast_retransmit_occurring = OPC_FALSE;
					}
				}
			else
				{
				/* SACK is not enabled; however, perform "fast" retransmission.			*/
				fast_retransmit_occurring = OPC_TRUE;

				/* Do not perform fast retransmission if New Reno is used and we		*/
				/* are in a recovery phase.
	*/
				if (tcp_flavor ==  TcpC_Flavor_New_Reno &&
					tcp_seq_lt (seg_ack, snd_recover))
					{
					/* Do not retransmit the packet, only add one MSS to cwnd	*/
					/* The next segment will be retransmitted later when		*/
					/* tcp_una_buf_process () procedure will be later called.		*/
					cwnd += snd_mss;
					}
				else
					{
					/** Start recovery process. **/

					/* Perform fast retransmission.							*/
					tcp_fast_retrans ();
					}
				}
			}

	/* Inflate the congestion window once for each additional packet assumed to	*/
	/* be stored by receiver only if both Fast Retransmit & Recovery is enabled	*/
	/* and we aren't doing SACK-based recovery.		(In SACK recovery, pipe will	*/
	/* get decremented instead when the scoreboard is updated.)				*/
	else if ((((tcp_flavor == TcpC_Flavor_Reno) || (tcp_flavor == TcpC_Flavor_New_Reno)) &&
		(dup_ack_cnt > tcp_parameter_ptr->fr_dup_ack_thresh)) &&
		((SACK_RECOVERY) == 0))
		{
		/* Update the congestion window if Fast Recovery is also used.		*/
		/* TCP Reno and New Reno will support Fast Recovery.				*/
		cwnd += snd_mss;
		}

	/* Update the congestion window statistic.			*/
	/* Record the congestion window statistics.	*/
	tcp_cwnd_stat_update ();

	FOUT;
	}

static void
tcp_sack_processing (Packet* pkptr)
	{
	TcpT_Sackoption*		new_sacklist_ptr;/* temp storage for old value of snd_nxt. */

	FIN (tcp_sack_processing (pkptr));

	/* Check if SACK is permitted and process further only if SACK		*/
	/* option is et in the incoming segment.						*/
```

```
	if (SACK_PERMITTED && op_pk_nfd_is_set (pkptr, "SACK Option"))
		{
		/* Create a list of SACK blocks received in the incoming segment.*/
		new_sacklist_ptr = tcp_sackoption_get (pkptr);
		if (new_sacklist_ptr == OPC_NIL)
			{
			tcp_conn_warn ("Unable to process the SACK Option in the received packet.", OPC_NIL, OPC_NIL);
			}
		else
			{
			/* Update the scoreboard based on the above SACK list.		*/
			tcp_scoreboard_update_sack (new_sacklist_ptr);

			/* Update the number of selectively ACKed data.		*/
			if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
				op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->sacked_data_stathandle) == OPC_TRUE)
				{
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->sacked_data_stathandle, (double)
tcp_sack_number_sacked_bytes_find ());
				}
			}
		}

	FOUT;
	}


static void
tcp_retrans_timer_reset (void)
	{
	double			next_timeout_time;

	/** Reset the retransmission timer	**/
	FIN (tcp_retrans_timer_reset(void));

	if ((op_ev_valid (retrans_evh) && op_ev_pending (retrans_evh)) ||
	   (op_ev_valid (max_retrans_evh) && op_ev_pending (max_retrans_evh)))
		{
		/* Reset the retransmission backoff factor.					*/
		retrans_backoff = 1;

		/* Also clear the retransmission attempt count					*/
		num_retrans_attempts = 0;
		transmission_start_time = OPC_DBL_INFINITY;

		/* Cancel the old timeout event.							*/
		if (op_ev_valid (retrans_evh))
			{
			if (op_ev_cancel (retrans_evh) == OPC_COMPCODE_FAILURE)
				tcp_conn_warn ("Unable to cancel old retransmission timeout.",
					"Spurious retransmission may take place.", OPC_NIL);
			}
		else
			{
			if (op_ev_cancel (max_retrans_evh) == OPC_COMPCODE_FAILURE)
				tcp_conn_warn ("Unable to cancel old retransmission timeout.",
					"Spurious resets might be sent out.", OPC_NIL);
			}
		}

	/* If there is still data in the buffer or if a FIN is not being		*/
	/* timed, reset the retransmission timeout.  The best choice for */
	/* this timeout would probably be (retrans_rto) + (time the data at		*/
	/* the head of the buffer was last	sent), but we don't have a		*/
	/* graceful way to determine the second term above. So we use the		*/
```

```
	/* current time + current_rto.								*/
	if (((op_sar_buf_size (una_buf) > 0) || ((snd_fin_valid) && tcp_seq_lt (snd_una, snd_max)))
		&& (op_ev_valid (retrans_evh) == OPC_FALSE))
		{
		/* Compute the next RTO expiration time.	*/
		next_timeout_time = Tcp_Slowtimo_Next_Timeout_Time_Obtain (op_sim_time (), current_rto, timer_gran);

		/* Schedule the retransmission timeout.			*/
		retrans_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_retrans, OPC_NIL);

		if (op_ev_valid (retrans_evh) == OPC_FALSE)
			tcp_conn_warn ("Unable to schedule retransmission timeout.",
				"No retransmission will take place.", OPC_NIL);
		}

	FOUT;
	}


static void
tcp_cwnd_stat_update (void)
	{
	/** Updates the congestion window statistic. **/
	FIN (tcp_cwnd_stat_update ());

	if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
		op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->cwnd_size_stathandle) == OPC_TRUE)
		{
		op_stat_write (tcb_ptr->tcp_conn_stat_ptr->cwnd_size_stathandle, (double) cwnd);
		}

	FOUT;
	}


/***** Event processing. *****/

static TcpT_Event*
tcp_ev_analyze (const char* state_name)
	{
	Boolean				conn_id_trace_active = OPC_FALSE;
	TcpT_Event*			evptr = OPC_NIL;

	/** Initialize variables for this invocation. **/
	FIN (tcp_ev_analyze (state_name));

	/* Record the state name. */
	strcpy (tcb_ptr->state_name, state_name);

	/* Check for active ODB trace. */
	if (op_sim_debug () == OPC_TRUE)
		{
		conn_id_trace_active = op_prg_odb_ltrace_active (tcp_conn_id_str);
		tcp_trace_active = conn_id_trace_active || op_prg_odb_ltrace_active ("tcp");
		tcp_retransmission_trace_active = conn_id_trace_active || op_prg_odb_ltrace_active ("tcp_retransmissions");
		tcp_extns_trace_active = conn_id_trace_active || op_prg_odb_ltrace_active ("tcp_extensions");
		}
	else
		{
		tcp_trace_active = OPC_FALSE;
		tcp_retransmission_trace_active = OPC_FALSE;
		tcp_extns_trace_active = OPC_FALSE;
		}

	/* Determine the source of the interrupt. */
```

```
	if (op_intrpt_type () == OPC_INTRPT_SELF)
		{
		/* Since this is a self interrupt, we haven't been		*/
		/* passed an event pointer.  Use the static event		*/
		/* record to hold information about this event.			*/
		evptr = &event_record;
		evptr->event = op_intrpt_code ();
		}
	else
		{
		/* All other interrupts are invocations by the manager. */
		evptr = (TcpT_Event *) op_pro_argmem_access ();
		if (evptr == OPC_NIL)
			{
			tcp_conn_error ("Unable to get event pointer from argument memory.",
				"Socket process invoked without event specification.", OPC_NIL);
			}
		}

	/* If we received a congestion experienced notification from IP		*/
	/* then, we must store it so that all segments sent from this		*/
	/* process will have its "ECE" flag enabled (until a "CWR" flag		*/
	/* from the peer TCP connection is received).					*/
	if (evptr->congestion_experienced == OPC_TRUE)
		{
		tcb_ptr->ecn_status |= TcpC_Ecn_Send_ECE;
		}

	FRET (evptr);
	}

static void
tcp_timeout_retrans (void* PRG_ARG_UNUSED (input_ptr), int PRG_ARG_UNUSED (code))
	{
	char				msg [128];
	Packet*				seg_ptr;
	TcpT_Flag			flags = TCPC_FLAG_NONE;
	double				next_timeout_time;
	int					una_buf_size;
	double				current_time;

	/** Process a retransmission timeout.						**/
	FIN (tcp_timeout_retrans ());

	/* Retransmit only if the surrounding node is not failed.		*/
	/* Otherwise cancell all pending timers.					*/
	if (tcp_parameter_ptr->node_failed == OPC_TRUE)
		{
		FOUT;
		}

	/* Store the current simulation time into a local variable.		*/
	current_time = op_sim_time ();

	/* Reset the count for number of duplicate ACKs received.		*/
	dup_ack_cnt = 0;

	/* Print a diagnostic message if tracing is on.				*/
	if (tcp_trace_active || tcp_retransmission_trace_active)
		{
		sprintf (msg, "<SND.UNA = %u> <RTT = %g> <RTO = %g> <backoff = %d>",
			snd_una, retrans_rtt, current_rto, (int) retrans_backoff);
		op_prg_odb_print_major ("Retransmission Timeout Expired: Resending Segment", msg, OPC_NIL);
		}
```

```
/* Per RFC 2018, if SACK is occuring, the scoreboard is always cleared after a retransmission timeout. */
if (SACK_PERMITTED && (tcp_scoreboard_and_sacklist_ptr_valid == OPC_TRUE))
	{
	tcp_scoreboard_clear ();
	}

/* Update the number of selectively ACKed data.	*/
if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
	op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->sacked_data_stathandle) == OPC_TRUE)
	{
	op_stat_write (tcb_ptr->tcp_conn_stat_ptr->sacked_data_stathandle, 0.0);
	}

/* Write a simulation log message.				*/
tcp_retransmissions_log_write ();

/* Write retransmission count statistics.		*/
tcp_conn_retrans_stat_write ();

/* Increment the count of retransmissions		*/
/* attempted. This count would be cleared every		*/
/* time a valid ack is received.				*/
num_retrans_attempts++;

/* IF this is the first retransmission attempt	*/
/* set the transmission start time to be the	*/
/* the last send time.						*/
if (num_retrans_attempts == 1)
	{
	/* We are in for the first retransmission		*/
	/* which means the actual transmission			*/
	/* started current_rto seconds back .			*/
	transmission_start_time = current_time - current_rto;
	}

/* Cache the last sent segment sequence number		*/
/* for use when receeiving ECN-enabled flags (ECE).*/
if (tcp_flavor != TcpC_Flavor_New_Reno)
	{
	snd_recover = snd_max;
	}

/* Set the slow-start threshold to half the current flight size */
/* (but at least two segments) and reset the current window		*/
/* size to one segment (refer page 310 in TCP/IP Illustrated	*/
/* Vol. I by W. Richard Stevens and to RFC 2581 - equation 3.		*/
ssthresh = (snd_max - snd_una) / 2;

if (ssthresh <= 2.0*snd_mss)
	ssthresh = 2.0*snd_mss;
cwnd = snd_mss;

/* Update the congestion window statistic.				*/
tcp_cwnd_stat_update ();

/* Get a segment (maximum size, if possible) from the		*/
/* retransmission buffer.							*/
una_buf_size = (int) op_sar_buf_size (una_buf);
if (una_buf_size == 0)
	{
	/* If there is nothing in the buffer, we must be		*/
	/* resending a dataless control (SYN or FIN)		*/
	seg_ptr = OPC_NIL;
	seg_len = 0;
	}
```

```
else
	{
	seg_ptr = op_sar_srcbuf_seg_access (una_buf, snd_mss * 8);
	if (seg_ptr == OPC_NIL)
		{
		tcp_conn_warn ("Unable to get segment from nonempty retransmission buffer.",
			"Attempting to send dataless segment instead.", OPC_NIL);
		seg_len = 0;
		}
	else
		{
		seg_len = (TcpT_Size)op_pk_total_size_get (seg_ptr) / 8;
		}
	}

/* If RTT measurements are being done on a segment in this	*/
/* sequence range, or Karn's algorithm is being used, then	*/
/* reset the timer.	*/
if ((rtt_active && tcp_seq_ge (rtt_seq, snd_una)) || (karns_algo_enabled == OPC_TRUE))
	{
	rtt_active = 0;
	}

/* First segment in the stream is SYN.	*/
if (snd_una == iss)
	{
	flags |= TCPC_FLAG_SYN;
	seg_len++;

	/* Include an ACK if the SYN has been received from	*/
	/* the other side.	*/
	if (syn_rcvd)
		{
		flags |= TCPC_FLAG_ACK;
		}
	}
else
	{
	/* Everything but SYN that gets retransmitted carries */
	/* an ACK.	*/
	flags |= TCPC_FLAG_ACK;
	}

/* Last segment in the stream is FIN.	*/
if (snd_fin_valid && (snd_una + seg_len == snd_fin_seq))
	{
	flags |= TCPC_FLAG_FIN;
	seg_len++;
	}

/* Reset the next send sequence number to the oldest	*/
/* unacknowledged sequence number plus the number of bytes*/
/* in the current segment to be transmitted.	*/
snd_nxt = snd_una + seg_len;

/* Print a diagnostic message if tracing is on.	*/
if (tcp_trace_active || tcp_retransmission_trace_active)
	{
	sprintf (msg, "Resetting SND.NXT to = %u", snd_nxt);
	op_prg_odb_print_minor (msg, OPC_NIL);
	}

/* If this is not an MSS-sized packet, it must be on an	*/
/* application packet boundary.  Set PUSH.	*/
if (seg_len < snd_mss)
```

```
		{
		flags |= TCPC_FLAG_PSH;
		}

	/* Check for the case when this segment happens to be the	*/
	/* last segment (happening to be MSS-sized) that this		*/
	/* process needs to send. If yes, then set the PUSH flag		*/
	/* (Note that this is also described in the tcp_output.c */
	/* file for BSD implementation.) This is done in real life		*/
	/* to keep happy those implementations which only give data	*/
	/* to the user when a buffer fills or a PUSH flag comes in		*/
	/* (like the case in this TCP model.)								*/
	else if ((seg_len == snd_mss) &&
				((una_buf_size == snd_mss*8) && (op_sar_buf_size (snd_buf) == 0.0)))
		{
		flags |= TCPC_FLAG_PSH;
		}

	/* Don't need to check for RST; we never retransmit those.	*/

	/* If this falls in outgoing urgent data, set URG.				*/
	if (snd_up_valid && tcp_seq_ge (snd_up, snd_una))
		flags |= TCPC_FLAG_URG;

	/* Update the last send time.										*/
	last_snd_time = op_sim_time ();

	/* Since ECT code point should not be set for retransmitted	*/
	/* segments (section 6.1.5 in RFC-3168), indicate it to IP.	*/
	tcp_ecn_request_to_ip (0);

	/* Send the packet to the lower layer (e.g., IP).	*/
	tcp_seg_send (seg_ptr, snd_una, flags);

	/* Double the backoff factor.	*/
	retrans_backoff *= 2;

	/* Store the sequence number of the retransmitted segment.	*/
	/* Account for the length of the retransmitted packet.		*/
	/* (this could also be set during una_buf processing.)		*/
	max_retrans_seq = snd_una + seg_len - 1;

	/* Calculate the current RTO; restrict it to within limits. */
	current_rto *= 2;
	current_rto = MIN (current_rto, rto_max);

	/* Record the RTO value.  */
	if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
		op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->rto_stathandle) == OPC_TRUE)
		{
		op_stat_write (tcb_ptr->tcp_conn_stat_ptr->rto_stathandle, current_rto);
		}

	/* Compute the next RTO expiration time.	*/
	next_timeout_time = Tcp_Slowtimo_Next_Timeout_Time_Obtain (current_time, current_rto, timer_gran);

	/* Schedule the next retransmission after evaluating the		*/
	/* the boundary conditions on the retransmissions. TCP			*/
	tcp_conn_retrans_timeout_schedule (next_timeout_time);

	/* Collect statistics related to delays in sending segments		*/
	tcp_seg_send_delay_stat_record ();

	FOUT;
	}
```

```
static void
tcp_conn_retrans_timeout_schedule (double next_timeout_time)
            {
            /** Schedule the next retransmission after evaluating the    **/
   /** the boundary conditions on the retransmissions. TCP      **/
   /** can limit the number of retransmissions either to a      **/
   /** maximum number of attempts or to a maximum time duration **/
            FIN (tcp_conn_retrans_timeout_schedule (next_timeout_time))

            /* Handle according to the mode in which the threshold for       */
            /* retransmissions is computed.                                   */
             switch (max_retrans_mode)
       {
       case TcpC_Max_Retrans_Limit_By_Attempts:
          {
          if ((num_retrans_attempts+1) > max_retrans_attempts)
             {
             /* Schedule an interrupt to send out the reset  */
             /* instead of another retransmission time out.  */
             max_retrans_evh = op_intrpt_schedule_call  (next_timeout_time, TCPC_MAX_RETRANS_REACHED,
                                              tcp_connection_on_max_retrans_reset,  OPC_NIL);
             if (op_ev_valid (max_retrans_evh) == OPC_FALSE)
                {
                tcp_conn_warn ("Unable to schedule retransmission timeout.",
                   "No RST will be sent after retransmission limit.", OPC_NIL);
                }
             }
          else
             {
             /* Schedule the retransmission timeout.                    */
             retrans_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_retrans, OPC_NIL);
             if (op_ev_valid (retrans_evh) == OPC_FALSE)
                {
                tcp_conn_warn ("Unable to schedule retransmission timeout.",
                   "No retransmission will take place.", OPC_NIL);
                }
             }

          break;
          }

       case TcpC_Max_Retrans_Limit_By_Interval:
          {
          /* Check if the by next timeout time the maximum    */
          /* interval for retransmission attempts would be    */
          /* exceeded.                                         */
          if ((next_timeout_time - transmission_start_time) > max_retrans_interval)
             {
             /* Schedule an interrupt to send out the reset  */
             /* at the start time plus the maximum           */
             /* retransmission interval.                     */
             max_retrans_evh = op_intrpt_schedule_call  ((transmission_start_time + max_retrans_interval),
                                               TCPC_MAX_RETRANS_REACHED,
                                               tcp_connection_on_max_retrans_reset, OPC_NIL);
             if (op_ev_valid (max_retrans_evh) == OPC_FALSE)
                {
                tcp_conn_warn ("Unable to schedule retransmission timeout.",
                   "No RST will be sent after retransmission limit.", OPC_NIL);
                }
             }
          else
             {
             /* Schedule the retransmission timeout.                    */
             retrans_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_retrans, OPC_NIL);
```

```
            if (op_ev_valid (retrans_evh) == OPC_FALSE)
                {
                tcp_conn_warn ("Unable to schedule retransmission timeout.",
                    "No retransmission will take place.", OPC_NIL);
                }
            }
                                break;
        }

    default:
        {
        /* Schedule the retransmission timeout.                    */
        retrans_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_retrans, OPC_NIL);
        if (op_ev_valid (retrans_evh) == OPC_FALSE)
            {
            tcp_conn_warn ("Unable to schedule retransmission timeout.",
                "No retransmission will take place.", OPC_NIL);
            }

        break;
        }

    /* End of Switch */
    }

    FOUT;
    }

static void
tcp_fast_retrans (void)
            {
            char                                    msg [128];
            TcpT_Seq                                onxt;
            TcpT_Scoreboard_Entry*  sack_entry_ptr;

    /** Process a retransmission due to Fast Retransmit.  Once a threshold number of duplicate         **/
    /** ACKs has been reached, the oldest segment in the retransmission buffer will be resent          **/
    /** even though no timeout has occured (assuming FRFR is enabled).  The congestion window  **/
    /** will be cut approximately in half following retransmission.                                                                      **/
    /**
                                                                      **/
    /** Logic:  Duplicate ACKs containing no new info (data or window update) may indicate                        **/
    /** either packet loss or packet reordering.  After a certain number in a row have been  **/
    /** received, the former is assumed. Retransmission occurs immediately.  In addition, since         **/
    /** duplicate ACKs are being received, packet loss must not be due to network/remote host         **/
    /** failure.  Thus, congestion window can be halved instead of cut to 1 MSS as after a                **/
    /** retransmission timeout.
                                                      **/
    FIN (tcp_fast_retrans ());

            /* If New Reno is used, store the end of the end recovery process.          */
            /* Note that this function is called only once during its recovery             */
            /* within one window of data.                                                                                          */
            if (tcp_flavor == TcpC_Flavor_New_Reno)
                        {
                        snd_recover = snd_max;
                        }

    if (tcp_trace_active || tcp_retransmission_trace_active)
                        {
                        sprintf (msg, "<SND.UNA = %u> <RTT = %g> <RTO = %g> <backoff = %d>",
                            snd_una, retrans_rtt, current_rto, (int) retrans_backoff);
                        op_prg_odb_print_major ("Fast Retransmit Threshold Reached: Resending Segment", msg, OPC_NIL);
                        }
```

```
/* Write a simulation log message.                                    */
tcp_retransmissions_log_write ();

        /* If RTT measurements are currently being taken, reset the timer.        */
        rtt_active = 0;

/* drop the slow start threshold to half the current   */
        /* flight size        (RFC 2581, eq. 3)                                              */
ssthresh = (snd_max - snd_una)/2;

/* ssthresh must be at least 2 maximum sized segments                         */
if (ssthresh < (2 * snd_mss))
                {
                ssthresh = 2 * snd_mss;
                }

        /* If packet send threshold was set to MSS boundary, slow start           */
        /* threshold should be a multiple of maximum segment size.                         */
        if (tcp_parameter_ptr->seg_snd_thresh == TcpC_MSS_Boundary)
                {
                /** Packet Send Threshold attribute was set to MSS boundary. **/

                /* Round down slow start threshold to a multiple of the maximum segment size.  */
                ssthresh = (((int) (ssthresh/snd_mss))* snd_mss);
                }

/* If retransmission is based on SACKs, set up the pipe variable to     */
/* estimate the amount of outstanding data.                                          */
if ((SACK_RECOVERY))
                {
                /* bytes of outstanding data is the amount of data transmitted but not     */
                /* ack'd, minus the data included in the rcvd packets which generated     */
                /* dup ACKs, minus the data in the segment presumed lost                   */

                /* If packet send threshold was set to MSS boundary, slow start            */
                /* threshold should be a multiple of maximum segment size.                         */
                if (tcp_parameter_ptr->seg_snd_thresh == TcpC_MSS_Boundary)
                        {
                        pipe = (snd_nxt - snd_una) - (snd_mss * dup_ack_cnt) - snd_mss;
                        }
                else
                        {
                        /* The number of data in a pipe is the number of unACKed data minus  the          */
                        /* number of SACK ACKed data minus the size of the packet presumably          */
                        /* lost. Calculate the size of the lost packet. This is the difference          */
                        /* between the start of the first SACK block in the SACK list, and                  */
                        /* snd_una. If this difference is greater than mss, set it to mss.              */
                        if (op_prg_list_size (scoreboard_ptr->entries) > 0)
                                {
                                /** There are some data that have been SACK ACKed.            */

                                /* Take the first entry from the SACK list. Entries are ordered by          */
                                /* sequence numbers, so this is indeed the first SACKed block.                    */
                                sack_entry_ptr = (TcpT_Scoreboard_Entry *) op_prg_list_access (scoreboard_ptr->entries,
OPC_LISTPOS_HEAD);

                                if (tcp_seq_ge ((sack_entry_ptr->start - snd_una),snd_mss))
                                        {
                                        /** There is more than one packet which has not been ACKed.*/
                                        /* Assume that only one packet was dropped.                                                */
                                        pipe = (snd_nxt - snd_una) - tcp_sack_number_sacked_bytes_find () - snd_mss;
                                        }
                                else
                                        {
                                        /** The size of the dropped packet is less than MSS.  **/
```

```
					pipe = (snd_nxt - snd_una) - tcp_sack_number_sacked_bytes_find () - (sack_entry_ptr->start - 
snd_una);
					}
				}
			else
				{
				/* No assumptions can be made regarding teh packet sizes.	*/
				/* Assume the size of the dropped packet was mss.			*/
				pipe = (snd_nxt - snd_una) - tcp_sack_number_sacked_bytes_find () - snd_mss;
				}
			}

		if (pipe < 0)
		    pipe = 0;
		}

/* Do not want to continue retransmission of data following lost packet		*/
/* Thus, save current value of snd_nxt so it can be restored afterwards.		*/
onxt = snd_nxt;
snd_nxt = snd_una;

/* Only want to send one packet now.	*/
cwnd = snd_mss;

	/* Cache the last sent segment sequence number	*/
	/* for use when receeiving ECN-enabled flags (ECE).*/
	snd_recover = snd_max;

/* Retransmit the missing packet.  Only one will be transmitted due to			*/
/* cwnd, if SACK is not enabled.  If SACK is enabled, still only transmits	*/
/* one packet because tcp_snd_data_size () limits una_buf processing to max	*/
/* of one MSS at a time during SACK-based recovery.						*/
tcp_una_buf_process (OPC_FALSE);

/* Cut the congestion window approximately in half when supporting FRFR	*/
	/* Fast Retransmit is possible only if "TCP Tahoe" or "TCP Reno" is used*/
	/* In case of "TCP Reno" after Fast Retransmit the congestion window is		*/
	/* reduced to half of the minimum of (ssthresh,rcv_win). Also each			*/
	/* duplicate packet is considered an ACK for an already sent packet and		*/
	/* congestion window is thus incremented for each duplicate ACK rcvd.		*/
	/* (Reno Flavor), Otherwise the congestion window would be set to 1 MSS		*/
	if ((tcp_flavor == TcpC_Flavor_Reno) || (tcp_flavor == TcpC_Flavor_New_Reno))
		{
		/* restore the value of send_next, allowing it to advance if necessary	*/
		snd_nxt = MAX(snd_nxt, onxt);

		/* Print a diagnostic message if tracing is on.				*/
		if (tcp_trace_active)
			{
			sprintf (msg, "Setting SND.NXT to = %u", snd_nxt);
			op_prg_odb_print_minor (msg, OPC_NIL);
			}

	cwnd = ssthresh;

		/* If SACK is not being used to determine packet retransmission, inflate		*/
		/* the congestion window once for each packet assumed to be stored by		*/
		/* receiver (i.e. those packets which caused the dup ACKS)				*/
		if (!(SACK_RECOVERY))
			{
			cwnd += (dup_ack_cnt * snd_mss);
			}
		}

	else if (tcp_flavor == TcpC_Flavor_Tahoe)
```

```
			{
			/* In case of "TCP Tahoe" after Fast Retransmit the congestion		*/
			/* window is reduced to one MSS and slow start is re-initiated.*/
			cwnd = snd_mss;
			}

		/* Record the congestion window statistics.	*/
		tcp_cwnd_stat_update ();

	/* Reset the retransmission timer */
		tcp_retrans_timer_reset ();

	/* Store the sequence number of the retransmitted segment.	*/
	/* Account for the length of the retransmitted packet.			*/
	max_retrans_seq = snd_una + seg_len - 1;

		/* Collect statistics related to delays in sending segments		*/
		tcp_seg_send_delay_stat_record ();

		FOUT;
	}

static void
tcp_timeout_delay_ack (void* PRG_ARG_UNUSED (input_ptr), int PRG_ARG_UNUSED (code))
		{
		/** Don't wait any more for outgoing data.	**/
		/** Send a dataless ACK instead.				**/
		FIN (tcp_timeout_delay_ack ());

		/* Retransmit only if the surrounding node is not failed.		*/
		/* Otherwise cancell all pending timers.						*/
		if (tcp_parameter_ptr->node_failed == OPC_TRUE)
			{
			FOUT;
			}

		/* If we are in the "Segment/Clock Based" mode of	*/
		/* generating ACKs, reset the counter used to store	*/
		/* the number of segments received without sending	*/
		/* a delayed acknowledgment.							*/
		if (tcp_del_ack_scheme == TcpC_Segment_And_Timer_Based)
			{
			tcp_segments_rcvd_without_sending_ack = 0;
			}

		/* Send the delayed ACK packet. */
		tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_ACK);

		FOUT;
		}

static void
tcp_timeout_persist (void* PRG_ARG_UNUSED (input_ptr), int PRG_ARG_UNUSED (code))
		{
		char			msg [256];
		Packet*			data_ptr;
		TcpT_Flag		flags = TCPC_FLAG_ACK;
		double			next_timeout_time;

		/** All sent data has been acknowledged, but the remote		**/
		/** receive window is still closed.  Send a single byte **/
		/** of data after the persistence timeout has expired	**/
		/** so we can get a window size update.					**/
		FIN (tcp_timeout_persist ());
```

```
/* Retransmit only if the surrounding node is not failed.          */
/* Otherwise cancell all pending timers.                                           */
if (tcp_parameter_ptr->node_failed == OPC_TRUE)
        {
        FOUT;
        }

if (op_sar_buf_size (snd_buf) > 0.0)
        {
        /* Print a diagnostic message if tracing is on.           */
        if (tcp_trace_active)
                {
                sprintf (msg, "<SND.UNA = %u> <SND.MAX = %u> <SND.NXT = %u> <SND.WND = %u>",
                        snd_una, snd_max, snd_nxt, snd_wnd);
                op_prg_odb_print_major ("Persistence Timeout Expired: Sending one octet of data", msg, OPC_NIL);
                }

        /* Send a single byte of new data.                                  */
        data_ptr = op_sar_srcbuf_seg_remove (snd_buf, 8);
        if (data_ptr == OPC_NIL)
                {
                tcp_conn_warn ("Unable to get single-byte segment from unsent data buffer.",
                        "Will not send segment although persistence timeout has expired.", OPC_NIL);
                FOUT;
                }

        /* If this falls in outgoing urgent data, set URG. */
        if (snd_up_valid && tcp_seq_ge (snd_up, snd_max))
                flags |= TCPC_FLAG_URG;

        /* Add the byte to the retransmission buffer.           */
        op_sar_rsgbuf_seg_insert (una_buf, op_pk_copy (data_ptr), 0, 0);

        /* Since ECT code point should not be set for non-     */
        /* data segments, indicate this to IP.                              */
        tcp_ecn_request_to_ip (0);

        /* Send the data and update the sequence number of   */
        /* next segment to be sent.                                                  */
        tcp_seg_send (data_ptr, snd_nxt, flags);
        snd_nxt++;

        /* Update the maximum send sequence number. In an*/
        /* ideal case, when the system has not observed any   */
        /* retransmissions, SND.NXT and SND.MAX are equal.          */
        /* Refer page 808 (TCP/IP Illustrated Volume 2)                 */
        if (tcp_seq_ge (snd_nxt, snd_max))
                {
                snd_max++;
                }

        /* Compute the next RTO expiration time.    */
        next_timeout_time = Tcp_Slowtimo_Next_Timeout_Time_Obtain (op_sim_time (), current_rto, timer_gran);

        /* Schedule a retransmission timeout.  Note that no     */
        /* retransmission timeout is currently pending.                */
        retrans_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_retrans, OPC_NIL);

        if (op_ev_valid (retrans_evh) == OPC_FALSE)
                tcp_conn_warn ("Unable to schedule retransmission timeout.",
                        "No retransmission will take place.", OPC_NIL);
        }
else if (fin_segment_sent == OPC_FALSE)
        {
        /* Print a diagnostic message if tracing is on.           */
```

```
			if (tcp_trace_active)
				{
				sprintf (msg, "<SND.UNA = %u> <SND.MAX = %u> <SND.NXT = %u> <SND.WND = %u>",
					snd_una, snd_max, snd_nxt, snd_wnd);
				op_prg_odb_print_major ("Persistence Timeout Expired: Send buffer is empty.", msg, OPC_NIL);
				}

			/* If there is now no outstanding unacknowledged data, but the*/
			/* remote receive window size is zero, set the persistence			*/
			/* timeout so we can poll the remote receive window size.			*/
			if (snd_una == snd_max && snd_wnd == 0)
				{
				if (!op_ev_valid (persist_evh) || !op_ev_pending (persist_evh))
					{
					/* Compute the next persistance expiration time.			*/
					next_timeout_time = Tcp_Slowtimo_Next_Timeout_Time_Obtain (op_sim_time (), persist_timeout,
timer_gran);

					persist_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_persist, OPC_NIL);
					if (op_ev_valid (persist_evh) == OPC_FALSE)
						tcp_conn_warn ("Unable to schedule persistence timeout.",
						"Remote receive window size is closed but will not be polled.", OPC_NIL);
					}
				}
			}

	FOUT;
	}

static void
tcp_command_send (Packet* pk_ptr, TcpT_Flag flags)
	{
	char			str0 [256];
	TcpT_Size		pk_size;

	/** The application has issued a SEND command.  Add the	**/
	/** provided data packet to the unsent data buffer.			**/
	FIN (tcp_command_send (pk_ptr, flags));

	if (tcp_trace_active)
		{
		sprintf (str0, "Packet id (" SIMC_PK_ID_FMT "), tree id (" SIMC_PK_ID_FMT ") %s",
				 op_pk_id (pk_ptr), op_pk_tree_id (pk_ptr),
				(flags & TCPC_FLAG_URG ? "[URGENT]" : ""));
		op_prg_odb_print_major ("TCP received command: SEND", str0, OPC_NIL);
		}

	if (op_pk_total_size_get (pk_ptr) == 0)
		{
		tcp_conn_error ("Received zero-size packet from application.",
			"All data packets sent to TCP must have nonzero size.", OPC_NIL);
		}
	else
		{
		/* If the application data packet is not on a byte			*/
		/* boundary, the packet is padded to the nearest byte.  */
		pk_size = (TcpT_Size) op_pk_total_size_get (pk_ptr);

		if ((pk_size % 8) != 0)
			{
			pk_size += (8 - (pk_size % 8));
			op_pk_total_size_set (pk_ptr, pk_size);
			}
		}
```

```
	op_pk_stamp (pk_ptr);

	/* Put the packet into the unsent data buffer. */
	op_sar_segbuf_pk_insert (snd_buf, pk_ptr, 0);

	/* If the URG flag was on, set the urgent pointer.	*/
	/* Note that this model always uses the "correct"	*/
	/* (RFC 1122) interpretation of the urgent pointer,	*/
	/* not the common (BSD) interpretation.			*/
	if (flags & TCPC_FLAG_URG)
		{
		snd_up_valid = 1;
		snd_up = snd_nxt + (OpT_Sar_Size) (op_sar_buf_size (snd_buf) / 8) - 1;
		}

	FOUT;
	}

static void
tcp_command_receive (int num_pks)
	{
	char			msg [128];

	/** The application has issued a RECEIVE command.		**/
	/** Increment the count of packets requested.			**/
	FIN (tcp_command_receive (num_pks));

	/* Set the new number of requested packets. */
	num_pks_req += num_pks;

	if (tcp_trace_active)
		{
		sprintf (msg, "Application now waiting for %d packets.", num_pks_req);
		op_prg_odb_print_major ("TCP received command: RECEIVE", msg, OPC_NIL);
		}

	FOUT;
	}


static void
tcp_seg_send (Packet* data_pk_ptr, TcpT_Seq seq, TcpT_Flag flags)
	{
	Packet*				seg_ptr;
	TcpT_Seg_Fields*		tcp_seg_fd_ptr;
	char				str0 [256];
	unsigned int			my_timestamp;

	/** Send the provided data packet with the indicated flags across		**/
	/** the connection; (seq) is the outgoing sequence number.			**/
	FIN (tcp_seg_send (data_pk_ptr, seq, flags));

	/* Check if the ECE flags needs to be set in the segment being sent	*/
	/* This acts like an indication for the remote peer to slow down in	*/
	/* order to avoid potential congestion in the network.			*/
	if (tcb_ptr->ecn_status & TcpC_Ecn_Send_ECE)
		{
		flags |= TCPC_FLAG_ECE;
		}

	/* Check if the CWR flag needs to be set this segment to be sent.	*/
	if (tcb_ptr->ecn_status & TcpC_Ecn_Send_CWR)
		{
		flags |= TCPC_FLAG_CWR;
```

```
	/* The TCP sender sets the CWR flag in the TCP header of the*/
	/* first new data packet sent after the window reduction.			*/
	tcb_ptr->ecn_status ^= TcpC_Ecn_Send_CWR;
	}

/* Create a TCP segment to encapsulate the data. */
seg_ptr = op_pk_create_fmt ("tcp_seg_v2");

/*	Create "fields" structure that contains information like src	*/
/*	port, dest port, seq num, ack, ack_num etc.					*/
tcp_seg_fd_ptr = tcp_seg_fdstruct_create ();
if (seg_ptr == OPC_NIL)
	{
	tcp_conn_error ("Unable to create TCP segment.", OPC_NIL, OPC_NIL);
	}

/* Set the TCP port numbers. */
tcp_seg_fd_ptr->src_port  = tcb_ptr->local_port;
tcp_seg_fd_ptr->dest_port		= tcb_ptr->rem_port;
tcp_seg_fd_ptr->seq_num = seq;

/* Set the fast-lookup keys maintained by the TCP manager process	*/
/* for efficient lookup of connection processes.				*/
tcp_seg_fd_ptr->local_key  = tcb_ptr->local_key;
tcp_seg_fd_ptr->remote_key = tcb_ptr->remote_key;

/* Set the flags passed in to this procedure in the segment to be sent.	*/
tcp_seg_fd_ptr->flags = flags;

/* If this segment is an ACK, set the ACK flag and sequence number. */
/* We always want to use the most recent acknowledgement number.	*/
if (flags & TCPC_FLAG_ACK)
	{
	tcp_seg_fd_ptr->ack_num = rcv_nxt;

	/* Since there is an ACK going out with this segment, turn	*/
	/* off the timer for sending a dataless ACK, if it is set.		*/
	if (op_ev_valid (ack_evh) && op_ev_pending (ack_evh))
		{
		if (op_ev_cancel (ack_evh) == OPC_COMPCODE_FAILURE)
			{
			tcp_conn_warn ("Unable to cancel acknowledgement timer.",
				"A dataless acknowledgement may be sent inappropriately.", OPC_NIL);
			}
		}
	}

/* Set the advertized window. If the usage threshold is set	*/


/* to zero, then the complete receive buffer is advertized.	*/
if (rcv_buf_usage_thresh == 0.0)
	{
	/* Advertise the full window.	*/
	rcv_wnd = rcv_buff;

	rcv_wnd = cwnd;
	}

/* Advertise the size of the receive buffer.			*/
if (window_scaling_enabled == TCPC_OPTION_STATUS_ENABLED)
	{
	/* Shift by zero if scaling not enabled.			*/
	tcp_seg_fd_ptr->rcv_win = rcv_wnd >> rcv_scale;
	}
```

```
else
		{
		tcp_seg_fd_ptr->rcv_win = rcv_wnd;
		}

if (tcp_trace_active || tcp_extns_trace_active)
		{
		sprintf (str0, "The receive window just sent is %u.", rcv_wnd);
		op_prg_odb_print_minor (str0, OPC_NIL);
		}

/* The base "width" of this segment in sequence number		*/
/* space is the total size of the encapsulated data.		*/
if (data_pk_ptr == OPC_NIL)
		{
		/* This must be a pure ACK packet -- hence zero segment length.		*/
		seg_len = 0;

		/* Since ECT code point should not be set for dataless ACKs,	*/
		/* indicate this to IP. Refer to 6.1.4 in RFC-3168 for details.	*/
		tcp_ecn_request_to_ip (0);
		}
else
		{
		seg_len = (TcpT_Size) op_pk_total_size_get (data_pk_ptr) / 8;
		if (op_pk_nfd_set (seg_ptr, "data", data_pk_ptr) == OPC_COMPCODE_FAILURE)
				tcp_conn_error ("Unable to set data in TCP segment.", OPC_NIL, OPC_NIL);

		/* A packet with data is being created. This packet will		*/
		/* also carry an ACK. Reset the number of unACKed segments.		*/
		tcp_segments_rcvd_without_sending_ack = 0;

		/*		Cancel the timer pending to send a dataless acknowledgment.	*/
		if (op_ev_valid (ack_evh) == OPC_TRUE)
				op_ev_cancel (ack_evh);
		}

/* If SACK is enabled, and an out-of-order data has been			*/
/* received, add a SACK option to the packet.					*/
if (SACK_PERMITTED && (op_prg_list_size (rcv_rec_list) > 0))
		{
	tcp_sackoption_set (seg_ptr);
		}

/* Check if the urgent point flag needs to be set. It indicates		*/
/* that received data must be sent up immediately.				*/
if (flags & TCPC_FLAG_URG)
		{
		tcp_seg_fd_ptr->urgent_pointer = snd_up;
		}

/* Both SYN and FIN must be acknowledged, so they are assigned		*/
/* one byte in sequence number space.  They do not contribute	*/
/* to the actual size of the data packet, however.				*/
if (flags & TCPC_FLAG_SYN)
		{
		/* Increment the segment length to account for this SYN.		*/
		seg_len++;

		/* Initialize the variable used to store the sequence			*/
		/* number of a retransmitted segment. Account for the			*/
		/* length of this transmitted SYN packet.					*/
		max_retrans_seq = snd_una + seg_len - 1;

		/* Add the MSS option to an outgoing SYN.  Note that this		*/
```

```
		/* is not necessarily SND.MSS, because we may already have */
		/* received the MSS Option from the remote side.			*/
		if (op_pk_nfd_set (seg_ptr, "MSS Option", snd_mss) == OPC_COMPCODE_FAILURE)
			tcp_conn_error ("Unable to set MSS option in TCP segment.", OPC_NIL, OPC_NIL);

		/* If possible, add the Window Scaling Option to the outgoing SYN.	*/
		/* This can be done under two conditions:						*/
		/*		1.  This is an active open (ACK is not set)				*/
		/*		2.  This is a passive open (ACK set) but Window Scaling Option		*/
		/*		    was set in the received SYN.						*/
		if ((window_scaling_enabled == TCPC_OPTION_STATUS_ENABLED) && (!(flags & TCPC_FLAG_ACK) ||
wnd_scale_rcvd))
			{
			/* Value of requested rcv_scale is determined by the size of	*/
			/* the receive buffer. Use maximum scale factor possible.		*/
			while ((requested_rcv_scale < TCPC_WS_MAX_WND_SHIFT) &&
				(TCPC_MAX_WND_SIZE << requested_rcv_scale < rcv_buff))
				{
				requested_rcv_scale++;
				}

			/* Set the option in the outgoing SYN.						*/
		  if (op_pk_nfd_set (seg_ptr, "Window Scaling Option", requested_rcv_scale) == OPC_COMPCODE_FAILURE)
				{
				tcp_conn_error ("Unable to set Window Scaling Option in TCP SYN segment.", OPC_NIL, OPC_NIL);
				}

			/* Indicate that the window scale option has been sent.				*/
		  wnd_scale_sent = OPC_TRUE;

			if (tcp_trace_active || tcp_extns_trace_active)
				{
				sprintf (str0, "Requested a window scaling factor of %d.", requested_rcv_scale);
				op_prg_odb_print_minor (str0, OPC_NIL);
				}
			}

		/* If possible, add the SACK-Permitted Option to the outgoing SYN.	*/
		/* This can be done under two conditions:						*/
		/*		1.  This is an active open (ACK is not set)				*/
		/*		2.  This is a passive open (ACK set) but SACK-Permit Option		*/
		/*			was set in the received SYN.						*/
		if ((sack_enabled == TCPC_OPTION_STATUS_ENABLED) && (!(flags & TCPC_FLAG_ACK) || sack_permit_rcvd))
			{
			/* Set the option in the outgoing SYN.						*/
			if (op_pk_nfd_set (seg_ptr, "SACK-Permitted Option", OPC_TRUE) == OPC_COMPCODE_FAILURE)
				{
				tcp_conn_error ("Unable to set SACK-Permitted Option in TCP SYN segment.", OPC_NIL, OPC_NIL);
				}

			/* Indicate that the SACK option has been sent.					*/
		  sack_permit_sent = OPC_TRUE;

			if (tcp_trace_active || tcp_extns_trace_active)
				{
				op_prg_odb_print_minor ("Sent a SACK-Permitted Option.", OPC_NIL);
				}
			}

		/* Check whether TCP timestamp option is supported. If so, set the		*/
		/* option in the outgoing segment.							*/
		if (flags & TCPC_FLAG_ACK)
			{
			/* The process is sending a SYN-ACK, in response to a SYN message */
			if (conn_supports_ts == TCPC_OPTION_STATUS_ENABLED)
```

```
				{
				/* This connection supports Timestamp. Create a timestamp	*/
				/* option field in the segment and set it.						*/
				my_timestamp = (unsigned int) ((floor) ((op_sim_time () - conn_start_time) / (tcp_parameter_ptr-
>timestamp_clock / 1000.0)));
				tcp_seg_timestamp_set (seg_ptr, timestamp_info.ts_recent, my_timestamp);

				/* Store last ACK number. */
				timestamp_info.last_ack_sent = rcv_nxt;

				/* Print trace information.  */
				if (tcp_extns_trace_active)
					{
					sprintf (str0, "Setting echo reply timestamp to %f and last ack sent to: %d", time-
stamp_info.ts_recent, timestamp_info.last_ack_sent);
					op_prg_odb_print_minor (str0, OPC_NIL);
					}
				}
			}
		else
			{
			/* We are sending a SYN packet. Check to see if Timestamp option is enabled.	*/
			if (conn_supports_ts == TCPC_OPTION_STATUS_ENABLED)
				{
				/* This side supports Timestamp option. Create a		*/
				/* timestamp option field for this pkt and set it.		*/
				timestamp_info.ts_recent = 0.0;
				tcp_seg_timestamp_set (seg_ptr, timestamp_info.ts_recent, conn_start_time);

				/* Print trace information.  */
				if (tcp_extns_trace_active)
					{
					op_prg_odb_print_minor ("This side supports Timestamp option.", OPC_NIL);
					}
				}
			}
		}
	else
		{
		/* A packet with ACK is being sent out.		*/
		if (conn_supports_ts == TCPC_OPTION_STATUS_ENABLED)
			{
			/* This connection supports Timestamp. Set timestamp			*/
			/* to the current time and echo the received timestamp			*/
			my_timestamp = (unsigned int) ((floor) ((op_sim_time () - conn_start_time) / (tcp_parameter_ptr-
>timestamp_clock / 1000.0)));
			tcp_seg_timestamp_set (seg_ptr, timestamp_info.ts_recent, my_timestamp);

			/* Store last ACK number. */
			timestamp_info.last_ack_sent = rcv_nxt;

			/* Print trace information.  */
			if (tcp_extns_trace_active)
				{
				sprintf (str0, "Setting echo reply timestamp to %u and last ack sent to: %u", timestamp_info.ts_recent,
timestamp_info.last_ack_sent);
				op_prg_odb_print_minor (str0, OPC_NIL);
				}
			}
		}

	/* The FIN statement also occupies one "byte" length.		*/
	if (flags & TCPC_FLAG_FIN)
		{
		seg_len++;
```

```
		}

	/* In trace mode, issue a statement describing the segment. */
	if (tcp_trace_active)
		tcp_seg_msg_print ("Sending -->", seq, rcv_nxt, seg_len, flags);

	/* Set the sequence width of the packet. */
	tcp_seg_fd_ptr->data_len = seg_len;

	/*	Set the structure in the packet.	*/
	op_pk_nfd_set (seg_ptr, "fields", tcp_seg_fd_ptr, tcp_seg_fdstruct_copy, tcp_seg_fdstruct_destroy, sizeof (TcpT_Seg_Fields));

	/* Store the information that will be carried with the ICI. */
	ip_encap_ici_info.dest_addr = tcb_ptr->rem_addr;
	ip_encap_ici_info.src_addr = tcb_ptr->local_addr;
	op_ici_attr_set (ip_encap_ici_info.ip_encap_req_ici_ptr, "Type of Service", tcb_ptr->type_of_service);
	op_ici_install (ip_encap_ici_info.ip_encap_req_ici_ptr);

	/* Stamp this segment so that the destination TCP can compute segment delays.		*/
	op_pk_stamp (seg_ptr);

	/* Send the packet to the lower layer. Note that the stream interrupt is forced;	*/
	/* This is necessary because the ici is reused.
	*/
	op_pk_send_forced (seg_ptr, TCPC_OUTSTRM_NETWORK);

	/* Update the statistics monitoring sequence numbers of sent segments. */
	if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL)
		{
		if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->sent_seg_seq_no_stathandle) == OPC_TRUE)
			op_stat_write (tcb_ptr->tcp_conn_stat_ptr->sent_seg_seq_no_stathandle, seq);

		if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->sent_seg_ack_no_stathandle) == OPC_TRUE)
			op_stat_write (tcb_ptr->tcp_conn_stat_ptr->sent_seg_ack_no_stathandle, rcv_nxt);
		}

	FOUT;
	}


static void
tcp_ecn_request_to_ip (int set_ect_codepoint)
	{
	/** For ECN-enabled connections:						**/
	/**  1. Pure ACKs must be sent with the non-ECT codepoint	**/
	/**     refer to section 6.1.4 in RFC-3168).			**/
	/**  2. Data must be sent with the ECT codepoint set (refer	**/
	/**     to section 6.1.2 in RFC-3168).					**/
	FIN (tcp_ecn_request_to_ip (set_ect_codepoint));

	/* Specify request only is ECN is supported on this TCP connection.	*/
	if (tcb_ptr->ecn_status & TcpC_Ecn_Supported)
		{
		if (op_ici_attr_set (ip_encap_ici_info.ip_encap_req_ici_ptr, "ECN", set_ect_codepoint) == OPC_COMPCODE_FAILURE)
			tcp_conn_error ("Unable to set ECN field in ICI sent to IP.", OPC_NIL, OPC_NIL);
		}

	FOUT;
	}

static void
tcp_seg_receive (Packet* seg_ptr, TcpT_Flag flags)
	{
	Packet*					data_ptr;
	Packet*					pkptr;
```

```
int                                             i, list_size;
int                                             i_th_segment, total_segments;
TcpT_Seq                                rcv_nxt_old;
TcpT_Seq                                rcv_nxt_bfr_ooo_processed;
TcpT_Seq                                seg_up;
TcpT_Seg_Record*                seg_rec_ptr;
TcpT_Seg_Record*                new_rec_ptr;
static Pmohandle        segrec_pmh;
static Boolean                  out_of_order_seg_pmh_created = OPC_FALSE;
Boolean                                 segment_exists = OPC_FALSE;
TcpT_Size                               pkt_size;
OpT_Sar_Size                    bit_index;
OpT_Sar_Size                    bit_count;
int                                             rcv_buf_size;
int                                             rcv_buf_usage_limit;
int                                             discard_buf_size;
Boolean                                 complete_pkt_rcvd = OPC_FALSE;
static Sbhandle                 discard_buf;
static Boolean                  discard_buf_init = OPC_FALSE;
char                                    err0 [128], err1 [128], err2 [128];

TcpT_Seg_Fields*                pk_fd_ptr;              /* Structure containing fields of the received TCP              */
                                                                                                /* segment (modeled like this for sim effi-
ciency)         */
TcpT_Size                               rcv_buf_free;
Boolean                                 ack_sent = OPC_FALSE;

/** Process the newly received segment.  Put as much of the     **/
/** data as is possible into the received data buffer.          **/
FIN (tcp_seg_receive (seg_ptr, flags));

/* A resegmentation buffer "clip_buf" is used to separate       */
/* the overlapping portions of received segments from the       */
/* portions that are new w.r.t. rcv_buf contents.                       */
if (discard_buf_init == OPC_FALSE)
        {
        discard_buf = op_sar_buf_create (OPC_SAR_BUF_TYPE_RESEGMENT, OPC_SAR_BUF_OPT_DEFAULT);
        discard_buf_init = OPC_TRUE;
        }

/* Store the starting value for RCV.NXT.        */
rcv_nxt_old = rcv_nxt;

/* Check for urgent pointer.            */
if (flags & TCPC_FLAG_URG)
        {
        /* Access fields data structure from the packet for     */
        /* obtaining urgent pointer from the packet.            */
        op_pk_nfd_access (seg_ptr, "fields", &pk_fd_ptr);
        seg_up = pk_fd_ptr->urgent_pointer;

        if (tcp_seq_gt (seg_up, rcv_up) || !rcv_up_valid)
                rcv_up = seg_up;
        rcv_up_valid = 1;
        }

/* Check for FIN control.                                                                       */
if (flags & TCPC_FLAG_FIN)
        {
        /* The FIN takes up one unit of sequence space width.           */
        /* Account for this and resume processing the segment.          */
        seg_len--;

        /* Record the sequence number of the FIN.  The FIN is           */
        /* the first octet after any data in this segment.              */
```

```
			rcv_fin_valid = 1;
			rcv_fin_seq = seg_seq + seg_len;
			}

	/* Process the data contained in the segment.*/
	if (op_pk_nfd_is_set (seg_ptr, "data") == OPC_TRUE)
			{
			/* Extract data from the arriving segment.	*/
			if (op_pk_nfd_get (seg_ptr, "data", &data_ptr) == OPC_COMPCODE_FAILURE)
				tcp_conn_error ("Unable to get data from received TCP segment.", OPC_NIL, OPC_NIL);

			/* If the received packet is due to a retransmission	*/
			/* from the other end of this connection, it may be for*/
			/* any amount of data.  This means that if there are	*/
			/* "holes" in the segments that have been received by */
			/* this process, the received packet may fill them,		*/
			/* overlap the already received ones, and exceed the	*/
			/* maximum sequence that we may have already received.	*/

			/* Test whether this data has arrived in sequence.		*/
			if (tcp_seq_le (seg_seq, rcv_nxt))
				{
				/* Arriving data is in sequence. Check if the */
				/* segment has the PUSH flag set (the PUSH flag	*/
				/* covers up to the last byte of the data).		*/
				if (flags & TCPC_FLAG_PSH)
				   {
				   /* Setting of the PUSH flag by the remote TCP indicates	*/
				   /* that the segment forms the data packet boundary.			*/
				   /* This indicates that this segment will complete a  */
				   /* data packet reception.								*/
				   complete_pkt_rcvd = OPC_TRUE;
				   push_seq = seg_seq;
				   }

				/* Check if there is overlapping data in the packet	*/
				if (tcp_seq_lt (seg_seq, rcv_nxt))
					{
					if (tcp_trace_active)
						{
						sprintf (err1, "Initial Overlap Sequence: %u \trcv_nxt: %u.", seg_seq, rcv_nxt);
						op_prg_odb_print_minor ("Overlapping TCP segment data received.", err1, OPC_NIL);
						}

					/* This is the next expected segment; buffer it		*/
					op_sar_rsgbuf_seg_insert (discard_buf, data_ptr, seg_seq, 0.0);

	 /* Obtain the range of bits in the SAR buffer   */
	 op_sar_rsgbuf_lbl_seg_range_get (discard_buf, seg_seq, &bit_index, &bit_count);

					/* Print debugging information, if enabled.		*/
					if (tcp_trace_active)
						{
						sprintf (err1, "Initial Overlap Sequence: %u; Initial RCV.NXT: %u", seg_seq, rcv_nxt_old);
						sprintf (err2, "Flushing %d bytes starting %u.", (rcv_nxt - seg_seq), seg_seq);
						op_prg_odb_print_major ("Overlapping TCP segment data received.", err1, err2, OPC_NIL);
						}

	 /* Flush the initial bits that have already	*/
	 /* arrived before this segment			*/
	 op_sar_rsgbuf_bits_flush_abs (discard_buf, bit_index, (rcv_nxt - seg_seq) * 8);

					/* Remove the remaining amount from discard buffer.		*/
					discard_buf_size = (OpT_Sar_Size) op_sar_buf_size (discard_buf);
					if (discard_buf_size > 0)
```

```
                    {
                    /* Remove the contents and insert them in the           */
                    /* receive buffer.                                      */
                    pkptr = op_sar_srcbuf_seg_remove (discard_buf, op_sar_buf_size (discard_buf));

                    /* Insert this segment in the receive buffer.   */
                    op_sar_rsmbuf_seg_insert (rcv_buf, pkptr);

                    /* Update the sequence number of the next               */
                    /* expected segment                                     */
                    rcv_nxt = rcv_buf_seq + (OpT_Sar_Size) op_sar_buf_size (rcv_buf) / 8;
                    }
            else
                    {
                    /* The incoming data is completely overlapping.         */
                    tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_ACK);
                    }
            }
    else
            {
            /* There is no overlapping data and this is     */
            /* the next expected segment                            */
            op_sar_rsmbuf_seg_insert (rcv_buf, data_ptr);

            /* Update the sequence number of the next               */
            /* expected segment                                     */
            rcv_nxt = rcv_buf_seq +  (OpT_Sar_Size) op_sar_buf_size (rcv_buf) / 8;

            /* Print trace message, if enabled.   */
            if (tcp_trace_active)
                    {
                    op_prg_odb_print_major ("Data received in sequence. Adding it to receive buffer.", OPC_NIL);
                    }

            /* Process the out-of-order received segments.          */
            list_size = op_prg_list_size (rcv_rec_list);
            total_segments = list_size;
            if (tcp_trace_active && list_size > 0)
                    {
                    sprintf (err0, "Updated RCV.NXT from %u to %u.", rcv_nxt_old, rcv_nxt);
                    op_prg_odb_print_major ("Processing received-out-of-order list:", err0, OPC_NIL);
                    }

            /* Record the value of the next expected sequence number        */
            /* before out-of-order list is processed.                       */
            /* After out-of-order list is processed, this value will  */
            /* be used to determine whether this segment filled in all      */
            /* or part in a sequence space.                                 */
            rcv_nxt_bfr_ooo_processed = rcv_nxt;

            /* Move previously received out-of-order segments into the      */
            /* receive buffer if all preceding data has been received.      */
            /* Note that the records are sorted by sequence number.         */
            for (i_th_segment = 0, i = 0; i < list_size; i++)
                    {
                    seg_rec_ptr = (TcpT_Seg_Record *) op_prg_list_access (rcv_rec_list, i);
                    if (seg_rec_ptr == OPC_NIL)
                            {
                            tcp_conn_warn ("Unable to get segment record from reordering list.",
                                    "Skipping to next record in list.", OPC_NIL);
                            continue;
                            }
                    else
                            {
                            if (tcp_trace_active)
```

```
{
i_th_segment++;
sprintf (err0, "Accessing segment %d of %d ....", i_th_segment, to-
tal_segments);

sprintf (err1, "   Sequence number: %u", seg_rec_ptr->seq);
op_prg_odb_print_minor ("", err0, err1, OPC_NIL);
}
}

/* Obtain the size of the segment in bytes      */
pkt_size = (TcpT_Size) op_pk_total_size_get (seg_rec_ptr->data_ptr) / 8.0;

/* Check if there an overlap in this out of order segment.          */
if (tcp_seq_le (seg_rec_ptr->seq, rcv_nxt))
    {
    /* Remove the record from this list.                            */
    op_prg_list_remove (rcv_rec_list, i);
    list_size--; i--;

    /* If the amount of retransmission is greater than        */
    /* the size of the previously "in-order" buffered         */
    /* segment, the segment must be discarded since the   */
    /* retransmitted packet already includes this             */
    /* segment.
*/
    if (tcp_seq_le (seg_rec_ptr->seq + pkt_size, rcv_nxt))
        {
        /* The data just received overlaps the current             */
        /* segment in the out-of-order received list.   */
        /* Destroy the duplicate segment                                   */
        op_pk_destroy (seg_rec_ptr->data_ptr);

        if (tcp_trace_active)
            {
            sprintf (err1, "data. Destroying it. (from seq: %u of size: %d bytes)",
seg_rec_ptr->seq, (TcpT_Size) pkt_size);

            op_prg_odb_print_minor ("Retransmitted TCP segment includes al-
ready stored", err1, OPC_NIL);

            }

        /* Free the segment record pointer              */
        op_prg_mem_free (seg_rec_ptr);
        }
    else
        {
        /* Check for a setting of the PUSH flag in the segment                     */
        /* being buffered. Advance the push flag if necessary, and        */
        /* recognize any packets completed as a result of this              */
        /* segment arrival.
*/
        if (seg_rec_ptr->push_flag == 1)
            {
            /* Setting of the PUSH flag by the remote TCP indicates          */
            /* that the segment forms the data packet boundary.              */
            /* This indicates that this segment will complete a      */
            /* data packet reception.
*/

            complete_pkt_rcvd = OPC_TRUE;
            push_seq = seg_rec_ptr->seq;
            }

        /* Parts of this segment can be buffered (i.e., taken out            */
        /* of out-of-order list and place in the receive buffer.) */
        op_sar_rsgbuf_seg_insert (discard_buf, seg_rec_ptr->data_ptr, seg_rec_ptr-
>seq, 0.0);
```

```
/* Partial overlap? -- remove the extra data contained in */
/* out-of-order segment.
*/
if (tcp_seq_gt (rcv_nxt, seg_rec_ptr->seq))
	{
	/* Obtain the range of bits in the SAR buffer */
	op_sar_rsgbuf_lbl_seg_range_get (discard_buf, seg_rec_ptr->seq,
&bit_index, &bit_count);

	/* Flush the initial bits that have already */
	/* arrived before this segment */
	op_sar_rsgbuf_bits_flush_abs (discard_buf, bit_index, (rcv_nxt -
seg_rec_ptr->seq) * 8);

	/* Print trace message, if enabled. */
	if (tcp_trace_active)
		{
		sprintf (err0, "Flushing %d bytes and transferring %d bytes
to receive buffer.",
			(int) (rcv_nxt - seg_rec_ptr->seq),
			(int) (pkt_size - (rcv_nxt - seg_rec_ptr->seq)));
		}
	}
else
	{
	/* Print trace message, if enabled. */
	if (tcp_trace_active)
		{
		sprintf (err0, "Transferring this segment (size: %d bytes) to
receive buffer.",
			(int) (pkt_size - (rcv_nxt - seg_rec_ptr->seq)));
		}
	}

/* Remove the remaining amount from discard buffer. */
pkptr = op_sar_srcbuf_seg_remove (discard_buf, op_sar_buf_size
(discard_buf));

/* Insert this segment in the receive buffer. */
op_sar_rsmbuf_seg_insert (rcv_buf, pkptr);

/* Update the sequence number of the next */
/* expected segment
*/
rcv_nxt = rcv_buf_seq + (OpT_Sar_Size) (op_sar_buf_size (rcv_buf) / 8);

/* Print trace message, if enabled. */
if (tcp_trace_active)
	{
	sprintf (err1, "Updating RCV.NXT to %u (rcv_buf_seq is %u)",
rcv_nxt, rcv_buf_seq);
	op_prg_odb_print_minor (err0, err1, OPC_NIL);
	}

/* Deallocate memory allocated for the segment record. */
/* (the record has been removed from the segment list.) */
op_prg_mem_free (seg_rec_ptr);
}
		}
	}

/* Check whether this segment filled in all or part in */
/* a sequence space. This happens if the current */
/* receive next variable is larger than it was before */
```

```
		/* processing out-of-order buffer.					*/
		if (tcp_seq_gt (rcv_nxt, rcv_nxt_bfr_ooo_processed))
			{
			/* RFC 2581 (chapter 4.2) states that if a segment	*/
			/* fills sequence space, and immediate ACK should	*/
			/* be generated. Send it.					*/
			tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_ACK);
			ack_sent = OPC_TRUE;
			}
		}

	/* Update the sequence number of the next		*/
	/* expected segment, if not already updated. */
	if (rcv_nxt != rcv_nxt_old)
		{
		if (tcp_trace_active)
			{
			sprintf (err0, "Updating RCV.NXT from %u to %u.", rcv_nxt_old, rcv_nxt);
			op_prg_odb_print_minor (err0, OPC_NIL);
			}
		}
	else
		{
		if (tcp_trace_active)
			{
			sprintf (err0, "RCV.NXT remains at %u.", rcv_nxt);
			op_prg_odb_print_minor (err0, OPC_NIL);
			}
		}

	/* Since a segment has been received in order, it is advancing	*/
	/* rcv.nxt.  Update the sacklist based on the new value which	*/
	/* will be cumulatively acknowledged.					*/
	if (SACK_PERMITTED && (op_prg_list_size (sacklist_ptr->entries) > 0))
		{
		tcp_sacklist_update_newack (rcv_nxt);
		}
	}
else
	{
	/* This is not the next expected segment. Retain	*/
	/* it in the out-of-order list.				*/
	if (out_of_order_seg_pmh_created == OPC_FALSE)
		{
		/* Create a pooled memory type for out-of-order	*/
		/* segment records.					*/
		segrec_pmh = op_prg_pmo_define ("TCP out-of-order segments", sizeof (TcpT_Seg_Record), 32);

		/* Set flag to indicate that a pooled memory has	*/
		/* been created to store out-of-order segments.		*/
		out_of_order_seg_pmh_created = OPC_TRUE;
		}

	/* Create a new out-of-order segment list record.	*/
	new_rec_ptr = (TcpT_Seg_Record *) op_prg_pmo_alloc (segrec_pmh);
	if (new_rec_ptr == OPC_NIL)
		tcp_conn_error ("Unable to create segment reordering record.", OPC_NIL, OPC_NIL);
	new_rec_ptr->seq = seg_seq;
	new_rec_ptr->data_ptr = data_ptr;
	new_rec_ptr->push_flag = (flags & TCPC_FLAG_PSH);

	/* Retain the record, keeping the records sorted	*/
	/* by sequence number.					*/
	list_size = op_prg_list_size (rcv_rec_list);
	for (i = 0; i < list_size; i++)
```

```
				{
				seg_rec_ptr = (TcpT_Seg_Record *) op_prg_list_access (rcv_rec_list, i);
				if (seg_rec_ptr == OPC_NIL)
					{
					tcp_conn_warn ("Unable to get segment record from reordering list.",
						"Skipping to next record in list.", OPC_NIL);
					}
				else
					{
					if (tcp_seq_le (seg_seq, seg_rec_ptr->seq))
						{
						/* Check to see if the segment exists in 	*/
						/* the list.								*/
						if (seg_seq == seg_rec_ptr->seq)
							{
							/* A segment exists in the out-of-order 	*/
							/* list. Ignore the current segment.	*/
							segment_exists = OPC_TRUE;

							/* Print trace message, if enabled.	*/
							if (tcp_trace_active)
								{
								sprintf (err0, "Segment (SEG.SEQ: %u) exists in the out-of-order
list.", seg_seq);

								op_prg_odb_print_minor (err0, "Ignoring this segment.", OPC_NIL);
								}

							/* Deallocate memory allocated to this 	*/
							/* out-of-order received segment.		*/
							op_pk_destroy (new_rec_ptr->data_ptr);
							op_prg_mem_free (new_rec_ptr);
							}

						/* Stop scanning the list.					*/
						break;
						}
					}
				}

			/* If the segment does not already exist in the 			*/
			/* list, insert the record into the list.			*/
			if (segment_exists != OPC_TRUE)
				{
				/* Print trace message, if enabled.	*/
				if (tcp_trace_active)
					{
					sprintf (err0, "Inserting this segment (SEG.SEQ: %u) in out-of-order list.", seg_seq);
					op_prg_odb_print_minor (err0, OPC_NIL);
					}

				/* Insert the segment in an out-of-order received list.	*/
				op_prg_list_insert (rcv_rec_list, new_rec_ptr, i);

			   /* Add the data contained in this record to the sacklist. */
				if (SACK_PERMITTED)
					{
					tcp_sacklist_update_block (seg_seq, seg_seq + seg_len);
					}
				}

			/* Generate an immediate acknowledgment to indicate 		*/
			/* receipt of this out-of-order segment (RFC-2001).	*/
			tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_ACK);
			}
		}
```

```
/* Set the advertized window. If the usage threshold is set        */
/* to zero, then the complete receive buffer is advertized.        */
if (rcv_buf_usage_thresh == 0.0)
	{
	/* Advertise the full window.          */
	rcv_wnd = rcv_buff;

	rcv_wnd = cwnd;
	}
else if (rcv_buf_usage_thresh == -1.0)
	{
	/* Update RCV.WND so the right edge of the receive window */
	/* remains at the same sequence number.                                        */
	if (rcv_wnd > rcv_nxt - rcv_nxt_old)
		rcv_wnd -= (rcv_nxt - rcv_nxt_old);
	else
		rcv_wnd = 0;
	}
else
	{
	/* Receiver-side Silly Window Syndrome avoidance (see RFC 1122,     */
	/* pp. 97-98).  Do not allow the receive window to move in          */
	/* small increments; instead, wait until the right edge can         */
	/* jump a whole MSS (or one half of the buffer size, if that is   */
	/* smaller).  This discourages the sender from succumbing to   */
	/* silly window syndrome and sending many small packets.            */
	rcv_buf_size = (OpT_Sar_Size) (op_sar_buf_size (rcv_buf) / 8.0);

	if (rcv_buff > rcv_buf_size)
		{
		/* Calculate the usage buffer threshold.        */
		rcv_buf_usage_limit = floor (rcv_buff * rcv_buf_usage_thresh);

		/* Find how much buffer is free.      */
		rcv_buf_free = rcv_buff - (rcv_buf_size % rcv_buf_usage_limit);

		if (rcv_buf_free >= MIN (rcv_buff / 2, snd_mss))
			rcv_wnd = rcv_buf_free;
		else
			rcv_wnd = 0;
		}
	else if (op_sar_rsmbuf_pk_count (rcv_buf) == 0)
		{
		/* If no packet can be constructed from the contents    */
		/* of the receive buffer, we must keep the window open.          */
		rcv_wnd = snd_mss;
		}
	else
		{
		/* Buffer space is full and there is no pressing need. */
		rcv_wnd = 0;
		}
	}

/* If the FIN control has been received, and everything                */
/* before it has been acknowledged, we can do all of              */
/* the FIN processing, including acknowledging the FIN                  */
/* and performing the state transition.                                          */
if (rcv_fin_valid && rcv_fin_seq == rcv_nxt)
	{
	if (ev_ptr->event == TCPC_EV_RCV_ACK_OF_FIN ||
		ev_ptr->event == TCPC_EV_RCV_FIN_ACK_OF_FIN)
		{
		ev_ptr->event = TCPC_EV_RCV_FIN_ACK_OF_FIN;
```

```
			}
		else
			{
			ev_ptr->event = TCPC_EV_RCV_FIN;
			}

		if (fin_segment_sent == OPC_TRUE)
			{
			/* A FIN has already been sent. The arrival of this FIN		*/
			/* segment occupies one byte length. Subtract it from */
			/* the sequence number to be sent next.					*/
			snd_nxt--;
			}

		rcv_nxt++;
		}

	/* If anything new has been received, send an acknowledgement.		*/
	/* However, avoid sending an ACK if this process will anyways		*/
	/* be sending ACKs (e.g., from transition FINWAIT to CLOSING		*/
	/* and FINWAIT2 to TIME_WAIT states.), or if an ACK has already	*/
	/* been sent.											*/
	if (tcp_seq_gt (rcv_nxt, rcv_nxt_old) && (ev_ptr->event != TCPC_EV_RCV_FIN) && (ack_sent == OPC_FALSE))
		tcp_ack_schedule ();

	FOUT;
	}

static void
tcp_snd_data_process (void)
	{
	double					old_snd_buf_size;
	double					new_buffer_size;
	Boolean					unacked_data_exist_before_send_started;

	/** The TCP connection may have data in any of	**/
	/** the following two buffers:				**/
	/**   1. Unacknowledged Data Buffer - una_buf	**/
	/**   2. Unsent Data Buffer - snd_buf			**/
	/** Process the contents of these buffers.		**/
	FIN (tcp_snd_data_process ());

	/* If this event caused fast retransmission to take place,		*/
	/* then do not try to send another segment. This is because	*/
	/* one segment has already been sent.					*/
	if (fast_retransmit_occurring == OPC_TRUE)
		FOUT;

	/* Find whether there are any unACKed data at this moment.	*/
	unacked_data_exist_before_send_started = tcp_seq_lt (snd_una, snd_nxt) ? OPC_TRUE : OPC_FALSE;

	/* Find the size of send buffer before sending data.			*/
	old_snd_buf_size = op_sar_buf_size (snd_buf);

	/* If data was transmitted through Fast Retransmit and SACK	*/
	/* is enabled, keep trying to fill in "holes" in receiver's		*/
	/* receive buffer for as long as possible.				*/
	if (SACK_RECOVERY)
		{
		/* Keep resending data as long as there are holes and	*/
		/* the "pipe" allows.							*/
		 while (scoreboard_ptr->more_retran == OPC_TRUE && tcp_snd_total_data_size
(unacked_data_exist_before_send_started) > 0)
			{
			tcp_una_buf_process (unacked_data_exist_before_send_started);
```

```
                }
        }
    else if (tcp_seq_lt (snd_nxt, snd_max))
        {
        /* This indicates that we are performing retransmission          */
        /* (potentially "go-back-n").                                    */

        /* Retransmit data from unacknowledged buffer.                   */
        tcp_una_buf_process (unacked_data_exist_before_send_started);
        }

    if (fin_segment_sent == OPC_TRUE)
        {
        /* Since FIN has already been sent, no new data should           */
        /* be sent out.                                                  */
        FOUT;
        }

    /* If unsent data can be sent (indicated by the fact that            */
    /* the maximum send sequence number is equal to sequence             */
    /* number of next segment to be sent), send it.                      */
    if (tcp_seq_ge (snd_nxt, snd_max))
        {
        tcp_snd_buf_process (unacked_data_exist_before_send_started);
        }

    /** Update delay statistics.                                        **/

    /* Find the size of send buffer after data might have beeb sent.     */
    new_buffer_size = op_sar_buf_size (snd_buf);

    if (new_buffer_size < old_snd_buf_size)
        {
        /* The size of send buffer is less then it was when we */
        /* entered this function. This means that data have              */
        /* been sent. Write and reset delay statistics.                  */

        /* Record statistics.        */
        if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL)
            {
            /* Send window was a reason why packets were not  */
            /* sent before this event was executed. Now, when  */
            /* we sent the data, update the delay statistics.  */
            if (snd_wnd_limit_time > 0.0 &&
                op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->snd_wnd_delay_stathandle) == OPC_TRUE)
                {
                op_stat_write (tcb_ptr->tcp_conn_stat_ptr->snd_wnd_delay_stathandle, op_sim_time() -
snd_wnd_limit_time);

                /* Reset the value of receive-window-limit time.         */
                snd_wnd_limit_time = 0.0;
                }

            /* If congestion window was a reason why packets were        */
            /* not sent before this event was executed. Now,             */
            /* when we sent the data, update the delay statistics.  */
            if (cwnd_limit_time > 0.0 &&
                op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->cwnd_delay_stathandle) == OPC_TRUE)
                {
                op_stat_write (tcb_ptr->tcp_conn_stat_ptr->cwnd_delay_stathandle, op_sim_time() - cwnd_limit_time);
                cwnd_limit_time = 0.0;
                }
            }

        if (new_buffer_size > 0.0)
```

```
				{
				/* There are still some data in the send buffer. They	*/
				/* were not		sent out due to a small window size.		*/
				if (cwnd < snd_wnd)
					{
					/* Data were not sent due to small congestion window.		*/
					/* Record the current time.									*/
					cwnd_limit_time = op_sim_time ();
					}
				else if (cwnd > snd_wnd)
					{
					/* Data were not sent due to small receive window.	*/
					/* Record the current time.								*/
					snd_wnd_limit_time = op_sim_time();
					}
				else
					{
					/* Data were not sent due to small receive and		*/
					/* congestion window.										*/
					/* Record the current time.								*/
					cwnd_limit_time = op_sim_time ();
					snd_wnd_limit_time = op_sim_time();
					}
				}
			}

	/* Write the number of unacknowledged data.		*/
	if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
		op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle) == OPC_TRUE)
		{
		op_stat_write (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle, (double) snd_max - snd_una);
		}

	FOUT;
	}

static void
tcp_una_buf_process (Boolean unacked_data_exist)
	{
	TcpT_Flag					flags = TCPC_FLAG_NONE;
	TcpT_Size					una_data_sent;
	double						retxn_buffer_size;
	TcpT_Size					bytes_to_be_resent;
	Packet*						seg_ptr;
	TcpT_Size					segment_size;
	Packet*						dup_una_seg_ptr;
	double						una_buf_size;
	TcpT_Sackblock*				resend_block_ptr = OPC_NIL;
	int								bytes_to_flush;
	double						dup_una_buf_bytes_size;

	/** Review the contents of the unacknowledged data	**/
	/** buffer (retransmission buffer). Send data when	**/
	/** acknowledgments are received after transmitting	**/
	/** segments resulting from RTO timer expirations	**/
	/** or after Fast Retransmit, if SACK is enabled.		**/
	FIN (tcp_una_buf_process (unacked_data_exist));

	/* Determine the size of the unacknowledged data		*/
	/* buffer. It contains data that has not been			*/
	/* acknowledged for retransmission purposes.				*/
	una_buf_size = op_sar_buf_size (una_buf) / 8;

	/* If it is a case that the currently processed ACK		*/
	/* frees the unacknowledged data buffer, then				*/
```

```
/* return from this procedure immediately. We                    */
/* should now be sending from the "unsent data                   */
/* buffer" (snd_buf), if data is available there.          */
if (una_buf_size == 0.0)
        {
        /* Unacknowledged buffer is empty, so no more         */
        /* retransmission is required. If more                          */
        /* retransmission is enabled then disable it.   */
        if (scoreboard_ptr != OPC_NIL)
                scoreboard_ptr->more_retran = OPC_FALSE;

        FOUT;
        }

/* Check if the duplicate buffer needs to be              */
/* initialized. The initialization is required            */
/* when this routine is called for the first              */
/* time after data/acknowledgment is received.                  */
if (dup_una_buf_init == OPC_FALSE)
        {
        dup_una_buf_init = OPC_TRUE;
        dup_una_buf = op_sar_buf_create (OPC_SAR_BUF_TYPE_RESEGMENT, OPC_SAR_BUF_OPT_DEFAULT);
        }
else
        {
        /* Obtain the size (in bytes) of the duplicate  */
        /* unacknowledged data buffer.                                  */
        retxn_buffer_size = op_sar_buf_size (dup_una_buf);

        if (retxn_buffer_size > 0.0)
                {
                /* Re-initialize the duplicate UNA.BUFF              */
                op_sar_srcbuf_bits_flush (dup_una_buf, 0, retxn_buffer_size);
                }
        }

/* Determine the amount of data that can be sent.       */
/* This amount is based on available window size.      */
bytes_to_be_resent = tcp_snd_total_data_size (unacked_data_exist);

if (bytes_to_be_resent == 0.0)
        {
        FOUT;
        }
else
        {
        /* It may be a case that the unacknowledged data buffer                 */
        /* contains fewer bytes than what can be sent. In this case,      */
        /* rest of data bytes will be accessed from the unsent buffer.    */
        bytes_to_be_resent = (bytes_to_be_resent < una_buf_size) ? bytes_to_be_resent : una_buf_size;

        if (bytes_to_be_resent == 0.0)
                {
                FOUT;
                }

        /* Access the data from the unacknowledged*/
        /* data buffer.                                                              */
        dup_una_seg_ptr = op_sar_srcbuf_seg_access (una_buf, (int) op_sar_buf_size (una_buf));

        /* Place this data in the duplicate buffer         */
        /* created to resend the retransmitted data     */
        op_sar_rsgbuf_seg_insert (dup_una_buf, dup_una_seg_ptr, 0, 0.0);
        }
```

```
/* If we are performing SACK recovery, we need to handle the              */
/* dup_una_buf specially to avoid retransmitting data which has            */
/* been selectively ACKnowledged.                                                      */
if (SACK_RECOVERY)
            {
            /* Find the next segment to be retransmitted.             */
            if (scoreboard_ptr->more_retran)
                        {
                        resend_block_ptr = tcp_scoreboard_find_retransmission (bytes_to_be_resent);
                        if (resend_block_ptr == OPC_NIL)
                                    {
                                    tcp_conn_warn ("Unable to find next block to retransmit in SACK scoreboard.",
                                                "Will wait for retransmission timeout before retransmitting data.", OPC_NIL);
                                    }

                        /* How many bytes need to be sent?                                    */
                        bytes_to_be_resent = resend_block_ptr->end - resend_block_ptr->start;

                        /* There should be data left if more_retran is*/
                        /* true, but double-check to confirm.                        */
                        if (bytes_to_be_resent == 0.0)
                                    {
                           op_prg_mem_free (resend_block_ptr);
                           FOUT;
                           }
                        }
            else
                        {
                        FOUT;
                        }

            /* Determine the number of SACKed bytes falling between      */
            /* end of last retransmission and start of this one.       */

            /* This is the data that has already been selectively     */
            /* ACKed, and so we should not re-tarnsmit them. Thus,          */
            /* we need to flush the data from the dup_una_buf.                */
   bytes_to_flush = resend_block_ptr->start - snd_una;
            }
else
            {
            /* We are not performing SACK-based recovery. Flush            */
            /* the data already sent.                                                         */
            bytes_to_flush = tcp_seq_gt (snd_nxt, snd_una) ? snd_nxt - snd_una : 0;
            }

/* If nothing needs to be resent, return from this function          */
if (bytes_to_be_resent == 0)
            {
            FOUT;
            }

/* Flush the data from the buffer, if any.        */
if (bytes_to_flush > 0)
            {
            if (op_sar_srcbuf_bits_flush (dup_una_buf, 0, bytes_to_flush * 8) == OPC_COMPCODE_FAILURE)
                        {
                        tcp_conn_warn ("Unable to flush bytes from UNA buffer.",
                                    "Data may be unecessarily retransmitted.", OPC_NIL);
                        }
            }

/* Find the size of dup_una_buf.      */
dup_una_buf_bytes_size = op_sar_buf_size (dup_una_buf)/8;
```

```
/* Do not attempt to resend more data than what is in the buffer.            */
bytes_to_be_resent = (bytes_to_be_resent < dup_una_buf_bytes_size) ? bytes_to_be_resent : dup_una_buf_bytes_size;

/* Initialize the variable containing information          */
/* on how much data has been sent.                          */
una_data_sent = 0;

/* If data can be sent, send at using maximum                */
/* segment size boundary. Repeat until no more data  */
/* should be sent.                                                */
while (una_data_sent < bytes_to_be_resent)
	{
	/* Compare the number of bytes to be sent with          */
	/* MSS to determine segment size to be sent.           */
	if ((bytes_to_be_resent - una_data_sent) > snd_mss)
		segment_size = snd_mss;
	else
		segment_size = bytes_to_be_resent - una_data_sent;

	/* Get the segment data.    */
	seg_ptr = op_sar_srcbuf_seg_remove (dup_una_buf, segment_size * 8);

	if (seg_ptr == OPC_NIL)
		{
		tcp_conn_warn ("Unable to get segment from nonempty retransmission buffer.",
			"Attempting to send dataless segment instead.", OPC_NIL);
		segment_size = 0;
		}

	/* This is like a retransmitted segment and carries       */
	/* an ACK (all but SYN segment carry this.)                  */
	flags |= TCPC_FLAG_ACK;

	/* If this is not an MSS-sized packet, it must be          */
	/* on an application packet boundary.  Set PUSH.       */
	if (segment_size < snd_mss)
		flags |= TCPC_FLAG_PSH;

	/* If this falls in outgoing urgent data, set URG.         */
	if (snd_up_valid && tcp_seq_ge (snd_up, snd_una))
		{
		flags |= TCPC_FLAG_URG;
		}

	/* Last segment in the stream is FIN.           */
	if (snd_fin_valid && (snd_una + segment_size == snd_fin_seq))
		{
		flags |= TCPC_FLAG_FIN;
		}

	/* Since ECT code point should not be set for retransmitted      */
	/* segments (section 6.1.5 in RFC-3168), indicate it to IP.       */
	tcp_ecn_request_to_ip (0);

	/* Send the packet.            */
	tcp_seg_send (seg_ptr, SACK_RECOVERY ? resend_block_ptr->start : snd_nxt, flags);

	/* Write TCP retransmission statistics.          */
	tcp_conn_retrans_stat_write ();

	if (snd_fin_valid && (snd_una + segment_size == snd_fin_seq))
		{
		segment_size++;
		snd_nxt++;
		bytes_to_be_resent++;
```

```
			}

		/* Update the maximum send sequence number. If we are in SACK recovery,	*/
		/* do not need to do so because the next packet to be retransmitted		*/
		/* will always be selected based on exactly what data has been			*/
		/* retransmitted. Also, SACK has its own congestion control algorithm */
		/* so there is no need to resend the congestion window.						*/
		if (!SACK_RECOVERY)
			{
			/* Update the maximum send sequence number.				*/
			snd_nxt += segment_size;

			/* If no data is in transit and nothing's been sent		*/
			/* for a long time, slow-start to avoid congestion.		*/
			if (snd_nxt == snd_una && (op_sim_time () - last_snd_time > current_rto))
				{
				/* Reset the congestion window to the inital */
				/* window size used for slow-start.					*/
				ssthresh = MAX (cwnd, ssthresh);
				cwnd = initial_window_size;

				/* Record the congestion window statistics.	*/
				tcp_cwnd_stat_update ();
				}
			}

		/* Update the last send time.							*/
		last_snd_time = op_sim_time ();

		/* Set a retransmission timeout, if necessary.		*/
		tcp_retrans_timeout_check_and_schedule ();

		/* If the round-trip time measurement isn't already	*/
		/* active, measure the RTT for this segment.		*/
		if (!rtt_active)
			{
			rtt_active = 1;
			rtt_base_time = op_sim_time ();
			rtt_seq = snd_nxt;
			}

		/* Update send sequence variables.					*/
		una_data_sent += segment_size;
		}

	/* If using SACK, free memory used to determine what		*/
	/* block should be retransmitted.							*/
	if (resend_block_ptr != OPC_NIL)
		{
		op_prg_mem_free (resend_block_ptr);
		}

	/* If everything up to the FIN has been sent, send the */
	/* FIN as well.													*/
	if (snd_fin_valid && snd_nxt == snd_fin_seq && fin_segment_sent == OPC_FALSE)
		{
		/* Send the FIN segment, and set the state variable	*/
		/* to indicate that it has been sent.				*/
		tcp_seg_send (TCPC_DATA_NONE, snd_nxt++, TCPC_FLAG_ACK | TCPC_FLAG_FIN);
		fin_segment_sent = OPC_TRUE;

		/* Update the maximum send sequence number. In an*/
		/* ideal case, when the system has not observed any	*/
		/* retransmissions, SND.NXT and SND.MAX are equal.		*/
		/* Refer page 808 (TCP/IP Illustrated Volume 2)		*/
```

```
			if (tcp_seq_ge (snd_nxt, snd_max))
				snd_max++;

			/* Set a retransmission timeout, if necessary		*/
			/* for FIN segments.								*/
			tcp_retrans_timeout_check_and_schedule ();
			}

		FOUT;
		}

static void
tcp_snd_buf_process (Boolean unacked_data_exist)
		{
		Packet*				data_pkt_ptr;
		TcpT_Flag			flags = TCPC_FLAG_NONE;
		Boolean				seg_sent_flag = OPC_FALSE;

		/** Review the contents of the unsent data	**/
		/** buffer.  Send data when possible.			**/
		FIN (tcp_snd_buf_process (unacked_data_exist));

		/* Determine whether any segments should be sent.	*/
		/* If so, send one, and repeat until no more data		*/
		/* should be sent.								*/
		while ((seg_len = tcp_snd_data_size (unacked_data_exist)) > 0)
			{
			/* Get the segment data. */
			data_pkt_ptr = op_sar_srcbuf_seg_remove (snd_buf, seg_len * 8);
			if (data_pkt_ptr == OPC_NIL)
				tcp_conn_error ("Unable to get segment from send buffer.", OPC_NIL, OPC_NIL);

			/* Determine flags for the outgoing segment. */
			flags |= TCPC_FLAG_ACK;

			/* If this falls in outgoing urgent data, set URG. */
			if (snd_up_valid && tcp_seq_ge (snd_up, snd_nxt))
				flags |= TCPC_FLAG_URG;

			/* If sending this segment empties the send buffer, we		*/
			/* are at an application packet boundary.  Set PUSH.	*/
			if (op_sar_buf_size (snd_buf) == 0)
				flags |= TCPC_FLAG_PSH;

			/* If no data is in transit and nothing's been sent		*/
			/* for a long time, slow-start to avoid congestion.		*/
			if ((snd_nxt == snd_una) && (op_sim_time () - last_snd_time > current_rto))
				{
				/* Reset the congestion window to one segment.		*/
				ssthresh = MAX (cwnd, ssthresh);

				/* Set the congestion window to inital window		*/
				/* size used for slow-start.						*/
				cwnd = initial_window_size;

				/* Update the congestion window statistic. */
				/* Record the congestion window statistics.	*/
				tcp_cwnd_stat_update ();
				}
			last_snd_time = op_sim_time ();

			/* Add this segment to the retransmission queue.		*/
			/* Set a retransmission timeout, if necessary.		*/
			op_sar_rsgbuf_seg_insert (una_buf, op_pk_copy (data_pkt_ptr), 0, 0);
```

```
		/* If the round-trip time measurement isn't already	*/
		/* active, measure the RTT for this segment.		*/
		if (!rtt_active)
			{
			rtt_active = 1;
			rtt_base_time = op_sim_time ();
			rtt_seq = snd_nxt;
			}

		/* Since ECT code point should be set for data		*/
		/* segments, indicate this to IP.  Refer to		*/
		/* 6.1.2 in RFC-3168 for more details.				*/
		tcp_ecn_request_to_ip (1);

		/* Send the segment to the power (IP) layer. */
		tcp_seg_send (data_pkt_ptr, snd_nxt, flags);

		/* Enable the segment sent flag to schedule	*/
		/* the retransmission timer.						*/
		seg_sent_flag = OPC_TRUE;

		/* Update send sequence variables.				*/
		snd_nxt += seg_len;

		/* Update the maximum send sequence number.		*/
		if (tcp_seq_ge (snd_nxt, snd_max))
			{
			snd_max += seg_len;
			}

		/* Write the number of unacknowledged data.		*/
		if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
			op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle) == OPC_TRUE)
			{
			op_stat_write (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle, (double) snd_max - snd_una);
			}

		/* If SACK-based recovery is currently			*/
		/* occuring, increment the pipe by the			*/
		/* amount of data being sent.						*/
		if (SACK_RECOVERY)
		   {
		   pipe += seg_len;
		   }
		}

	/* If everything up to the FIN has been sent, send the FIN as well. */
	if (snd_fin_valid && snd_nxt == snd_fin_seq && fin_segment_sent == OPC_FALSE)
		{
		/* Update the maximum send sequence number. In an*/
		/* ideal case, when the system has not observed any	*/
		/* retransmissions, SND.NXT and SND.MAX are equal.		*/
		/* Refer page 808 (TCP/IP Illustrated Volume 2)			*/
		if (tcp_seq_ge (snd_nxt, snd_max))
			snd_max++;

		/* Send the FIN segment, and set the state variable	*/
		/* to indicate that it has been sent.					*/
		tcp_seg_send (TCPC_DATA_NONE, snd_nxt++, TCPC_FLAG_ACK | TCPC_FLAG_FIN);
		fin_segment_sent = OPC_TRUE;
		}

	/* Scehdule a retransmission timer for SYN, DATA or FIN segments	*/
	/* This will take care of retransmitting SYN, DATA or FIN segments	*/
	/* if they itself or their ACK's got lost or discarded				*/
```

```
	if (seg_sent_flag == OPC_TRUE)
		tcp_retrans_timeout_check_and_schedule ();

	FOUT;
	}

static TcpT_Size
tcp_snd_total_data_size (Boolean unacked_data_exist)
	{
	TcpT_Size		total_wnd, avail_wnd, option_size;
	int			result = 0;
	int			num_seg;

	/** Determine the size of the next data segment that should be **/
	/** sent.  This is normally the minimum of the available remote	**/
	/** window size, the available congestion window, the number of	**/
	/** bytes of data in the unsent buffer, and the maximum segment	**/
	/** size.  If Nagle algorithm is being used, however, no		**/
	/** no segment smaller than the maximum size will be sent if **/
	/** there is outstanding unacknowledged data.				**/
	/** Returns the segment size, or zero if none should be sent.	**/
	FIN (tcp_snd_total_data_size (unacked_data_exist));

	/* First, handle the case in which retransmission due to the	*/
	/* Fast Retransmit threshold being reached is currently		*/
	/* occurring. In this case, always send one MSS packet.			*/
	if (fast_retransmit_occurring == OPC_TRUE)
		{
		FRET (snd_mss);
		}

	/* Next, handle the case in which we are currently performing */
	/* retransmission due to Fast Retransmit when SACK is being		*/
	/* used -- use Sally Floyd's Pipe Algorithm.  This estimates	*/
	/* how much data is currently out in the "pipe" between sender*/
	/* and receiver, and only retransmits if the pipe contains		*/
	/* less data than allowed by the congestion window.			*/

	/* If there is no more data to retransmit, data will be pulled	*/
	/* from the send buffer.  Calculation of data size in this case	*/
	/* will be handled below.							*/
	if (SACK_RECOVERY && scoreboard_ptr->more_retran)
		{
		/* First check how whether congestion window allows any data to be sent.	*/
		if (cwnd <= pipe)
			FRET (0);

		/* Second, check whether remote receive buffer is able to accept data.	*/
		if (snd_wnd <= snd_max - snd_una)
			FRET (0);

		/* The segment that will be created can carry TS and	*/
		/* SACK option fields in the TCP header. These fields		*/
		/* decrease the amount of space available for data.		*/
		/* Find how much space will be consumed by TCP options.	*/
		option_size = tcp_conn_option_size_get ();

		/* If a full segment can be sent, send it.	*/
		if (pipe + snd_mss <=  cwnd)
			{
			result = snd_mss;
			}
		else
			{
			/* Otherwise send whatever is allowed by congestion and receive windows.	*/
```

```
				result = cwnd - pipe;
				}

			/* Decrease the amount of available data by the size of option field.	*/
			if ((result + option_size) > snd_mss)
				result = result > option_size ? snd_mss - option_size : 0;

			FRET (result);
			}

		/* The total window size is the minimum of the remote window	*/
		/* (as advertised) and the congestion window.			*/
		total_wnd = MIN (snd_wnd, cwnd);

		/* The available window size is the total window size minus the	*/
		/* amount of data already sent in this window.			*/
		if (tcp_seq_gt ((snd_una + total_wnd), snd_nxt))
			{
			avail_wnd = snd_una + total_wnd - snd_nxt;
			}
		else
			{
			/* The remote window has shrunk; do not use negative window size. */
			avail_wnd = 0;
			}

		/* Second test (Nagle algorithm): don't send small segments	*/
		/* if there is outstanding unacknowledged data.			*/
		/* There are 2 versions of Nagle algorithm:			*/
		/* 1. RFC version which specifies that a small segment		*/
		/*    should not be sent if there are any outstanding data	*/
		/*    at the time of segment creation (e.i. now).		*/
		/*    This is a per-segment implementation.			*/
		/* 2. Optimized version which prevents small packets to be	*/
		/*    sent only if there was outstanding data before the	*/
		/*    sending process started (e.i. before a first segment	*/
		/*    in this sequence of sending data was created).		*/
		/*    This is a per-send implementation.			*/
		/* The model implements both versions.				*/
		if (nagle_support != TcpC_Nagle_Disabled)
			{
			if (((nagle_support == TcpC_Nagle_Enabled_RFC) && (tcp_seq_gt (snd_nxt, snd_una)))
					|| ((nagle_support == TcpC_Nagle_Enabled_Optimized) && (unacked_data_exist == OPC_TRUE)))
				{
				/* No data can be sent. Start clock used to calculate Nagle */
				/* delay statistics. However, do it only if the clock has	*/
				/* not yet started, since otherwise there is a packet	*/
				/* which could not be sent in the send buffer		*/
				if (nagle_limit_time == 0.0)
					nagle_limit_time = op_sim_time ();

				/* No packet should be sent, return packet size 0.	*/
				FRET (0);
				}
			else
				{
				/* Packet can be sent out, write the statistics and reset	*/
				/* the value of the last time when a packet could not	*/
				/* be sent due to the limitations given by Nagle's algorithm.	*/
				if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
					nagle_limit_time > 0.0 &&
					op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->nagle_delay_stathandle) == OPC_TRUE)
					{
					op_stat_write (tcb_ptr->tcp_conn_stat_ptr->nagle_delay_stathandle, op_sim_time() - nagle_limit_time);
					nagle_limit_time = 0.0;
```

```
                }
            }
        }

    /* The segment that will be created can carry TS and  */
    /* SACK option fields in the TCP header. These fields             */
    /* decrease the amount of space available for data.               */
    /* Find how much space will be consumed by TCP options.      */
    option_size = tcp_conn_option_size_get ();

    /* Decrease the amount of available data by the size of option field.       */
    if (option_size > 0)
        {
        if (option_size < avail_wnd)
            {
            /* Get the number of segments that will be sent.         */
            num_seg = (int) (ceil (avail_wnd/snd_mss));

            /* The available window should be decreased to take  */
            /* into account the "option" overhead per segemnt.     */
            avail_wnd = avail_wnd - num_seg * option_size;
            }
        else
            {
            /* Options would consume the complete available window.      */
            /* Do not send anything.                                     */
            FRET (0);
            }
        }

    /* If neither of the above cases is true, use the minimum. */
    FRET (avail_wnd);

    }


static TcpT_Size
tcp_snd_data_size (Boolean unacked_data_exist)
    {
    TcpT_Size           total_wnd, avail_wnd, option_size, in_flight;
    long int        snd_data_size = -1;
    char                str [128];
    SimT_Sar_Size   snd_buf_size;

    /** Determine the size of the next data segment that should be **/
    /** sent.  This is normally the minimum of the available remote             **/
    /** window size, the available congestion window, the number of         **/
    /** bytes of data in the unsent buffer, and the maximum segment          **/
    /** size.  If Nagle algorithm is enabled, no segment smaller     **/
    /** than the maximum segment size will be sent if there is                  **/
    /** outstanding unacknowledged data.                                         **/
    /**
        **/
    /** Lastly, no data should be sent if a full segment cannot be    **/
    /** sent given that send buffer contains more data than MSS     **/
    /** (Silly Window Syndrome avoidance, RFC 1122).                          **/
    /**
        **/
    /** Returns the segment size, or zero if none should be sent.     **/
    FIN (tcp_snd_data_size (unacked_data_exist));

    /* First, handle the case in which retransmission due to the        */
    /* Fast Retransmit threshold being reached is currently                   */
    /* occurirng. In this case, always send one MSS packet.                          */
    if (fast_retransmit_occurring == OPC_TRUE)
```

```
		{
		snd_data_size = snd_mss;
		}
	else
		{
		snd_buf_size = op_sar_buf_size (snd_buf) / 8;

		/* Next, handle the case in which we are currently performing */
		/* retransmission due to Fast Retransmit when SACK is being		*/
		/* used -- use Sally Floyd's Pipe Algorithm.  This estimates	*/
		/* how much data is currently out in the "pipe" between sender*/
		/* and receiver, and only retransmits if the pipe contains		*/
		/* less data than allowed by the congestion window.				*/

		/* If there is no more data to retransmit, data will be pulled	*/
		/* from the send buffer.  Calculation of data size in this case	*/
		/* will be handled below.											*/
		if ((snd_buf_size != 0) && SACK_RECOVERY)
			{
			/* Calculate how much data is in-flight.	*/
			in_flight = snd_max - snd_una;

			/* Do not send any data if the congestion window does not allow.		*/
			if ((cwnd <= pipe) || (snd_wnd <= in_flight))
				FRET (0);

			/* Determine how much can be sent based on the current CWND		*/
			/* and in-flight data.												*/
			avail_wnd = MIN (cwnd - pipe, snd_wnd - in_flight);

			if (avail_wnd >= snd_mss)
				{
				snd_data_size = snd_mss;
				}
			else
				{
				/* We have less than MSS-size data to send -- we should		*/
				/* be doing Silly Window Syndrome avoidance.					*/

				/* If we have more data to send (meaning send buffer is not	*/
				/* empty), we should not send less than MSS-size segments.	*/
				if (snd_buf_size > (SimT_Sar_Size) avail_wnd)
					{
					/* Set the packet size to 0. This value will then be returned			*/
					/* to the calling function as a packet size, and no data will be sent.	*/
					avail_wnd = 0;
					}

				snd_data_size = avail_wnd;
				}
			}
		else
			{
			/* SACK is not used. Determine the number of bytes of data available to be sent.*/
			if (snd_buf_size == 0)
				{
				snd_data_size = 0;
				avail_wnd = 0;
				}
			else
				{
				/* The total window size is the minimum of the remote window		*/
				/* (as advertised) and the congestion window.						*/
				total_wnd = MIN (snd_wnd, cwnd);
```

```
/* The available window size is the total window size minus the           */
/* amount of data already sent in this window.                            */
if (tcp_seq_gt ((snd_una + total_wnd), snd_nxt))
	{
	avail_wnd = snd_una + total_wnd - snd_nxt;

	/* Silly Window Syndrome avoidance (RFC 1122): If we have more  */
	/* data to send (meaning send buffer is not empty), we should  */
	/* not send packets with a size less than MSS.                          */
	if ((avail_wnd < snd_mss) && (snd_buf_size > (SimT_Sar_Size) avail_wnd))
		{
		/* Set the available window to 0. This value will then be returned           */
		/* to the calling function as a packet size, and no data will be sent.       */
		avail_wnd = 0;
		snd_data_size = avail_wnd;
		}
	}
else
	{
	/* The remote window has shrunk; do not use negative window size. */
	avail_wnd = 0;
	}

/* Has the send data size been determined?   */
if (snd_data_size == -1)
	{
	/* Check two common cases first. */

	/* First test: if both the available window and the data buffer    */
	/* are larger than MSS, send a maximum size segment.                 */
	if ((avail_wnd >= snd_mss) && (snd_buf_size >= (SimT_Sar_Size) snd_mss))
		{
		snd_data_size = snd_mss;
		}
	else
		{
		/* Second test (Nagle algorithm): don't send small segments     */
		/* if there is outstanding unacknowledged data.                     */
		/* There are 2 versions of Nagle algorithm:                         */
		/* 1. RFC version which specifies that a small segment             */
		/*    should not be sent if there are any outstanding data  */
		/*    at the time of segment creation (e.i. now).                   */
		/*    This is a per-segment implementation.                         */
		/* 2. Optimized version which prevents small packets to be      */
		/*    sent only if there was outstanding data before the         */
		/*    sending process started (e.i. before a first segment       */
		/*    in this sequence of sending data was created).            */
		/*    This is a per-send implementation.                             */
		/* The model implements both versions.
*/
		if (nagle_support != TcpC_Nagle_Disabled)
			{
			if (((nagle_support == TcpC_Nagle_Enabled_RFC) && (tcp_seq_gt (snd_nxt,
snd_una)))
				|| ((nagle_support == TcpC_Nagle_Enabled_Optimized) &&
(unacked_data_exist == OPC_TRUE)))
				{
				/* No data can be sent. Start clock used to calculate Nagle */
				/* delay statistics. However, do it only if the clock has          */
				/* not yet started, since otherwise there is a packet             */
				/* which could not be sent in the send buffer
	*/
				if (nagle_limit_time == 0.0)
					nagle_limit_time = op_sim_time ();
```

```
								/* No packet should be sent, return packet size 0.	*/
								snd_data_size = 0;
								}
							else
								{
								/* Packet can be sent out, write the statistics and reset
	*/
								/* the value of the last time when a packet could not
	*/
								/* be sent due to the limitations given by Nagle's algorithm.	*/
								if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
									nagle_limit_time > 0.0 &&
									op_stat_valid (tcb_ptr->tcp_conn_stat_ptr-
>nagle_delay_stathandle) == OPC_TRUE)
									{
									op_stat_write (tcb_ptr->tcp_conn_stat_ptr-
>nagle_delay_stathandle, op_sim_time() - nagle_limit_time);
									nagle_limit_time = 0.0;
									}
								}
							}
						}

					/* If neither of the above cases is true, use the minimum. */
					if (snd_data_size == -1)
						{
						snd_data_size = MIN (avail_wnd, snd_buf_size);
						}
					}
				}
			}

		/* Finally, the segment that will be created can carry	*/
		/* TS and SACK option fields in the TCP header.	These	*/
		/* fields decrease the amount of space available for	*/
		/* data.												*/
		if (snd_data_size > 0)
			{
			option_size = tcp_conn_option_size_get ();

			if (avail_wnd + option_size > snd_mss)
				snd_data_size = snd_data_size > option_size ? snd_mss - option_size : 0;
			}
		}

	/* Generate a trace message if appropriate. */
	if (tcp_trace_active)
		{
		sprintf (str, "Send data size = %d", (int) snd_data_size);
		op_prg_odb_print_minor (str, OPC_NIL);
		}

	FRET ((TcpT_Size) snd_data_size);
	}

static void
tcp_rcv_buf_process (void)
	{
	Ici*			ici_ptr;
	Packet*			pk_ptr;
	OpT_Packet_Size		pk_size;
	char			str [128];
	int				num_pks_avail, urgent = 0;
	TcpT_Size		rcv_buf_free;
	double			time_delay;
```

```
OpT_Packet_Size                     byte_thru;
int                                 rcv_buf_size;
int                                 rcv_buf_usage_limit;
TcpT_Seq                    old_rcv_buf_seq;

/** Forward data from the received data buffer to the application.                    **/
/** As in many TCP implementations, data is forwarded to the application          **/
/** as soon as it is available and has been requested; PUSH is ignored.           **/
FIN (tcp_rcv_buf_process ());

/* Store the current value of the sequence number of the next expected segment. */
old_rcv_buf_seq = rcv_buf_seq;

/* Keep forwarding packets to the application until we run      */
/* out of RECEIVE requests or urgent data or until we run       */
/* out of complete packets in the received data buffer.         */
while ((num_pks_req > 0 || rcv_up_valid) && (num_pks_avail = op_sar_rsmbuf_pk_count (rcv_buf)) > 0)
        {
        /* Remove the packet from the received data buffer.             */
        pk_ptr = op_sar_rsmbuf_pk_remove (rcv_buf);

        if (pk_ptr == OPC_NIL)
                tcp_conn_error ("Unable to get application packet from received data buffer.",
                        OPC_NIL, OPC_NIL);

        /* Compute the interval spent by the packet within the TCP layer. */
        time_delay = op_sim_time () - op_pk_stamp_time_get (pk_ptr);
        op_stat_write (tcp_delay_handle, time_delay);
        op_stat_write (tcp_global_delay_handle, time_delay);

        /*      Calculate Statistics. In the model code (below), we     */
        /*      will record the "<units>/sec" statistic in <units>       */
        /*      where <units> can be "bytes" or "packets". OPNET's */
        /*      statistics "capture mode" feature will be used to       */
        /*      record it in <units>/sec.                                                       */
        pk_size = (OpT_Packet_Size) op_pk_total_size_get (pk_ptr);

        /*      Record thruput just as packet size or just one          */
        /*      packet for proper computation of sum/time based         */
        /*      statistics.                                                                                                     */
        byte_thru = (pk_size / 8);

        /* Record statistics local to each connection.          */
        if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL)
                {
                if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->ete_delay_stathandle) == OPC_TRUE)
                        op_stat_write (tcb_ptr->tcp_conn_stat_ptr->ete_delay_stathandle, time_delay);

                if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->thruput_packets_stathandle) == OPC_TRUE)
                        op_stat_write (tcb_ptr->tcp_conn_stat_ptr->thruput_packets_stathandle, 1.0);

                if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_sec_stathandle) == OPC_TRUE)
                        {
                        op_stat_write (tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_sec_stathandle, byte_thru);
                        op_stat_write (tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_sec_stathandle, 0.0);
                        }

                if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_stathandle) == OPC_TRUE)
                        {
                        op_stat_write (tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_stathandle, byte_thru);
                        }

                if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->thruput_packets_sec_stathandle) == OPC_TRUE)
                        {
                        op_stat_write (tcb_ptr->tcp_conn_stat_ptr->thruput_packets_sec_stathandle, 1.0);
```

```
					op_stat_write (tcb_ptr->tcp_conn_stat_ptr->thruput_packets_sec_stathandle, 0.0);
					}
				}

			/* Record local statistics (for all connections).		*/
			op_stat_write (byte_thru_handle,   (double) byte_thru);
			op_stat_write (packet_thru_handle, 1.0);

			op_stat_write (byte_sec_thru_handle,   (double) byte_thru);
			op_stat_write (packet_sec_thru_handle, 1.0);

			/* Record extra data-points to enable proper computation of			*/
			/* the "sum/time" based statistics.								*/
			op_stat_write (byte_sec_thru_handle,   0.0);
			op_stat_write (packet_sec_thru_handle, 0.0);

			/* Generate a trace message if appropriate. */
			if (tcp_trace_active)
				{
				sprintf (str, "Packet id (" SIMC_PK_ID_FMT "), tree id (" SIMC_PK_ID_FMT "), size (%d bytes), sequence num-
ber (%u)",
						op_pk_id (pk_ptr), op_pk_tree_id (pk_ptr), (int) byte_thru, rcv_buf_seq);
				op_prg_odb_print_minor ("Forwarding packet to application:", str, OPC_NIL);
				}

			/* Determine whether the urgent pointer applies to this packet. */
			if (rcv_up_valid && tcp_seq_ge (rcv_up, rcv_buf_seq))
				urgent = 1;

			/* Update the sequence number marker. */
			rcv_buf_seq += byte_thru;

			/* One RECEIVE command is being satisfied. */
			num_pks_req--;
			if (num_pks_req < 0)
				num_pks_req = 0;

			/* Forward the packet to the application.  Generate	*/
			/* only one interrupt for this batch of		packets by		*/
			/* sending all packets quietly except the last one.	*/
			if ((num_pks_req == 0 && !rcv_up_valid) || num_pks_avail == 1)
				{
				/* Install the ICI used for connection identification. */
				ici_ptr = op_ici_create ("tcp_status_ind");
				if ((ici_ptr == OPC_NIL) ||
					(op_ici_attr_set (ici_ptr, "conn_id", tcb_ptr->conn_id) == OPC_COMPCODE_FAILURE) ||
					(op_ici_attr_set (ici_ptr, "status", TCPC_IND_SEG_FWD) == OPC_COMPCODE_FAILURE))
					{
					tcp_conn_error ("Unable to successfully create the status indication ICI.", OPC_NIL, OPC_NIL);
					}

				/* Set the urgent flag if any of these packets were urgent. */
				if (op_ici_attr_set (ici_ptr, "urgent", urgent) == OPC_COMPCODE_FAILURE)
					tcp_conn_warn ("Unable to set urgent field in status indication ICI.",
						OPC_NIL, OPC_NIL);

				/* Install the ICI and send the packet. */
				op_ici_install (ici_ptr);
				op_pk_send (pk_ptr, tcb_ptr->strm_index);
				}
			else
				{
				op_pk_send_quiet (pk_ptr, tcb_ptr->strm_index);
				}
```

```
		/* If all urgent data has been forwarded, the urgent pointer is no longer valid. */
		if (rcv_up_valid && tcp_seq_gt (rcv_buf_seq, rcv_up))
			rcv_up_valid = 0;
		}

	/* Set the advertized window. If the usage threshold is set	*/
	/* to zero, then the complete receive buffer is advertized.	*/
	/* Otherwise use the value that was previously calculated	*/
	/* in tcp_seg_receive (0 function.						*/
	if (rcv_buf_usage_thresh == 0.0)
		{
		/* Advertise the full window, as the threshold setting */
		/* indicates that the data is immediately forwarded to */
		/* the higher layer.										*/
		rcv_wnd = rcv_buff;

		rcv_wnd = cwnd;
		}
	else if (rcv_buf_usage_thresh == -1.0)
		{
		rcv_wnd = rcv_wnd + (rcv_buf_seq - old_rcv_buf_seq);
		}
	else
		{
		rcv_buf_size = (OpT_Sar_Size) (op_sar_buf_size (rcv_buf) / 8.0);

		if (rcv_buff > rcv_buf_size)
			{
			/* Calculate the usage buffer threshold.	*/
			rcv_buf_usage_limit = floor (rcv_buff * rcv_buf_usage_thresh);

			/* Find how much buffer is free.	*/
			rcv_buf_free = rcv_buff - (rcv_buf_size % rcv_buf_usage_limit);

			if (rcv_buf_free >= MIN (rcv_buff / 2, snd_mss))
				rcv_wnd = rcv_buf_free;
			else
				rcv_wnd = 0;
			}
		else if (op_sar_rsmbuf_pk_count (rcv_buf) == 0)
			{
			/* If no packet can be constructed from the contents	*/
			/* of the receive buffer, we must keep the window open.	*/
			rcv_wnd = snd_mss;
			}
		else
			{
			/* Buffer space is full and there is no pressing need. */
			rcv_wnd = 0;
			}
		}

	FOUT;
	}

static void
tcp_ack_schedule (void)
	{
	/**	Based on the type of delayed acknowledgment scheme used	**/
	/**	for which this TCP connection process, call the			**/
	/** appropriate acknowledgment scheduling procedure.			**/
	FIN (tcp_ack_schedule ());

	switch (tcp_del_ack_scheme)
```

```
        {
        case TcpC_Purely_Timer_Based:
            {
            /*      This TCP implementation sends a delayed ACK             */
            /*      every "max_ack_delay" time interval. This          */
            /*      acknowledgment is for the data received upto       */
            /*      the timer expiration.                                        */
            tcp_clock_based_ack_schedule ();

            break;
            }

        case TcpC_Segment_And_Timer_Based:
            {
            /*      This TCP implementation sends an acknowledgment  */
            /*      if any one of the following conditions is met:     */
            /*        - if no ack was sent for the previous            */
            /*          segment received.                                        */
            /*        - if a segment was received, and no other        */
            /*          segment arrives within the "max_ack_delay"     */
            /*          interval.                                                        */
            tcp_seg_and_timer_based_ack_schedule ();

            break;
            }

        default:
            {
            break;
            }
        }

    FOUT;
    }

static void
tcp_seg_and_timer_based_ack_schedule (void)
  {
    /** This procedure either schedules the sending of a dataless ACK     **/
    /**     or sends one. Note that we do not immediately send an ACK upon       **/
    /**     receiving a packet; instead, we send an acknowledgment when any      **/
    /**     one of the following conditions are met:                                    **/
    /**       - if no ack was sent for the previous segment received.            **/
    /**       - if a segment was received, and no other segment arrives          **/
    /**         within the "max_ack_delay" time interval.                   **/
    FIN (tcp_seg_and_timer_based_ack_schedule ());

    /*      Increment the number of segments received without sending an       */
    /*      acknowledgment.
    */
    tcp_segments_rcvd_without_sending_ack++;

    /*      If there are two segment (including the one which we just     */
    /*      received) for which an ACK has not been sent, send one.            */
    if (tcp_segments_rcvd_without_sending_ack == tcp_parameter_ptr->ack_frequency)
        {
        /*      The previous segment reception did not generate an ACK, and          */
        /*      is the max-unacked-segment that we've received for which no ACK      */
        /*      has been sent yet. Reset the variable to count the number of    */
        /*      segments received without sending an ACK.                            */
        tcp_segments_rcvd_without_sending_ack = 0;

        /* Send a dataless acknowledgment.         */
        tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_ACK);
```

```
		/*	Cancel the timer pending to send a dataless acknowledgment.	*/
		if (op_ev_valid (ack_evh) == OPC_TRUE)
			op_ev_cancel (ack_evh);
		}
	else
		{
		/* Set the timer to send the dataless ACKs. This	*/
		/* function will schedule an ACK only if none is	*/
		/* already scheduled.					*/
		tcp_clock_based_ack_schedule ();
		}

	FOUT;
	}

static void
tcp_clock_based_ack_schedule (void)
	{
	double			next_ack_del_time = 0.0;
	double			current_time = 0.0;

	/** Schedule the sending of a dataless ACK.  Note that we do not	**/
	/**	immediately send an ACK upon receiving a packet; instead, we	**/
	/**	wait for a limited time for an outgoing segment on which we can	**/
	/**	"piggyback" the ACK.  If no such segment is sent before the	**/
	/**	timer expires, a dataless ACK will be sent.			**/
	FIN (tcp_clock_based_ack_schedule ());

	/* Schedule an ACK only if none is already scheduled.		*/
	if (!op_ev_valid (ack_evh))
		{
		/*	Store the value of the current time.			*/
		current_time = op_sim_time ();

		/* Compute the time at which the next delayed ACK will be sent,	*/
		/* if no data is sent before this duration.  The following	*/
		/* algorithm tries to emulate the real world implementation	*/
		/* that this timer goes off at fixed points in time -- every	*/
		/* "max_ack_delay" time relative to when the system starts.	*/
		if (max_ack_delay == 0.0)
			{
			/* The requesting entity wishes to send the acknowledgment	*/
			/* immediately. Send a dataless acknowledgment.		*/
			tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_ACK);
			}
		else
			{
			/* The time at which the next delayed ACK should be sent	*/
			/* is taken as the next "max_ack_delay" timer expiration.	*/
			next_ack_del_time = Tcp_Fasttimo_Next_Timeout_Time_Obtain (current_time, max_ack_delay);

			/* Schedule the acknowledgment.				*/
			ack_evh = op_intrpt_schedule_call (next_ack_del_time, 0, tcp_timeout_delay_ack, OPC_NIL);
			if (op_ev_valid (ack_evh) == OPC_FALSE)
				{
				tcp_conn_warn ("Unable to schedule dataless acknowledgement.",
					"Received data will not be acknowledged until a segment is sent.",
					OPC_NIL);
				}
			}
		}

	FOUT;
	}
```

```
static void
tcp_fin_schedule (void)
	{
	/** No more SEND commands will be received from the application.	**/
	/** A FIN segment should be sent; note, however, that the FIN can't	**/
	/** be sent until all outgoing data has been sent.  Set flags to	**/
	/** indicate that the FIN should be sent after the pending data. **/
	FIN (tcp_fin_schedule ());

	snd_fin_valid = 1;
	snd_fin_seq = snd_max + (OpT_Sar_Size) (op_sar_buf_size (snd_buf) / 8);

	/* Since there is no more outgoing data, don't delay	*/
	/* sending of ACK's from this time onwards.				*/
	max_ack_delay = 0.0;

	if (op_sar_buf_size (una_buf) == 0.0 && op_sar_buf_size (snd_buf) == 0.0)
		{
		if (op_ev_valid (ack_evh) && op_ev_pending (ack_evh))
			{
			/* If there was a pending delayed ACK, send it now	*/
			/* with the FIN segment.							*/
			if (op_ev_cancel (ack_evh) == OPC_COMPCODE_FAILURE)
				{
				tcp_conn_warn ("Unable to cancel dataless acknowledgement.",
					"A spurious dataless acknowledgement may be sent.", OPC_NIL);
				}
			}

		/* Send a TCP segment indicating a FIN request.			*/
		tcp_fin_send ();
		}

	FOUT;
	}

static void
tcp_fin_send (void)
	{
	double				next_timeout_time;

	/** Sens the FIN segment. **/
	FIN (tcp_fin_send ());


	/* Send the FIN segment, and set the state variable	*/
	/* to indicate that it has been sent.				*/
	tcp_seg_send (TCPC_DATA_NONE, snd_nxt++, TCPC_FLAG_ACK | TCPC_FLAG_FIN);
	fin_segment_sent = OPC_TRUE;

	/* Update the maximum send sequence number. In an	*/
	/* ideal case, when the system has not observed any */
	/* retransmissions, SND.NXT and SND.MAX are equal.	*/
	/* Refer page 808 (TCP/IP Illustrated Volume 2)	*/
	if (tcp_seq_ge (snd_nxt, snd_max))
		snd_max++;

	if ((op_ev_valid (retrans_evh) == OPC_FALSE) || (op_ev_pending (retrans_evh) == OPC_FALSE))
		{
		/* Compute the next RTO expiration time.	*/
		next_timeout_time = Tcp_Slowtimo_Next_Timeout_Time_Obtain (op_sim_time (), current_rto, timer_gran);

		/* Schedule the retransmission timeout.				*/
		retrans_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_retrans, OPC_NIL);
		}
```

```
	if (op_ev_valid (retrans_evh) == OPC_FALSE)
		{
		tcp_conn_warn ("Unable to schedule retransmission timeout.",
			"FIN segment will not be retransmitted.", OPC_NIL);
		}

	FOUT;
	}

static void
tcp_syn_send (TcpT_Flag flags)
	{
	double				next_timeout_time;

	/** Send a SYN with the given flags. If the supplied	**/
	/** flags is NONE, then it is the first SYN packet;		**/
	/** otherwise, it is the SYN-ACK packet.			**/
	FIN (tcp_syn_send (flags));

	/* Determine the initial send sequence number.		*/
	/* Set the send sequence variables.			*/
	if (tcp_parameter_ptr->init_seq_num == TCPC_ISS_AUTO_COMPUTE)
		{
		snd_una = iss = (TcpT_Seq) (fmod (op_sim_time () * 250000.0,
			1.0 + (double )(unsigned )0xffffffff)) & 0xffffffff;
		}
	else
		{
		snd_una = iss = (TcpT_Seq) tcp_parameter_ptr->init_seq_num;
		}

	snd_nxt = snd_una + 1;

	/* Update the maximum send sequence number.			*/
	snd_max = snd_una + 1;

	/* Initialize the last send time.				*/
	last_snd_time = op_sim_time ();

	/* Check whether it is a SYN or a SYN-ACK segment sent operation?	*/
	if (flags & TCPC_FLAG_ACK)
		{
		/* We need to send a SYN-ACK, in response to a SYN message		*/
		/* received from the peer TCP connection.  Check if this		*/
		/* connection can support ECN.  This is set based on this		*/
		/* end's support for ECN and the contents of SYN received		*/
		/* from the other side.								*/
		if (tcb_ptr->ecn_status & TcpC_Ecn_Supported)
			{
			/* Set the ECN flags for SYN-ACK segment. Per RFC-3168,	*/
			/* only the ECE flag is set.					*/
			flags |= TCPC_FLAG_ECE;
			}
		}
	else
		{
		/* We need to send a SYN packet -- this is the first packet		*/
		/* sent to initiate the 3-way connection setup handshake.		*/

		/* We need to set all attributes configured as			*/
		/* passive to disabled.						*/
		tcp_conn_attr_set_passive_as_disabled ();

		/* Check if ECN capability is enabled for this process.		*/
```

```
		if (tcp_parameter_ptr->ecn_capability == TCPC_OPTION_STATUS_ENABLED)
			{
			/* Set the ECN flags for SYN segment. Per RFC-3168,		*/
			/* these are ECE as wella s CWR.					*/
			flags |= TCPC_FLAG_ECE;
			flags |= TCPC_FLAG_CWR;

			/* Since we have initiated a SYN message, indicate	*/
			/* this in the TCB data maintained for this process	*/
			/* If the remote side does not support ECN, status	*/
			/* will be reverted to not supported.			*/
			tcb_ptr->ecn_status |= TcpC_Ecn_Supported;
			}
		}

	/* Now, when we know remote address of the connection for both	*/
	/* active and passive opens, set the maximum segment size for*/
	/* a case when it was configured to "Auto-Assign.		*/
	/* This could not have been done earlier, because MSS depends	*/
	/* on IP version - to get MSS, different number of bytes are	*/
	/* deducted from MTU for IPv4 and IPv6 headers.				*/

	/* In case the user has opted for the MSS to be Auto-Assigned, then */
	/* the manager would read the first ip interface MTU size and store */
	/* it as the maximum segment size.			*/
	if (snd_mss == TCPC_MSS_AUTO_ASSIGNED)
		{
		/* Scans the first configured IP interface on this node and	*/
		/* returns its MTU size.				*/
		tcp_parameter_ptr->max_seg_size  = tcp_conn_mss_auto_assign (tcb_ptr->rem_addr);
		}

	/* Send the SYN segment.					*/
	tcp_seg_send (TCPC_DATA_NONE, iss, TCPC_FLAG_SYN | flags);

	/* Compute the next RTO expiration time.	*/
	next_timeout_time = Tcp_Slowtimo_Next_Timeout_Time_Obtain (op_sim_time (), current_rto, timer_gran);

	/* Schedule the retransmission timeout.			*/
	retrans_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_retrans, OPC_NIL);
	if (op_ev_valid (retrans_evh) == OPC_FALSE)
		{
		tcp_conn_warn ("Unable to schedule retransmission timeout.",
			"SYN segment will not be retransmitted.", OPC_NIL);
		}

	FOUT;
	}


/**	Functions related to the Window Scaling Option.	**/

static void
tcp_send_window_update (TcpT_Seq seq_num, TcpT_Seq ack_num, TcpT_Seq new_snd_wnd)
	{
	char			str0[64];

	/** Updates all variables related to the send window (i.e., snd_wl1, snd_wl2,	**/
	/** and snd_wnd) when a new segment is received.  Performs window scaling, if		**/
	/** necessary, by shifting the received window by the receive window scale			**/
	/** factor received in the SYN.
		**/
	FIN (tcp_send_window_update (seq_num, ack_num, new_snd_wnd));

	snd_wl1 = seq_num;
```

```
			snd_wl2 = ack_num;

			/* The send window should be just shifted by zero if window scaling is not enabled.		*/
			snd_wnd = new_snd_wnd << snd_scale;

			if (tcp_trace_active || tcp_extns_trace_active)
				{
				sprintf (str0, "The new send window is %u.", snd_wnd);
				op_prg_odb_print_minor (str0, OPC_NIL);
				}

			FOUT;
			}

static void
tcp_window_scale_option_process (Packet* pk_ptr)
	{
	char	str0[64];
			int			temp_ws_flag;

	/** Obtains requested window scale factor from a received SYN, if option was set.		**/
	/** Only performed if this host has window scaling enabled as well.					**/
	FIN (tcp_window_scale_option_process(pk_ptr));

			/* Initialize window scaling to be disabled.			*/
			temp_ws_flag = TCPC_OPTION_STATUS_DISABLED;

	if (op_pk_nfd_is_set (pk_ptr, "Window Scaling Option"))
				{
				if (op_pk_nfd_get (pk_ptr, "Window Scaling Option", &requested_snd_scale) == OPC_COMPCODE_FAILURE)
					{
					tcp_conn_warn ("Unable to get Window Scaling Option from SYN segment.",
						"Ignoring option.", OPC_NIL);
					}
				else
					{
					/* window scaling is available on both sides of the connection */
					/* so enable it.
			*/
					temp_ws_flag = TCPC_OPTION_STATUS_ENABLED;

					wnd_scale_rcvd = OPC_TRUE;
					requested_snd_scale = MIN (TCPC_WS_MAX_WND_SHIFT, requested_snd_scale); /* no scaling factor >14 */
					}

				if (tcp_trace_active || tcp_extns_trace_active)
					{
					sprintf (str0, "Received SYN requesting window scale of %d.", requested_snd_scale);
					op_prg_odb_print_minor (str0, OPC_NIL);
					}
				}

			/* If there is a transition from window_scaling_enabled being	*/
			/* passive to another state, then a trace message detailing			*/
			/* this change needs to be created.							*/
			if ((window_scaling_enabled == TCPC_OPTION_STATUS_PASSIVE) && (tcp_trace_active || tcp_extns_trace_active))
				{
				if (temp_ws_flag == TCPC_OPTION_STATUS_ENABLED)
					{
					op_prg_odb_print_minor ("Enabled window scaling for the connection.",
								"after SYN message with a supported window scale",
								"option has been received.", OPC_NIL);
					}
				else
					{
```

```
				op_prg_odb_print_minor ("Disabled window scaling for the connection.",
							"after SYN message with an unsupported window scale",
							"option has been received.", OPC_NIL);
				}
			}

		/* Configuration of window scaling needs to be done			*/
		/* here rather than in the initialization state since the	*/
		/* case where it may be passive needs to be handled.		*/
		if ((temp_ws_flag == TCPC_OPTION_STATUS_DISABLED) && (rcv_buff > TCPC_MAX_WND_SIZE))
			{
			/* Window scaling is disabled but the receive buffer specified	*/
			/* greater than the value which the 16-bit field can carry.		*/
			/* Truncate the receive buffer specification, and write a		*/
			/* simulation log message.											*/
			tcp_rcvbuff_truncate_log_write (rcv_buff, TCPC_MAX_WND_SIZE);
			rcv_buff = TCPC_MAX_WND_SIZE;
			}

		else if ((temp_ws_flag == TCPC_OPTION_STATUS_ENABLED) && (rcv_buff > TCPC_MAX_WND_WWS_SIZE))
			{
			tcp_rcvbuff_truncate_log_write (rcv_buff, TCPC_MAX_WND_WWS_SIZE);
			rcv_buff = TCPC_MAX_WND_WWS_SIZE;
			}

		/* Initial receive window size is the full buffer size. */
		rcv_wnd = rcv_buff;

		/* The ssthresh is either set to this value based on TCP/IP Illustrated vol2, pg 835 */
		if (temp_ws_flag == TCPC_OPTION_STATUS_ENABLED)
			ssthresh = TCPC_MAX_WND_WWS_SIZE;
		else
			ssthresh = TCPC_MAX_WND_SIZE;

		window_scaling_enabled = temp_ws_flag;

	FOUT;
	}

static void
tcp_window_scaling_initialize ()
	{
	char			str0[256];

	/** Once a connection has been fully established, compete the setup process for window scaling **/
	/** If scale factor was both sent and received, enable window scaling on all sent/rcvd packets **/
	FIN (tcp_window_scaling_initialize ());

	if (wnd_scale_sent && wnd_scale_rcvd)
			{
			snd_scale = requested_snd_scale;
			rcv_scale = requested_rcv_scale;
			}

	if (tcp_trace_active || tcp_extns_trace_active)
			{
			sprintf (str0, "Send scale factor will be %d and receive scale factor will be %d.",
				snd_scale, rcv_scale);
			op_prg_odb_print_minor (str0, OPC_NIL);
			}

	FOUT;
	}
```

```
/**	Functions related to the SACK Option.			**/

static TcpT_Sackoption*
tcp_sackoption_get (Packet* pk_ptr)
	{
	TcpT_Sackoption*	option_ptr;
	char				str0[256];

	/** Forms a list of sack blocks received in an incoming packet which has the SACK	**/
	/** option set. The list is obtained from the SACK Option field of the received		**/
	/** packet, and is passed on to tcp_scoreboard_update_sack.				**/
	FIN (tcp_sackoption_get (pk_ptr));

		/* This function is called after making sure that the SACK option field	*/
		/* is set in th eincoming packet. Obtain the set value.				*/
		op_pk_nfd_get (pk_ptr, "SACK Option", &option_ptr);

		if (tcp_trace_active || tcp_extns_trace_active)
			{
			sprintf (str0, "Received a SACK Option containing %d blocks.", option_ptr->size);
			op_prg_odb_print_major (str0, OPC_NIL);
			}

	FRET (option_ptr);
	}

static void
tcp_sackoption_set (Packet* pk_ptr)
	{
	TcpT_Sackoption*	option_ptr;
	TcpT_Sackblock*		block_ptr;
	int					i, sack_count;
	int					included_sack_count;	/* the number of sack blocks which will go into the option */

	/** Inserts the list of sack blocks into the SACK Option field of an outgoing pkt.	**/
	/** Adjusts packet size accordingly, as each block takes up 8 bytes in the header	**/
	/** and this has not yet been accounted for. The number of blocks which can be sent	**/
	/** in the option is limited by the size of the header.					**/
	FIN (tcp_sackoption_set (pk_ptr));

		if (op_prg_list_size (sacklist_ptr->entries) == 0)
			{
			/* SACK list is empty. There is no need to set SACK option fields in the outgoing message.	*/

			FOUT;
			}

		if (pk_ptr == OPC_NIL || sacklist_ptr == OPC_NIL)
			{
			if (tcp_trace_active)
				tcp_conn_warn ("Unable to set SACK option in outgoing packet.", OPC_NIL, OPC_NIL);

			FOUT;
			}

	sack_count = op_prg_list_size (sacklist_ptr->entries);
	included_sack_count = MIN (sack_count, TCPC_SACK_MAX_BLOCKS);

	option_ptr = (TcpT_Sackoption *) op_prg_mem_alloc (sizeof (TcpT_Sackoption));
	option_ptr->size = included_sack_count;

	/* each block takes up two spots in the array (star, end) */
	option_ptr->blocks = (TcpT_Seq *) op_prg_mem_alloc (2 * included_sack_count * sizeof (TcpT_Seq));

	/* copy the blocks to be included in the packet from the full list */
```

```
	for (i = 0; i < included_sack_count; i++)
			{
			block_ptr = (TcpT_Sackblock *) op_prg_list_access (sacklist_ptr->entries, i);
			option_ptr->blocks[2*i] = block_ptr->start;
			option_ptr->blocks[2*i + 1] = block_ptr->end;
			}

	if (op_pk_nfd_set (pk_ptr, "SACK Option", option_ptr, op_prg_mem_copy_create, op_prg_mem_free,
			sizeof (TcpT_Sackoption)) == OPC_COMPCODE_FAILURE)
			{
			tcp_conn_warn ("Unable to set SACK option in outgoing packet.", OPC_NIL, OPC_NIL);
			FOUT;
			}

	/* Adjust the total size of the packet upwards to account for additional header space used		*/
	/* by the SACK option.  Each block in the option takes up 8 bytes, in addition to 2 bytes		*/
	/* already used for kind/length fields of the option.						*/
	op_pk_total_size_set (pk_ptr,
			op_pk_total_size_get (pk_ptr) + (8 * TCPC_SACK_BLOCK_SIZE * included_sack_count));

	FOUT;
	}

static void
tcp_sack_initialize ()
	{
	/** Once a connection process has been fully established, complete the setup process for SACK **/
	/** transmission and reception.  If SACK is to be enabled, initialize SACK-related variables. **/
	FIN (tcp_sack_initialize ());

	if (SACK_PERMITTED)
			{
			scoreboard_ptr = (TcpT_Scoreboard*) op_prg_mem_alloc (sizeof (TcpT_Scoreboard));
			sacklist_ptr = (TcpT_Sacklist*) op_prg_mem_alloc (sizeof (TcpT_Sacklist));

			tcp_scoreboard_and_sacklist_ptr_valid = OPC_TRUE;

			scoreboard_ptr->entries = op_prg_list_create ();
			sacklist_ptr->entries = op_prg_list_create ();

			scoreboard_ptr->recovery_end = snd_una;
			scoreboard_ptr->last_retran_end = snd_una;
			scoreboard_ptr->more_retran = OPC_FALSE;

			scoreboard_entry_pmh = op_prg_pmo_define ("Scoreboard Entry", sizeof (TcpT_Scoreboard_Entry), 32);
			sackblock_pmh = op_prg_pmo_define ("Sackblock", sizeof (TcpT_Sackblock), 32);
			}

	FOUT;
	}

static void
tcp_scoreboard_clear ()
	{
	/** Removes all elements from the scoreboard. Record of blocks SACK'd by receiver will be lost. **/
	/** Performed after a retransmission timeout, or after we infer that the receiver has reneged	**/
	/** on a block which was previously SACKed (i.e. receiver previously SACK'd the block but does	**/
	/** not do so anymore. Presumably, this is because the receiver ran out of buffer space for	**/
	/** holding out-of-order packets.								**/
	FIN (tcp_scoreboard_clear ());

	if (scoreboard_ptr->entries != OPC_NIL)
			{
			op_prg_list_free (scoreboard_ptr->entries);
			}
```

```
scoreboard_ptr->recovery_end = snd_una;
scoreboard_ptr->last_retran_end = snd_una;
scoreboard_ptr->more_retran = OPC_FALSE;

if (tcp_trace_active || tcp_extns_trace_active)
		{
		op_prg_odb_print_major ("Cleared the scoreboard.", OPC_NIL);
		tcp_scoreboard_print ();
		}

FOUT;
}

static void
tcp_scoreboard_update_newack (TcpT_Seq ack_seq, TcpT_Seq old_snd_una)
	{
	TcpT_Scoreboard_Entry*		cur_entry_ptr;
		Boolean			scoreboard_updated = OPC_FALSE;
	char					str0[256];
	int						i;

	/** Updates the scoreboard when an advancing acknowledgement is received.  Delete any		**/
	/** blocks which are completely covered by the new cumulative ACK.  If the new ACK		covers	**/
	/** all data which was outstanding at the time of retransmission, come out of the recovery		**/
	/** phase and resume normal data sending/congestion control.  In case the ACK only partly		**/
	/** covers a SACK'd block, the data receiver has reneged on the SACK and the entire	**/
	/** scoreboard will be cleared.										**/
	FIN (tcp_scoreboard_update_newack (ack_seq, old_snd_una));

	/* if data has been ACK'd up to the value of snd_nxt at time of retranmsission, then the		*/
	/* recovery phase is over */

		/* Need to make sure that this only occurs if retransmission already		*/
		/* happpened, as recovery_end is otherwise uninitialized.				*/
	if (dup_ack_cnt > tcp_parameter_ptr->fr_dup_ack_thresh &&
			tcp_seq_ge (ack_seq, scoreboard_ptr->recovery_end))
			{
			tcp_scoreboard_clear ();
			FOUT;
			}

	/* Otherwise, this must be a partial ACK.  Advances snd_una, but does		*/
		/* not ACK all data outstanding when retransmission occured.			*/

	/* Special handling of pipe for partial ACKs.  Pipe is decremented by*/
		/* two packets. For explanation, see Fall & Floyd's "Simulation-based	*/
		/* Comparison of Taho, Reno and SACK TCP." Need to make sure that this	*/
		/* only occurs if retransmission already happpened, as pipe is		*/
		/* otherwise uninitialized.								*/
		/* Update the pipe variable. Decrement it by:					*/
		/* 1. the number of ACKed data, and							*/
		/* 2. the number of retransmitted data up to ACK sequence.		*/
		/* This is because we incremented the pipe when we sent			*/
		/* the original packet as well as when we retransmitted the packet.		*/
		/* For explanation, see Fall & Floyd's "Simulation-based Comparison	*/
		/* of Taho, Reno and SACK TCP."							*/

		/* To update pipe variable, first, decrement the pipe by cumulatively	*/
		/* ACKed data, and then increment it by already selective ACKed data. */
		if (tcp_seq_ge (scoreboard_ptr->last_retran_end, ack_seq))
			{
			pipe -= 2 * (ack_seq - old_snd_una);
			}
		else
```

```
                {
                pipe -= (ack_seq - old_snd_una) + (scoreboard_ptr->last_retran_end - old_snd_una);
                }

/* Loop through the entries in the scoreboard, deleting those which    */
        /* have been cumulatively ACK'd.                                                        */
for (i = 0; i < op_prg_list_size (scoreboard_ptr->entries); i++)
                {
                cur_entry_ptr = (TcpT_Scoreboard_Entry*) op_prg_list_access (scoreboard_ptr->entries, i);

                /* Determine if new ack reaches at least the start of the current block */
                if (tcp_seq_ge (ack_seq, cur_entry_ptr->start))
                        {
                        /* if ack falls mid-block, this indicates receiver reneging - clear the scoreboard */
                        if (tcp_seq_lt (ack_seq, cur_entry_ptr->end))
                                {
                                tcp_scoreboard_clear ();
                                break;
                                }

                        /* the ack reaches at least the end of this block - delete it */
                        else
                                {
                                /* Decrement the pipe by SACKed data.                                   */
                                pipe += cur_entry_ptr->end - cur_entry_ptr->start;

                                /* This data has also been resent, increment the           */
                                /* pipe by the size of this block.                                   */
                                if (tcp_seq_ge (scoreboard_ptr->last_retran_end, cur_entry_ptr->end))
                                        {
                                        pipe += cur_entry_ptr->end - cur_entry_ptr->start;
                                        }

                                op_prg_list_remove (scoreboard_ptr->entries, i);
                                op_prg_mem_free (cur_entry_ptr);

                                /* Set the scroboard entries deleted flag to TRUE  if   */
                                /* the current ack_seq acknowledges all the last retrans*/
                                /* data in the scoreboard.                                                   */
                                if (tcp_seq_ge (ack_seq, scoreboard_ptr->last_retran_end))
                                        {
                                        /* Data has been successfully ACKed upto "ack_seq";             */
                                        /* set scoreboard variable s to reflect this.                  */
                                        scoreboard_ptr->last_retran_end = ack_seq;

                                        scoreboard_updated = OPC_TRUE;
                                        }

                                /* indexes will be decremented for all following entries after removal */
                                /* adjust counter accordingly */
                                i--;
                                }
                        }

                /* found the first block which is not overlapped/subset */
                else
                        {
                        /* check if any data from the hole before this block has been ACK'd       */
                        if (tcp_seq_gt (ack_seq, cur_entry_ptr->retran_end))
                                {
                                cur_entry_ptr->retran_end = ack_seq;
                                }

                        break;
                        }
```

```
			}

	/* It might have happened that the scoreboard might have increased  */
	/* after the first retransmission and on deleting the current       */
	/* scoreboard enrty we should enable the more_retrans flag.         */
	if (op_prg_list_size (scoreboard_ptr->entries) && scoreboard_updated == OPC_TRUE)
	    scoreboard_ptr->more_retran = OPC_TRUE;

	if (tcp_trace_active || tcp_extns_trace_active)
			{
			sprintf (str0, "Updated the scoreboard due to receipt of ack %u.", ack_seq);
			op_prg_odb_print_major (str0, OPC_NIL);
			tcp_scoreboard_print ();
			}

	FOUT;
	}

static void
tcp_scoreboard_update_sack (TcpT_Sackoption* new_sacklist_ptr)
	{
	TcpT_Scoreboard_Entry*			new_entry_ptr;
	TcpT_Scoreboard_Entry*			cur_entry_ptr;
	TcpT_Scoreboard_Entry*			prev_entry_ptr;
	TcpT_Scoreboard_Entry*			tmp_entry_ptr;
	TcpT_Seq						sack_start;
	TcpT_Seq						sack_end;
	int								entry_count, new_sack_count, insert_index;
	    int									i, j, k;

	Boolean overlap = OPC_FALSE;		/* true if a SACK block overlaps or abuts a SACK block already received */
	Boolean contained = OPC_FALSE;		/* true if a SACK block is fully contained within one already received */
	Boolean between = OPC_FALSE;		/* true if a SACK block belongs between two blocks already received */
	Boolean ends = OPC_FALSE;			/* true if a SACK block belongs at one of the ends of the list */

	/** Updates the scoreboard based on a sack list newly-received in the SACK option.     The scoreboard    **/
	/** consists of a list of blocks which have been SACK'd by the data receiver.  Each entry records            **/
	/** the start and end of the block, plus a variable called retran_end.  This variable contains the    **/
	/** seqeuence number of the highest byte of data retransmitted PRIOR to the start of the block.  In           **/
	/** addition to the blocks, the scoreboard also contains vars recovery_end and last_retran_end.  **/
	/** Recovery_end is the value of snd_nxt at the time the first retransmission occured, and it        **/
	/** indicates the sequence number of the ACK which will allow us to come out of fast recovery.             **/
	/** Last_retran is the sequence number of the last byte retransmitted. It is used for determining  **/
	/** how much SACK'd data must be cleared out of the dup_una_buf before each retransmission (no need **/
	/** to retransmit this data).  It is also used for determining when there is no more data to            **/
	/** retransmit.  This function goes through the received SACK list and decides whether each block          **/
	/** SACKs only new data, SACKs only data already SACKed or SACKs a mixture.  In the first case, the **/
	/** block will simply be added to the scoreboard, and in the second, the block will simply be      **/
	/** discarded.  In case the block sacks a mixture of old and new data, the scoreboard will be       **/
	/** updated to reflect that the union of the old and new data has been SACKd.                        **/
	FIN (tcp_scoreboard_update (new_sacklist_ptr));

	/* Loop through all the SACK blocks, inserting new ones into the list and updating old ones */
	/* if necessary */

	if (new_sacklist_ptr == OPC_NIL)
			{
			tcp_conn_warn ("Unable to update scoreboard based on newly-received ACK.", OPC_NIL, OPC_NIL);
			FOUT;
			}

	new_sack_count = new_sacklist_ptr->size;
	for (k = 0; k < new_sack_count; k++)
			{
			sack_start		= new_sacklist_ptr->blocks[k*2];
```

```
		sack_end		= new_sacklist_ptr->blocks[k*2 + 1];

		entry_count = op_prg_list_size (scoreboard_ptr->entries);

		overlap	= OPC_FALSE;
		contained	= OPC_FALSE;
		between = OPC_FALSE;
		ends		= OPC_FALSE;

		/* check for invalid sacks */

		/* throw block out if does not represent a valid data range */
		if (tcp_seq_ge (sack_start, sack_end))
			{
			continue;
			}

		/* if block SACKS data already cumulatively ACK'd, throw it out		*/
		/* also throw it out if it SACKs the next block for which we expect an	*/
		/* ACK										*/
		if (tcp_seq_le (sack_end, snd_una) || tcp_seq_le (sack_start, snd_una))
			{
			continue;
			}

		/* valid sack */

		/* if scoreboard is empty, just insert a new block at the front of the list */
		if (entry_count == 0)
			{
			new_entry_ptr = (TcpT_Scoreboard_Entry*) op_prg_pmo_alloc (scoreboard_entry_pmh);

			new_entry_ptr->start			= sack_start;
			new_entry_ptr->end				= sack_end;
			new_entry_ptr->retran_end		= snd_una;

			op_prg_list_insert (scoreboard_ptr->entries, new_entry_ptr, OPC_LISTPOS_HEAD);

			scoreboard_ptr->more_retran = OPC_TRUE;

			/* Decrement the pipe by the number of ACKed data. */



			pipe -= (new_entry_ptr->end - new_entry_ptr->start);

			continue;
			}

/*** find insertion point for new block ****/

		/* 1. if new block is wholly contained within a previous sack block, set contained flag */
		/* 2. if new block overlaps a previous sack block, find pointer to first overlapped block */
		/* 3. otherwise, find pointer to first block whose start sequence is greater than new one */

		cur_entry_ptr = (TcpT_Scoreboard_Entry*)op_prg_list_access (scoreboard_ptr->entries, OPC_LISTPOS_HEAD);
		for (i = 0; i < op_prg_list_size (scoreboard_ptr->entries); i++)
			{
			prev_entry_ptr = cur_entry_ptr;
			cur_entry_ptr = (TcpT_Scoreboard_Entry*)op_prg_list_access (scoreboard_ptr->entries, i);

			/* check if this block is wholly contained within current block */
			if (tcp_seq_ge (sack_start, cur_entry_ptr->start) && tcp_seq_le (sack_end, cur_entry_ptr->end))
				{
				contained = OPC_TRUE;
```

```
			break;
			}

		/* check if this block belelongs at one of the ends of the list */
		else if (i == 0 && tcp_seq_lt (sack_end, cur_entry_ptr->start))
			{
			ends = OPC_TRUE;
			insert_index = OPC_LISTPOS_HEAD;
			break;
			}
		else if (i == (entry_count-1) && tcp_seq_gt (sack_start, cur_entry_ptr->end))
			{
			ends = OPC_TRUE;
			insert_index = OPC_LISTPOS_TAIL;
			break;
			}

		/* check if this block belongs completely between previous and current block */
		else if ((i != 0 && i != entry_count-1) &&
			(tcp_seq_gt (sack_start, prev_entry_ptr->end) &&
			tcp_seq_lt (sack_end, cur_entry_ptr->start)))
			{
			between = OPC_TRUE;
			insert_index = i;
			break;
			}

		/* if block does not overlap or touch the current block at all, keep searching */
		else if (tcp_seq_gt (sack_start, cur_entry_ptr->end))
			{
			continue;
			}

		/* otherwise, the block must overlap the current block because: */
		/* 1.  it starts at or before this block ends			*/
		/* 2.  it does not end before this one begins			*/
		else
			{
			overlap = OPC_TRUE;
			insert_index = i;
			break;
			}
		}

	/*** incorporate new block into the list ***/

	/* simply discard a SACK contained within one already received */
	if (contained)
		{
		continue;
		}

	/* add a block at the beginning or end */
	else if (ends)
		{
		new_entry_ptr = (TcpT_Scoreboard_Entry*) op_prg_pmo_alloc (scoreboard_entry_pmh);

		new_entry_ptr->start = sack_start;
		new_entry_ptr->end = sack_end;

		if (insert_index == OPC_LISTPOS_HEAD)
			{
			new_entry_ptr->retran_end = snd_una;
			op_prg_list_insert (scoreboard_ptr->entries, new_entry_ptr, OPC_LISTPOS_HEAD);
			}
```

```
            else if (insert_index == OPC_LISTPOS_TAIL)
                        {
                        new_entry_ptr->retran_end = cur_entry_ptr->end;
                        op_prg_list_insert (scoreboard_ptr->entries, new_entry_ptr, OPC_LISTPOS_TAIL);
                        }

            /* New entry has been added. This will create a need for a new retransmission.    */
            scoreboard_ptr->more_retran = OPC_TRUE;

            /* If new data has been received by the other host, and retransmission has             */
            /* already occured, decrement pipe by the amount of newly-received data.           */
            if (tcp_seq_lt (snd_una, scoreboard_ptr->recovery_end))
                        {
                        pipe -= (sack_end - sack_start);
                        }

continue;
            }

/* add a block between two others */
else if (between)
            {
            new_entry_ptr = (TcpT_Scoreboard_Entry*) op_prg_pmo_alloc (scoreboard_entry_pmh);

            new_entry_ptr->start = sack_start;
            new_entry_ptr->end = sack_end;

            new_entry_ptr->retran_end = cur_entry_ptr->end;

            /* adjust retransmission pointers so they do not overlap any sack blocks */
            if (tcp_seq_gt (new_entry_ptr->retran_end, new_entry_ptr->start))
                        {
                        new_entry_ptr->retran_end = new_entry_ptr->start;
                        }

            if (tcp_seq_lt (cur_entry_ptr->retran_end, new_entry_ptr->end))
                        {
                        cur_entry_ptr->retran_end = new_entry_ptr->end;
                        }

            op_prg_list_insert (scoreboard_ptr->entries, new_entry_ptr, insert_index);

            /* If new data has been received by the other host, and retransmission has             */
            /* already occured, decrement pipe by the amount of newly-received data.           */
            if (tcp_seq_lt (snd_una, scoreboard_ptr->recovery_end))
                        {
                        pipe -= (sack_end - sack_start);
                        }

            continue;
            }

/* finally, deal with case in which new block overlaps current block */

/* really, any new block which overlaps an old one but does not include all data in the old */
/* one might indicate that receiver has reneged. however, it could also just happen b/c of  */
/* out-of-order packets.  we assume the later here and just merge the two blocks.     */
else if (overlap)
  {
  /* if new block starts before current block, update start of current block appropriately */
  if (tcp_seq_lt (sack_start, cur_entry_ptr->start))
            {
            /* If new data has been received by the other host, and retransmission has         */
            /* already occured, decrement pipe by the amount of newly-received data.        */
```

```
        if (tcp_seq_lt (snd_una, scoreboard_ptr->recovery_end))
            {
            pipe -= (cur_entry_ptr->start - sack_start);
            }

        cur_entry_ptr->start = sack_start;

        /* adjust retransmission pointer so it doesn't overlap current block */
        if (tcp_seq_gt (cur_entry_ptr->retran_end, cur_entry_ptr->start))
            {
            cur_entry_ptr->retran_end = cur_entry_ptr->start;
            }
        }

    /* if new block ends after current block, update end of current block appropriately */
    if (tcp_seq_gt (sack_end, cur_entry_ptr->end))
        {
        /* If new data has been received by the other host, and retransmission has     */
        /* already occured, decrement pipe by the amount of newly-received data.       */
        if (tcp_seq_lt (snd_una, scoreboard_ptr->recovery_end))
            {
                                /* "pipe" variable stores the amount of unACKed data.     */
            /* The SACK list received from the other end may contain    */
            /* porions that are already ACKed (thereby, they are        */
            /* known to the sender in its scoreboard). Update "pipe"    */
            /* to decrement only the newly acked data.                  */

            /* First, decremnent the maximum amount that can be decremented.   */
            pipe -= (sack_end - sack_start);

            /* Then, adjust the the amount of known data block.         */
            pipe += (cur_entry_ptr->end - cur_entry_ptr->start);
            }

        cur_entry_ptr->end = sack_end;

        /* remove any blocks which are subsets of or overlapped by this new block */
        for (j = insert_index + 1; j < op_prg_list_size (scoreboard_ptr->entries); j++)
            {
            tmp_entry_ptr = (TcpT_Scoreboard_Entry*) op_prg_list_access (scoreboard_ptr->entries, j);

            /* found an overlapped or touched block */
            if (tcp_seq_ge (cur_entry_ptr->end, tmp_entry_ptr->start))
                {
                /* Account for the fact that have subtracted this data amount from     */
                /* pipe twice (once when orignially rcvd, and once above.)             */
                if (tcp_seq_lt (snd_una, scoreboard_ptr->recovery_end))
                    {
                    pipe += (tmp_entry_ptr->end - tmp_entry_ptr->start);
                    }

                if (tcp_seq_gt (tmp_entry_ptr->end, cur_entry_ptr->end))
                    cur_entry_ptr->end = tmp_entry_ptr->end;

                /* this block has been made obsolete by the newly updated block */
                op_prg_list_remove (scoreboard_ptr->entries, j);
                op_prg_mem_free (tmp_entry_ptr);

                /* indexing will decremented for all following entries after removal */
                /* adjust counter accordingly */
                j--;
                }

            /* found the first block which is not overlapped/subset */
            else
```

```
                                    {
                                    /* make sure that retran_end for this block hasn't somehow been  */
                                    /* passed by the end value for the previous block      */
                                    if (tcp_seq_gt (cur_entry_ptr->end, tmp_entry_ptr->retran_end))
                                        {
                                        tmp_entry_ptr->retran_end = cur_entry_ptr->end;
                                        }

                                    break;
                                    }
                                }
                            }
                        }
                    }

    /* deallocate memory used by new_sacklist */
    op_prg_mem_free (new_sacklist_ptr->blocks);
    op_prg_mem_free (new_sacklist_ptr);

    if (tcp_trace_active || tcp_extns_trace_active)
                    {
                    op_prg_odb_print_major ("Updated the scoreboard due to receipt of new SACK option.", OPC_NIL);
                    tcp_scoreboard_print ();
                    }

    FOUT;
    }

static TcpT_Sackblock*
tcp_scoreboard_find_retransmission (int max_bytes)
    {
    TcpT_Scoreboard_Entry*          cur_entry_ptr;
    TcpT_Sackblock*                             retran_block_ptr = OPC_NIL;
    int                                                     i;

    /** Returns start and end sequence numbers for the earlist segment which has not already been **/
    /** retransmitted.  Updates the scoreboard to indicate that segment is being retransmitted.   **/
    /** If all segments have been retransmitted, returns a block with start = end. Will not       **/
    /** return a segment longer than max_bytes.                                                   **/
    FIN (tcp_scoreboard_find_retransmission (max_bytes));

    retran_block_ptr = (TcpT_Sackblock*) op_prg_pmo_alloc (sackblock_pmh);
    retran_block_ptr->start = retran_block_ptr->end = 0;

    for (i = 0; i < op_prg_list_size (scoreboard_ptr->entries); i++)
                    {
                    cur_entry_ptr = (TcpT_Scoreboard_Entry *) op_prg_list_access (scoreboard_ptr->entries, i);

                    /* if all data up to start of block has been retransmitted, keep searching */
                    if (cur_entry_ptr->start == cur_entry_ptr->retran_end)
                        {
                        continue;
                        }

                    /* data between this block and the previous has not yet been retransmitted           */
                    else
                        {
                        /* try sending all missing data between this one and the previous */
                        retran_block_ptr->start = cur_entry_ptr->retran_end;
                        retran_block_ptr->end = cur_entry_ptr->start;

                        /* make sure length of retransmission block is not larger than what can be sent */
                        if (retran_block_ptr->end - retran_block_ptr->start > max_bytes)
                            {
                            retran_block_ptr->end = retran_block_ptr->start + max_bytes;
```

```
                                        }

                                /* in the case in which we are sending all data up to the start of the last block */
                                /* may have finished all retransmission */
                                else if (i == (op_prg_list_size (scoreboard_ptr->entries) - 1) &&
                                        tcp_seq_ge (cur_entry_ptr->end, scoreboard_ptr->recovery_end))
                                        {
                                        scoreboard_ptr->more_retran = OPC_FALSE;
                                        }

                                /* mark the data as retransmitted */
                                cur_entry_ptr->retran_end = retran_block_ptr->end;

                                break;
                                }
                        }

    /* special handling for case when the end of the last sack block doesn't coincide with the          */
    /* value of scoreboard_ptr->recovery_end. note that cur_entry_ptr now points to last block         */
    if ((retran_block_ptr->start == retran_block_ptr->end) &&
                        (scoreboard_ptr->more_retran))
                        {
                        /* The scoreboard does not have any more "holes", which have not yet been         */
                        /* retransmitted. Until we receive a new SACK option informing us about           */
                        /* a new missing packet, no retransmissions are necessary. We will be             */
                        /* sending data from send buffer instead.                                                              */
                        scoreboard_ptr->more_retran = OPC_FALSE;
                        FRET (retran_block_ptr);
                        }


    scoreboard_ptr->last_retran_end = retran_block_ptr->end;

    /* Increment the pipe by the amount of data we are retransmitting.      */
    pipe +=(retran_block_ptr->end - retran_block_ptr->start);

    if (tcp_trace_active || tcp_extns_trace_active)
                        {
                        op_prg_odb_print_major ("Updated the scoreboard to account for new retransmission.", OPC_NIL);
                        tcp_scoreboard_print ();
                        }

    FRET (retran_block_ptr);
    }

static void
tcp_sacklist_update_newack (TcpT_Seq ack_seq)
    {
    int                                     i, num_blocks;
    TcpT_Sackblock*             cur_sackblock_ptr;
    char                        str0[256];

    /** Update the list upon receipt of a new in-order segment.  Deletes any SACK blocks which  **/
    /** have been made obsolete by the newly-advanced rcv.nxt.             Ack_seq is the ACK which this   **/
    /** host is currently sending, and blocks which end before Ack_seq are redundent since they **/
    /** are now included in the cumulative ACK.
                                **/
    FIN (tcp_sacklist_update_newack (ack_seq));

    num_blocks = op_prg_list_size (sacklist_ptr->entries);
    for (i = 0; i < num_blocks; i++)
                        {
                        cur_sackblock_ptr = (TcpT_Sackblock *) op_prg_list_access (sacklist_ptr->entries, i);

                        /* determine whether this sackblock contains only data which is being cumulatively ACK'd */
```

```
			if (tcp_seq_le (cur_sackblock_ptr->end, ack_seq))
				{
				op_prg_list_remove (sacklist_ptr->entries, i);
				op_prg_mem_free (cur_sackblock_ptr);

				/* adjust counters because indexes of following blocks has been decremented */
				i--;
				num_blocks--;
				}

			/* determine whether this sackblock contains some data which is being ACK'd	*/
			/* if so, this indicates that the this receiver has reneged on a SACK because	*/
			/* the previously SACK'd data extends past the end of the new ACK, but is not	*/
			/* included in the ACK.	this is probably occuring because we ran out of buffer*/
			/* space and were no loonger able to store a segment in the out-of-order list	*/
			/* throw out the block in this case.						*/
			else if (tcp_seq_gt (ack_seq, cur_sackblock_ptr->start))
				{
				if (tcp_trace_active || tcp_extns_trace_active)
					{
					op_prg_odb_print_major("Data receiver is reneging on data previously SACKed", OPC_NIL);
					}
				op_prg_list_remove (sacklist_ptr->entries, i);
				op_prg_mem_free (cur_sackblock_ptr);

				/* adjust counters because indexes of following blocks has been decremented */
				i--;
				num_blocks--;
				}
			}

	if (tcp_trace_active || tcp_extns_trace_active)
			{
			sprintf (str0, "Updating SACK list due to advancing rcv_nxt: %u.", ack_seq);
			op_prg_odb_print_major (str0, OPC_NIL);
			tcp_sacklist_print ();
			}

	FOUT;
	}

static void
tcp_sacklist_update_block (TcpT_Seq start, TcpT_Seq end)
	{
	TcpT_Sackblock*			new_block_ptr;
	TcpT_Sackblock*			cur_block_ptr;
	int					i, num_sackblocks;

	/** Updates the list based on a newly received out-of-order segment.  Adds a new block to	**/
	/** the start and incorporates old ones into it if the new one covers/overlaps them.	**/
	/** The first SACK block will always reflect the most recently received segment, and the	**/
	/** others will be stored in order of when they were first sent in the SACK option.		**/
	FIN (tcp_sacklist_update_block (start, end));

	/* Insert the new block at the start of the list */
	new_block_ptr = (TcpT_Sackblock*) op_prg_pmo_alloc (sackblock_pmh);
	new_block_ptr->start = start;
	new_block_ptr->end = end;

	op_prg_list_insert (sacklist_ptr->entries, new_block_ptr, OPC_LISTPOS_HEAD);

	/* Look through old blocks to see if this new block includes/touches/subsets them. */
	num_sackblocks = op_prg_list_size (sacklist_ptr->entries);
	for (i = 1; i < num_sackblocks; i++)
			{
```

```
			cur_block_ptr = (TcpT_Sackblock*) op_prg_list_access (sacklist_ptr->entries, i);

			/* this block does not overlap at all */
			if (tcp_seq_lt (new_block_ptr->end, cur_block_ptr->start) ||
				tcp_seq_gt (new_block_ptr->start, cur_block_ptr->end))
				{
				continue;
				}

			/* a block which starts before the new one does */
			if (tcp_seq_lt (cur_block_ptr->start, new_block_ptr->start))
				{
				new_block_ptr->start = cur_block_ptr->start;
				}

			/* a block which ends after the new one does */
			if (tcp_seq_gt (cur_block_ptr->end, new_block_ptr->end))
				{
				new_block_ptr->end = cur_block_ptr->end;
				}

			/* delete this block.  it may have overlapped/tocuhed the new one at the beginning,	*/
			/* the end or both	the new block now includes the union of its own data and	*/
			/* data in this block.								*/
			op_prg_list_remove (sacklist_ptr->entries, i);
			op_prg_mem_free (cur_block_ptr);

			/* adjust counters to account for newly deleted list entry */
			i--;
			num_sackblocks--;
			}

	if (tcp_trace_active || tcp_extns_trace_active)
			{
			op_prg_odb_print_major ("Updating the SACK list based on newly-received out-of-order segment.", OPC_NIL);
			tcp_sacklist_print ();
			}

	FOUT;
	}


static void
tcp_sacklist_clear ()
	{
	/** Removes all elements from the sacklist.  Performed after there are no longer packets held **/
	/** in the out-of-order list, i.e. all holes in received data have been filled in.		**/
	FIN (tcp_sacklist_clear ());

	if (sacklist_ptr->entries != OPC_NIL)
		{
		op_prg_list_free (sacklist_ptr->entries);
		}

	if (tcp_trace_active || tcp_extns_trace_active)
		{
		op_prg_odb_print_major ("Clearing the SACK list.", OPC_NIL);
		}
	FOUT;
	}

static void
tcp_sack_memory_free ()
	{
	/** Deallocates all memory related to the SACK option. **/
```

```
    FIN (tcp_sack_memory_free ());

    if (scoreboard_ptr != OPC_NIL)
        {
        if (scoreboard_ptr->entries != OPC_NIL)
            {
            op_prg_list_free (scoreboard_ptr->entries);
            op_prg_mem_free (scoreboard_ptr->entries);
            }

        op_prg_mem_free (scoreboard_ptr);
        }

    if (sacklist_ptr != OPC_NIL)
        {
        if (sacklist_ptr->entries != OPC_NIL)
            {
            op_prg_list_free (sacklist_ptr->entries);
            op_prg_mem_free (sacklist_ptr->entries);
            }

        op_prg_mem_free (sacklist_ptr);
        }
    FOUT;
    }


static void
tcp_ev_error (int ev_type, const char* state_name)
        {
        char        str0 [128];

        FIN (tcp_ev_error (ev_type, state_name));

        sprintf (str0, "TCP connection error in state (%s) - Illegal Event (%d)",
                state_name, ev_type);
        op_prg_odb_print_major (str0, OPC_NIL);

        FOUT;
        }


static void
tcp_connection_on_max_retrans_reset (void* PRG_ARG_UNUSED(input_ptr), int PRG_ARG_UNUSED(code))
        {
        char                log_string [128];
        char                msg_string [128];

        /** Send a reset to the peer connection and abort        **/
        FIN (tcp_connection_on_max_retrans_reset (void));

        /* Retransmit only if the surrounding node is not failed.         */
        /* Otherwise cancell all pending timers.                          */
        if (tcp_parameter_ptr->node_failed == OPC_TRUE)
                {
                FOUT;
                }

        /* Create a message string to indicate the current         */
        /* retransmission limits, based on the factor that          */
        /* is used to impose the limits.                            */
        if (max_retrans_mode == TcpC_Max_Retrans_Limit_By_Attempts)
                {
                sprintf (log_string, " %d attempts", max_retrans_attempts);
                }
```

```
	else
		{
		sprintf (log_string, " %4.2f seconds", max_retrans_interval);
		}

	/* Print out a trace message.					*/
	if (tcp_trace_active)
		{
		strcpy (msg_string, "The configured maximum limit is ");
		strcat (msg_string, log_string);
		op_prg_odb_print_major (" The maximum limit on retranmission tries has been reached",
							log_string, OPC_NIL);
		}

	/* Write out a log message indicating that attempt	*/
	/* to successfully transmit a Control or User Data	*/
	/* was unsuccessful and the connection is being		*/
	/* reset.									*/
	tcp_max_retransmit_limit_reached_log_write (log_string);

	/* Write the statistics for number of connections aborted by TCP.	*/
	op_stat_write (tcp_parameter_ptr->num_conn_rst_sent_stathandle, 1.0);

	/* Send the reset segment.					*/
	tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_RST);

	/* Abort the connection.					*/
	tcp_conn_abort ();

	FOUT;
	}

static void
tcp_conn_app_notify_open (const char* state)
	{
	/** Notify the application that this connection is open. **/
	FIN (tcp_conn_app_notify_open (state));

	/* Check if the application has already been notified.  */
	if (conn_estab == 0)
		{
		/* Set that the conenction opening indication has been	*/
		/* indicated to the application layer.			*/
		conn_estab = 1;

		/* Send an indication ICI to the application layer.	*/
		tcp_conn_app_indication (state, TCPC_IND_ESTAB);
		}

	FOUT;
	}

static void
tcp_conn_app_notify_received_close (const char* state, int status)
	{
	/** Notify application that this connection has received	**/
	/** CLOSE from its remote peer.					**/
	FIN (tcp_conn_app_notify_received_close (state, status));

	/* Check if the application has already been notified.  */
	if (tcp_app_notified_for_close_rcvd == OPC_FALSE)
		{
		/* Set that the conenction closing indication has been */
		/* delivered to the application layer.			*/
		tcp_app_notified_for_close_rcvd = OPC_TRUE;
```

```
		/* Send an indication ICI to the application layer.			*/
		tcp_conn_app_indication (state, status);
		}

	FOUT;
	}


static void
tcp_conn_app_notify_conn_close (const char* state, int status)
	{
	/** Notify the application that this connection is closed		**/
	/** or aborted (depending upon the status argument).		**/
	FIN (tcp_conn_app_notify_conn_close (state, status));

	/* Check if the application has already been notified.  */
	if (tcp_app_notified_for_conn_closed == OPC_FALSE)
		{
		/* Set that the conenction closing indication has been */
		/* delivered to the application layer.						*/
		tcp_app_notified_for_conn_closed = OPC_TRUE;

		/* Send an indication ICI to the application layer.			*/
		tcp_conn_app_indication (state, status);
		}

	FOUT;
	}


static void
tcp_conn_abort (void)
	{
	/** Destroy this process, checking for errors.				**/
	FIN (tcp_conn_abort ());

	/*	Notify the application that the TCP connection			*/
	/*	has aborted.											*/
	/*		1. Check to see if the application has already*/
	/*		   been notified of closure, if not, inform it. */
	/*		2. Cancel the self-interrupt scheduled in the  */
	/*		   TIME-WAIT state, if any.							*/
	/*		3. Destroy this connection process.				*/
	/* Check if the application has already been notified.  */
	if (tcp_app_notified_for_conn_closed == OPC_FALSE)
		{
		/* Set that the conenction closing indication has been */
		/* delivered to the application layer.						*/
		tcp_app_notified_for_conn_closed = OPC_TRUE;

		/* Send an indication ICI to the application layer.			*/
		tcp_conn_app_indication (tcb_ptr->state_name, TCPC_IND_ABORTED);
		}

	if (op_ev_valid (time_wait_evh) == OPC_TRUE)
		op_ev_cancel (time_wait_evh);

	/* Destroy the process model								*/
	tcp_conn_process_destroy ();

	FOUT;
	}

static void
```

```
tcp_conn_app_indication (const char* state, int status)
	{
	Ici*			tmp_ici_ptr;
	Evhandle		evh;
	char			msg [256];

	/** Provide an indication to the application that this connection is open. **/
	FIN (tcp_conn_app_indication (state, status));

	/* Create an ICI to carry indication information.		*/
	tmp_ici_ptr = op_ici_create ("tcp_status_ind");
	if (tmp_ici_ptr == OPC_NIL ||
		op_ici_attr_set (tmp_ici_ptr, "conn_id", tcb_ptr->conn_id) == OPC_COMPCODE_FAILURE ||
		op_ici_attr_set (tmp_ici_ptr, "status",  status)				== OPC_COMPCODE_FAILURE)
		{
		tcp_conn_warn ("Unable to create or initialize status indication ICI.", OPC_NIL, OPC_NIL);
		}
	else
		{
		op_ici_install (tmp_ici_ptr);
		evh = op_intrpt_schedule_remote (op_sim_time (), 0, tcb_ptr->app_objid);
		if (op_ev_valid (evh) == OPC_FALSE)
			tcp_conn_warn ("Unable to schedule remote interrupt at application.", OPC_NIL, OPC_NIL);
		}

	if (tcp_trace_active)
		{
		sprintf (msg, "Entering state %s.", state);
		if (strcmp (state, "CLOSE_WAIT") == 0 || strcmp (state, "FINWAIT1") == 0)
			op_prg_odb_print_major ("Connection closed.", msg, OPC_NIL);
		else if (strcmp (state, "TIME_WAIT") == 0)
			op_prg_odb_print_major ("Closing connection.", msg, OPC_NIL);
		else
			op_prg_odb_print_major ("Connection fully established.", msg, OPC_NIL);
		}

	FOUT;
	}

static void
tcp_sacklist_print ()
	{
	int						i, cnt;
	TcpT_Sackblock*			cur_block_ptr;
	char					str0 [64];

	FIN (tcp_sacklist_print ());

   op_prg_odb_print_major ("***** Selective Acknowledgment List *****", OPC_NIL);
	if (sacklist_ptr == OPC_NIL || sacklist_ptr->entries == OPC_NIL)
		{
		tcp_conn_warn ("Unable to print sacklist.", OPC_NIL, OPC_NIL);
		}

	cnt = op_prg_list_size (sacklist_ptr->entries);
	for (i = 0; i < cnt; i++)
		{
		cur_block_ptr = (TcpT_Sackblock *) op_prg_list_access (sacklist_ptr->entries, i);
		sprintf (str0, "start: %u; end: %u", cur_block_ptr->start, cur_block_ptr->end);
		op_prg_odb_print_minor (str0, OPC_NIL);
		}

	FOUT;
	}
```

```
static void
tcp_scoreboard_print ()
	{
	int								i, cnt;
	TcpT_Scoreboard_Entry*		cur_entry_ptr;
	char							str0 [256];

	FIN (tcp_scoreboard_print ());

	op_prg_odb_print_major ("***** Scoreboard *****", OPC_NIL);
	if (scoreboard_ptr == OPC_NIL || scoreboard_ptr->entries == OPC_NIL)
		{
		tcp_conn_warn ("Unable to print sacklist.", OPC_NIL, OPC_NIL);
		FOUT;
		}

	sprintf (str0, "Last retran %u; Recovery end %u; More retran %s\n",
					scoreboard_ptr->last_retran_end, scoreboard_ptr->recovery_end,
					scoreboard_ptr->more_retran ? "True" : "False");
	op_prg_odb_print_minor (str0, OPC_NIL);

	cnt = op_prg_list_size (scoreboard_ptr->entries);
	for (i = 0; i < cnt; i++)
		{
		cur_entry_ptr = (TcpT_Scoreboard_Entry *) op_prg_list_access (scoreboard_ptr->entries, i);
		sprintf (str0, "start: %u; end: %u; retran_end: %u", cur_entry_ptr->start,
						cur_entry_ptr->end, cur_entry_ptr->retran_end);
		op_prg_odb_print_minor (str0, OPC_NIL);
		}

	FOUT;
	}


static TcpT_Size
tcp_sack_number_sacked_bytes_find ()
	{
	int								i, cnt;
	TcpT_Size						sacked_bytes = 0;
	TcpT_Scoreboard_Entry*		cur_entry_ptr;

	/** Get the number of selective ACKed bytes.					**/
	/** This function is used to write statistics "SACKed Data."	**/
	FIN (tcp_sack_number_sacked_bytes_find ());

	cnt = op_prg_list_size (scoreboard_ptr->entries);

	for (i = 0; i < cnt; i++)
		{
		cur_entry_ptr = (TcpT_Scoreboard_Entry *) op_prg_list_access (scoreboard_ptr->entries, i);
		sacked_bytes  = sacked_bytes + cur_entry_ptr->end - cur_entry_ptr->start;
		}

	FRET (sacked_bytes);
	}

static void
tcp_connection_statistics_register (Boolean active_session)
	{
	char				appl_name [64];
	char				stat_annotate_str [2048];
	char				rem_addr_nodename [OMSC_HNAME_MAX_LEN] = "Unknown";

	static int		ete_delay_stat_dim_size			= -1;
	static int		seg_ete_delay_stat_dim_size		= -1;
```

```
	static int		rcv_seg_seq_no_stat_dim_size	= -1;
	static int		seg_ack_no_stat_dim_size		= -1;
	static int		sent_seg_seq_no_stat_dim_size	= -1;
	static int		sent_seg_ack_no_stat_dim_size	= -1;
	static int		cwnd_size_stat_dim_size		= -1;
	static int		mean_seg_rtt_stat_dim_size		= -1;
	static int		mean_seg_rtt_dev_stat_dim_size	= -1;
	static int		thruput_bytes_stat_dim_size	= -1;
	static int		thruput_packets_stat_dim_size	= -1;
	static int		thruput_bytes_sec_stat_dim_size	= -1;
	static int		thruput_packets_sec_stat_dim_size = -1;
	static int		remote_rcv_win_stat_dim_size	= -1;
	static int		rto_stat_dim_size				= -1;
	static int		in_flight_data_stat_dim_size	= -1;
	static int		sacked_data_stat_dim_size		= -1;
	static int		nagle_delay_stat_dim_size		= -1;
	static int		snd_wnd_delay_stat_dim_size	= -1;
	static int		cwnd_delay_stat_dim_size		= -1;
	static int		retrans_cnt_stat_dim_size		= -1;

	/** Registers all statistics maintained by this connection.	**/
	FIN (tcp_connection_statistics_register ());

	tcp_global_delay_handle		= op_stat_reg ("TCP.Delay (sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
	tcp_seg_global_delay_handle = op_stat_reg ("TCP.Segment Delay (sec)",			OPC_STAT_INDEX_NONE,
OPC_STAT_GLOBAL);
	tcp_global_retrans_count_handle = op_stat_reg ("TCP.Retransmission Count",			OPC_STAT_INDEX_NONE,
OPC_STAT_GLOBAL);
	tcp_delay_handle			= op_stat_reg ("TCP.Delay (sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	tcp_seg_delay_handle		= op_stat_reg ("TCP.Segment Delay (sec)",			OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
	packet_thru_handle				= op_stat_reg ("TCP.Traffic Received (packets)",	OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
	byte_thru_handle			= op_stat_reg ("TCP.Traffic Received (bytes)",		OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
	packet_sec_thru_handle		= op_stat_reg ("TCP.Traffic Received (packets/sec)", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
	byte_sec_thru_handle		= op_stat_reg ("TCP.Traffic Received (bytes/sec)",	OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
	retrans_count_handle		= op_stat_reg ("TCP.Retransmission Count",			OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);

	/* Cache the maximum indices for the dimensioned statistics used by	*/
	/* this process model.  These are not node-specific as the same		*/
	/* process model resides on all nodes (for that matter, even the */
	/* statistic declaration is not node-specific.)					*/
	if (ete_delay_stat_dim_size == -1)
		{
		/* Obtain decalared indices for all statistics.  */
		op_stat_dim_size_get ("TCP Connection.Delay (sec)",
	OPC_STAT_LOCAL, &ete_delay_stat_dim_size);
		op_stat_dim_size_get ("TCP Connection.Segment Delay (sec)",
	OPC_STAT_LOCAL, &seg_ete_delay_stat_dim_size);
		op_stat_dim_size_get ("TCP Connection.Received Segment Sequence Number",OPC_STAT_LOCAL,
&rcv_seg_seq_no_stat_dim_size);
		op_stat_dim_size_get ("TCP Connection.Received Segment Ack Number",			OPC_STAT_LOCAL,
&seg_ack_no_stat_dim_size);
		op_stat_dim_size_get ("TCP Connection.Sent Segment Sequence Number",		OPC_STAT_LOCAL,
&sent_seg_seq_no_stat_dim_size);
		op_stat_dim_size_get ("TCP Connection.Sent Segment Ack Number",				OPC_STAT_LOCAL,
&sent_seg_ack_no_stat_dim_size);
		op_stat_dim_size_get ("TCP Connection.Congestion Window Size (bytes)",		OPC_STAT_LOCAL,
&cwnd_size_stat_dim_size);
```

```
			op_stat_dim_size_get ("TCP Connection.Segment Round Trip Time (sec)",			OPC_STAT_LOCAL,
&mean_seg_rtt_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Segment Round Trip Time Deviation", OPC_STAT_LOCAL,
&mean_seg_rtt_dev_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Traffic Received (bytes)",			OPC_STAT_LOCAL,
&thruput_bytes_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Traffic Received (packets)",			OPC_STAT_LOCAL,
&thruput_packets_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Traffic Received (bytes/sec)",		OPC_STAT_LOCAL,
&thruput_bytes_sec_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Traffic Received (packets/sec)",		OPC_STAT_LOCAL,
&thruput_packets_sec_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Remote Receive Window Size (bytes)",		OPC_STAT_LOCAL,
&remote_rcv_win_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Retransmission Timeout (seconds)",	OPC_STAT_LOCAL,
&rto_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Flight Size (bytes)",				OPC_STAT_LOCAL,
&in_flight_data_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Selectively ACKed Data (bytes)",		OPC_STAT_LOCAL,
&sacked_data_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Send Delay (Nagle's) (sec)",			OPC_STAT_LOCAL,
&nagle_delay_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Send Delay (RCV-WND) (sec)",			OPC_STAT_LOCAL,
&snd_wnd_delay_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Send Delay (CWND) (sec)",			OPC_STAT_LOCAL,
&cwnd_delay_stat_dim_size);
			op_stat_dim_size_get ("TCP Connection.Retransmission Count",			OPC_STAT_LOCAL,
&retrans_cnt_stat_dim_size);
			}

		/* Prepare annotation strings, if at least one stat needs to be collected.	*/
		if (tcb_ptr->conn_id < ete_delay_stat_dim_size				|| tcb_ptr->conn_id < seg_ete_delay_stat_dim_size
		||
			tcb_ptr->conn_id < rcv_seg_seq_no_stat_dim_size			|| tcb_ptr->conn_id < seg_ack_no_stat_dim_size
		||
			tcb_ptr->conn_id < sent_seg_seq_no_stat_dim_size	|| tcb_ptr->conn_id < sent_seg_ack_no_stat_dim_size		||
			tcb_ptr->conn_id < cwnd_size_stat_dim_size				|| tcb_ptr->conn_id < mean_seg_rtt_stat_dim_size
			||
			tcb_ptr->conn_id < mean_seg_rtt_dev_stat_dim_size	|| tcb_ptr->conn_id < thruput_bytes_stat_dim_size		||
			tcb_ptr->conn_id < thruput_packets_stat_dim_size	|| tcb_ptr->conn_id < thruput_bytes_sec_stat_dim_size		||
			tcb_ptr->conn_id < thruput_packets_sec_stat_dim_size || tcb_ptr->conn_id < remote_rcv_win_stat_dim_size		||
			tcb_ptr->conn_id < rto_stat_dim_size					|| tcb_ptr->conn_id < in_flight_data_stat_dim_size
		||
			tcb_ptr->conn_id < sacked_data_stat_dim_size			|| tcb_ptr->conn_id < nagle_delay_stat_dim_size
		||
			tcb_ptr->conn_id < snd_wnd_delay_stat_dim_size			|| tcb_ptr->conn_id < cwnd_delay_stat_dim_size
			||
			tcb_ptr->conn_id < retrans_cnt_stat_dim_size			|| tcb_ptr->conn_id < retrans_cnt_stat_dim_size)
			{
			/* Determine the remote node's name using the remote IP address.		*/
			if (! inet_address_valid (tcb_ptr->rem_addr))
				{
				strcpy (rem_addr_nodename, "Passive Session");
				}
			else
				{
				ipnl_inet_addr_to_nodename (tcb_ptr->rem_addr, rem_addr_nodename);
				oms_tan_dotted_hname_to_underscores (rem_addr_nodename);
				}

			/* Obtain the application names and the node to which connection is made.		*/
			if (active_session == OPC_FALSE)
				{
				tcp_appl_name_from_rem_port_get ((GnaT_App)tcb_ptr->local_port, appl_name);
				sprintf (stat_annotate_str, "Conn %d [%s]: (Port %d) <-> %s (Port %d)",
```

```
				tcb_ptr->conn_id, appl_name, tcb_ptr->local_port, rem_addr_nodename, tcb_ptr->rem_port);
			}
		else
			{
			tcp_appl_name_from_rem_port_get ((GnaT_App)tcb_ptr->rem_port, appl_name);
			sprintf (stat_annotate_str, "Conn %d [%s]: (Port %d) <-> %s (Port %d)",
				tcb_ptr->conn_id, appl_name, tcb_ptr->local_port, rem_addr_nodename, tcb_ptr->rem_port);
			}

		/* Allocate memory to record statistics.	*/
		if (tcb_ptr->tcp_conn_stat_ptr == OPC_NIL)
			{
			tcb_ptr->tcp_conn_stat_ptr = (TcpT_Conn_Stats *) op_prg_mem_alloc (sizeof (TcpT_Conn_Stats));
			}
		}

	/* Register statistics, if within collectable limits; rename them for easier viewing.		*/
	if (tcb_ptr->conn_id < ete_delay_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->ete_delay_stathandle = op_stat_reg ("TCP Connection.Delay (sec)", tcb_ptr->conn_id,
OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->ete_delay_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < seg_ete_delay_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->seg_ete_delay_stathandle = op_stat_reg ("TCP Connection.Segment Delay (sec)", tcb_ptr-
>conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->seg_ete_delay_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < rcv_seg_seq_no_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->rcv_seg_seq_no_stathandle = op_stat_reg ("TCP Connection.Received Segment Sequence
Number", tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->rcv_seg_seq_no_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < seg_ack_no_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->rcv_seg_ack_no_stathandle = op_stat_reg ("TCP Connection.Received Segment Ack Num-
ber", tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->rcv_seg_ack_no_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < sent_seg_seq_no_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->sent_seg_seq_no_stathandle = op_stat_reg ("TCP Connection.Sent Segment Sequence Num-
ber", tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->sent_seg_seq_no_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < sent_seg_ack_no_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->sent_seg_ack_no_stathandle = op_stat_reg ("TCP Connection.Sent Segment Ack Number",
tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->sent_seg_ack_no_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < cwnd_size_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->cwnd_size_stathandle = op_stat_reg ("TCP Connection.Congestion Window Size (bytes)",
tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->cwnd_size_stathandle, stat_annotate_str);
		}
```

```
	if (tcb_ptr->conn_id < mean_seg_rtt_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->mean_seg_rtt_stathandle = op_stat_reg ("TCP Connection.Segment Round Trip Time (sec)", 
tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->mean_seg_rtt_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < mean_seg_rtt_dev_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->mean_seg_rtt_dev_stathandle = op_stat_reg ("TCP Connection.Segment Round Trip Time 
Deviation", tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->mean_seg_rtt_dev_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < thruput_bytes_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_stathandle = op_stat_reg ("TCP Connection.Traffic Received (bytes)", tcb_ptr-
>conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < thruput_packets_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->thruput_packets_stathandle = op_stat_reg ("TCP Connection.Traffic Received (packets)", 
tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->thruput_packets_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < thruput_bytes_sec_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_sec_stathandle = op_stat_reg ("TCP Connection.Traffic Received (bytes/sec)", 
tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->thruput_bytes_sec_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < thruput_packets_sec_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->thruput_packets_sec_stathandle = op_stat_reg ("TCP Connection.Traffic Received (packets/
sec)", tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->thruput_packets_sec_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < remote_rcv_win_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->remote_rcv_win_stathandle = op_stat_reg ("TCP Connection.Remote Receive Window Size 
(bytes)", tcb_ptr->conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->remote_rcv_win_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < rto_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->rto_stathandle = op_stat_reg ("TCP Connection.Retransmission Timeout (seconds)", tcb_ptr-
>conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->rto_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < in_flight_data_stat_dim_size)
		{
		tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle = op_stat_reg ("TCP Connection.Flight Size (bytes)", tcb_ptr-
>conn_id, OPC_STAT_LOCAL);
		op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle, stat_annotate_str);
		}

	if (tcb_ptr->conn_id < sacked_data_stat_dim_size)
```

```
			{
			tcb_ptr->tcp_conn_stat_ptr->sacked_data_stathandle = op_stat_reg ("TCP Connection.Selectively ACKed Data (bytes)",
tcb_ptr->conn_id, OPC_STAT_LOCAL);
			op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->sacked_data_stathandle, stat_annotate_str);
			}

		if (tcb_ptr->conn_id < nagle_delay_stat_dim_size)
			{
			tcb_ptr->tcp_conn_stat_ptr->nagle_delay_stathandle = op_stat_reg ("TCP Connection.Send Delay (Nagle's) (sec)", tcb_ptr-
>conn_id, OPC_STAT_LOCAL);
			op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->nagle_delay_stathandle, stat_annotate_str);
			}

		if (tcb_ptr->conn_id < snd_wnd_delay_stat_dim_size)
			{
			tcb_ptr->tcp_conn_stat_ptr->snd_wnd_delay_stathandle = op_stat_reg ("TCP Connection.Send Delay (RCV-WND) (sec)",
tcb_ptr->conn_id, OPC_STAT_LOCAL);
			op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->snd_wnd_delay_stathandle, stat_annotate_str);
			}

		if (tcb_ptr->conn_id < cwnd_delay_stat_dim_size)
			{
			tcb_ptr->tcp_conn_stat_ptr->cwnd_delay_stathandle = op_stat_reg ("TCP Connection.Send Delay (CWND) (sec)", tcb_ptr-
>conn_id, OPC_STAT_LOCAL);
			op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->cwnd_delay_stathandle, stat_annotate_str);
			}

		if (tcb_ptr->conn_id < retrans_cnt_stat_dim_size)
			{
			tcb_ptr->tcp_conn_stat_ptr->retrans_count_stathandle = op_stat_reg ("TCP Connection.Retransmission Count", tcb_ptr-
>conn_id, OPC_STAT_LOCAL);
			op_stat_annotate (tcb_ptr->tcp_conn_stat_ptr->retrans_count_stathandle, stat_annotate_str);
			}

		FOUT;
		}


static void
tcp_seg_send_delay_stat_record ()
		{
		/** Collect statistics related to delays in sending segments		**/
		FIN (tcp_seg_send_delay_stat_record ());

		if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL)
			{
			/* Write the number of unacknowledged data.		*/
			if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle) == OPC_TRUE)
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->in_flight_data_stathandle, (double) snd_max - snd_una);

			/* Since we are sending data, write and reset delay statistics.	*/
			if (nagle_limit_time > 0.0 &&
				op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->nagle_delay_stathandle) == OPC_TRUE)
				{
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->nagle_delay_stathandle, op_sim_time() - nagle_limit_time);
				nagle_limit_time = 0.0;
				}

			if (snd_wnd_limit_time > 0.0 &&
				op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->snd_wnd_delay_stathandle) == OPC_TRUE)
				{
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->snd_wnd_delay_stathandle, op_sim_time() - snd_wnd_limit_time);
				snd_wnd_limit_time = 0.0;
				}
```

```
			if (cwnd_limit_time > 0.0 &&
				op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->cwnd_delay_stathandle) == OPC_TRUE)
				{
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->cwnd_delay_stathandle, op_sim_time() - cwnd_limit_time);
				cwnd_limit_time = 0.0;
				}
			}

	FOUT;
	}


static void
tcp_retrans_timeout_check_and_schedule ()
	{
	double		next_timeout_time;

	/** The function checks for a already scheduled retransmission timer	**/
	/** or if the maximum number of retransmissions have been reached. If	**/
	/** so then it simple returns otherwise it schedules a retransmission	**/
	/** timer for the current event. Generally retransmission timers are	**/
	/** scheduled for SYN, DATA and FIN segments only.			**/
	FIN (tcp_retrans_timeout_check_and_schedule ());

	/* Set a retransmission timeout, if necessary.	*/
	if ((!op_ev_valid (retrans_evh) || !op_ev_pending (retrans_evh)) &&
		(!op_ev_valid (max_retrans_evh) || !op_ev_pending (max_retrans_evh)))
		{
		/* Record the current RTO value.		*/
			if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL &&
				op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->rto_stathandle) == OPC_TRUE)
				{
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->rto_stathandle, current_rto);
				}

		/* Compute the next RTO expiration time.	*/
		next_timeout_time = Tcp_Slowtimo_Next_Timeout_Time_Obtain (op_sim_time (), current_rto, timer_gran);

		/* Schedule the retransmission timeout.		*/
		retrans_evh = op_intrpt_schedule_call (next_timeout_time, 0, tcp_timeout_retrans, OPC_NIL);

		if (op_ev_valid (retrans_evh) == OPC_FALSE)
			tcp_conn_warn ("Unable to schedule retransmission timeout.",
				"No retransmission will take place.", OPC_NIL);
		}

	FOUT;
	}

static Boolean
tcp_conn_app_open_indicate ()
	{
	Boolean				accept_status;
	Ici*				app_open_ind_ici_ptr;

	/* Let the application know that a SYN has been received.	*/
	/* Create the Open Indication ICI that will be sent	to the	*/
	/* higher layer.							*/
	FIN (tcp_conn_app_open_indicate ());

	/* Create an ICI.				*/
	/* The ici format to be used depends on whether or not the	*/
	/* higher layer supports InetT_Address structure.		*/
	if (tcb_ptr->inet_support)
		{
```

```
			app_open_ind_ici_ptr = op_ici_create ("tcp_open_ind_inet");

			/* Set the remote address in the ici.					*/
			op_ici_attr_set (app_open_ind_ici_ptr, "rem addr", &(tcb_ptr->rem_addr));
			}
		else
			{
			app_open_ind_ici_ptr = op_ici_create ("tcp_open_ind");

			/* Convert the address into the IpT_Address format			*/
			/* before setting it in the ici.						*/
			op_ici_attr_set (app_open_ind_ici_ptr, "rem addr",
				inet_ipv4_address_get (tcb_ptr->rem_addr));
			}

		/* Set the remote address, remote port and the conn ID.		*/
		op_ici_attr_set (app_open_ind_ici_ptr, "conn id",  tcb_ptr->conn_id);
		op_ici_attr_set (app_open_ind_ici_ptr, "rem port", tcb_ptr->rem_port);

		/* Install the ICI.				*/
		op_ici_install (app_open_ind_ici_ptr);

		/* Send a forced interrupt to the application.	*/
		op_intrpt_force_remote (TCPC_IND_CONNECT_REQUEST, tcb_ptr->app_objid);

		/* Once the application process has completed processing			*/
		/* control returns to the TCP layer and by now the application */
		/* would have indicated whether to accept or reject the connection.*/
		op_ici_attr_get (app_open_ind_ici_ptr, "accept status", &accept_status);

		/* Destroy the ICI.		*/
		op_ici_destroy (app_open_ind_ici_ptr);

		FRET (accept_status);
		}


static void
tcp_new_reno_retransmit (void)
		{
		char						msg [128];
		TcpT_Seq					onxt, cwnd_old;

		/** This function is called when New Reno is used and a partial ACK **/
		/** (ACK advancing snd_una) is received.							**/
		/** As specified in RFC 2582, the first unacknowledged segment is	**/
		/** retransmitted, and the congestion window is deflated by the		**/
		/** amount of new data acknowledged, and one MSS is added.			**/
		/** If permitted by the new value of cwnd, next segment is sent.		**/
		FIN (tcp_new_reno_retransmit ());

	if (tcp_trace_active || tcp_retransmission_trace_active)
			{
			sprintf (msg, "<SND.UNA = %u> <RTT = %g> <RTO = %g> <backoff = %d>",
				snd_una, retrans_rtt, current_rto, (int) retrans_backoff);
			op_prg_odb_print_major ("Partial ACK was received for New Reno", msg, OPC_NIL);
			}

		/* If RTT measurements are currently being taken, reset the timer.		*/
		rtt_active = 0;

	/* Retransmit the first unacknowledged segment. This will be done calling	*/
		/* tcp_una_buf_process (). Temporarily set the value of snd_nxt, so that		*/
		/* the next sent packet is indeed the first unACKed packet. We will then		*/
		/* reset snd_nxt back to its original value. To sent only one segment,		*/
```

```
	/* temporarily set the cwnd value to 1 MSS.							*/

	/* Store current snd_nxt value. This is being done as when we call una_buf_process	*/
	/* we need to start sending from snd_una, rather than snd_nxt. After the function*/
	/* call, values will be restored.
			*/
onxt	= snd_nxt;
snd_nxt		= snd_una;

/* Store current congestion window value. This is done to send just one segment.		*/
	cwnd_old		= cwnd;
cwnd		= snd_mss;

/* Retransmit the missing packet.  Only one will be transmitted due to cwnd.			*/
tcp_una_buf_process (OPC_FALSE);

	/* Restore the value of send_next.	*/
	snd_nxt = MAX(snd_nxt, onxt);

	/* Restore the congestion window value.	*/
	cwnd = cwnd_old;

	/* Reset the number of duplicate acknowledgements.  */
	dup_ack_cnt = 0;

/* Store the sequence number of the retransmitted segment.  */
/* Account for the length of the retransmitted packet.				*/
max_retrans_seq = snd_una + seg_len - 1;

	/* Collect statistics related to delays in sending segments		*/
	tcp_seg_send_delay_stat_record ();


	FOUT;
	}


/***** Timestamp-related functions *****/

static void
tcp_ts_info_process (Packet* pk_ptr)
	{
	TcpT_Seg_Option_TS*		rcvd_timestamp_ptr;
	double				measurement_start_time;
	char				str0 [256];
	char				str1 [256];

	/** Process the incoming packet w.r.t. information	**/
	/** available in the Timestamp option field.		**/
	FIN (tcp_ts_info_process (pk_ptr));

	/* Get the timestamp from the sender.		*/
	op_pk_nfd_access (pk_ptr, "Timestamp Option", &rcvd_timestamp_ptr);

	/* Check whether this is the next expected segment  The second			*/
	/* check also ensures PAWS (protection against wrapped sequences).	*/
	if (seg_len != 0 &&
		tcp_seq_ge (rcvd_timestamp_ptr->timestamp_value, timestamp_info.ts_recent) &&
		tcp_seq_le (seg_seq, timestamp_info.last_ack_sent))
		{
		/** Please note that this is a modified check from RFC 1323, since	**/
		/** the original algorithm was flawed. We are performing a check	**/
		/** according to TCP/IP Illustrated, Volume 2 by Wright and Stevens,**/
		/** page 870, Figure 26.2.								**/
```

```
			/* This is the next expected segment. Store the new timestamp value.	*/
			timestamp_info.ts_recent = rcvd_timestamp_ptr->timestamp_value;

			/* Print debugging/trace messages, if enabled.		*/
			if (tcp_extns_trace_active)
				{
				sprintf (str0, "Storing the timestamp (%u) from the remote peer.", timestamp_info.ts_recent);
				sprintf (str1, "Updating RTT measurements with echoed value: (%u).", rcvd_timestamp_ptr->timestamp_echo);

				op_prg_odb_print_major ("TCP received the next expected segment and ACK for new data.",
					str0, str1, OPC_NIL);
				}

			/* Update RTT measurements.		*/
			measurement_start_time = (double) ((rcvd_timestamp_ptr->timestamp_echo * (tcp_parameter_ptr-
>timestamp_clock/1000.0)) + conn_start_time);
			tcp_rtt_measurements_update (measurement_start_time);
			}
		else if (seg_len == 0)
			{
			if (tcp_extns_trace_active)
				{
				sprintf (str0, "timestamp stays at (%u).\n", timestamp_info.ts_recent);
				sprintf (str1, "Updating RTT measurements with echoed value: (%u).", rcvd_timestamp_ptr->timestamp_echo);

				op_prg_odb_print_major ("TCP received an ACK for new data.",
					"The segment does not contain any data.", str0, str1, OPC_NIL);
				}

			/* Update RTT measurements.		*/
			measurement_start_time = (double) ((rcvd_timestamp_ptr->timestamp_echo * (tcp_parameter_ptr-
>timestamp_clock/1000.0)) + conn_start_time);
			tcp_rtt_measurements_update (measurement_start_time);
			}

		FOUT;
		}


static void
tcp_conn_timestamp_init (Packet* seg_ptr, Boolean update_rtt)
		{
		double						measurement_start_time;
		TcpT_Seg_Option_TS*			rcvd_timestamp_ptr;

		/** Itilialize DS containing information related to timestamp option.	**/
		/** This function is called only if Timestamp option is supported for	**/
		/** the connection.
			**/
		FIN (tcp_conn_timestamp_init (seg_ptr, update_rtt));

		if (tcp_trace_active || tcp_extns_trace_active)
			{
			op_prg_odb_print_major ("Both sides of this connection support Timestamp option.", OPC_NIL);
			}

		/* Get the timestamp from the sender.							*/
		op_pk_nfd_access (seg_ptr, "Timestamp Option", &rcvd_timestamp_ptr);

		/* Initialize last sent ACK. */
		timestamp_info.last_ack_sent = rcv_nxt;

		/* Store the timestamp set by the remote peer.						*/
		timestamp_info.ts_recent = rcvd_timestamp_ptr->timestamp_value;
```

```
	if (update_rtt == OPC_TRUE)
		{
		/* Calculate the initial estimate of RTT.				*/
		measurement_start_time = (double) ((rcvd_timestamp_ptr->timestamp_echo * (tcp_parameter_ptr-
>timestamp_clock/1000.0)) + conn_start_time);
		tcp_rtt_measurements_update (measurement_start_time);
		}

	/* Turn off RTT calculation done once on per round-trip */
	/* time. timestamps will be used instead.				*/
	rtt_active		= 1;
	rtt_seq	= iss;

	FOUT;
	}

static void
tcp_timeout_ev_cancel (void)
	{
	/** Cancell all pending timeout events.		**/
	FIN (tcp_timeout_ev_cancel ());

	/* Clear all timeouts. */
	if (op_ev_valid (ack_evh) && op_ev_pending (ack_evh))
		op_ev_cancel (ack_evh);
	if (op_ev_valid (retrans_evh) && op_ev_pending (retrans_evh))
		op_ev_cancel (retrans_evh);
	if (op_ev_valid (max_retrans_evh) && op_ev_pending (max_retrans_evh))
		op_ev_cancel (max_retrans_evh);
	if (op_ev_valid (persist_evh) && op_ev_pending (persist_evh))
		op_ev_cancel (persist_evh);
	if (op_ev_valid (time_wait_evh) && op_ev_pending (time_wait_evh))
		op_ev_cancel (time_wait_evh);
	FOUT;
	}


/**** Error handling functions *****/

static void
tcp_conn_error (const char* msg0, const char* msg1, const char* msg2)
	{
	/** Print an error message and exit the simulation. **/
	FIN (tcp_conn_error (msg0, msg1, msg2));

	op_sim_end ("Error in TCP socket process (tcp_conn_v3):", msg0, msg1, msg2);

	FOUT;
	}

static void
tcp_conn_warn (const char* msg0, const char* msg1, const char* msg2)


	{
	/** Print a warning message and resume. **/
	FIN (tcp_conn_warn (msg0, msg1, msg2));

	op_prg_odb_print_major ("Warning from TCP socket process (tcp_conn_v3):",
		msg0, msg1, msg2, OPC_NIL);

	FOUT;
	}
```

```
/**** Passive Attribute handling functions ****/

static void
tcp_conn_attr_set_passive_as_disabled ()

	{
	/** There are currently four attributes that may be configured   **/
	/** as "Passive".  This function is called BEFORE the first SYN        **/
	/** of a TCP connection is sent, and any of these attributes       **/
	/** that are configured as "Passive" must be set to disabled.      **/
	FIN (tcp_conn_attr_set_passive_as_disabled ());

	if (sack_enabled == TCPC_OPTION_STATUS_PASSIVE)
		{
		if (tcp_trace_active || tcp_extns_trace_active)
			{
			op_prg_odb_print_minor ("Disabled SACK for the connection.",
				"after SYN message with an unsupported SACK",
				"option has been received.", OPC_NIL);
			}

		sack_enabled = TCPC_OPTION_STATUS_DISABLED;
		}

	if (conn_supports_ts == TCPC_OPTION_STATUS_PASSIVE)
		{
		if (tcp_trace_active || tcp_extns_trace_active)
			{
			op_prg_odb_print_minor ("Disabled timestamp for the connection.",
				"after SYN message with an unsupported timestamp",
				"option has been received.", OPC_NIL);
			}

		conn_supports_ts = TCPC_OPTION_STATUS_DISABLED;
		}

	if (window_scaling_enabled == TCPC_OPTION_STATUS_PASSIVE)
		{
		if (tcp_trace_active || tcp_extns_trace_active)
			{
			op_prg_odb_print_minor ("Disabled window scaling for the connection.",
				"after SYN message with an unsupported window scale",
				"option has been received.", OPC_NIL);
			}

		window_scaling_enabled = TCPC_OPTION_STATUS_DISABLED;
		}

	/* There is no need to assign ECN capability as disabled since */
	/* there is no SV for it.  If it is passive, then write out a          */
	/* message stating that it will not be enabled.                                   */
	if (tcp_parameter_ptr->ecn_capability == TCPC_OPTION_STATUS_PASSIVE)
		{
		if (tcp_trace_active || tcp_extns_trace_active)
			{
			op_prg_odb_print_minor ("Disabled ECN for the connection.",
				"after SYN message with an unsupported ECN",
				"option has been received.", OPC_NIL);
			}
		}


	/* Configuration of window scaling needs to be done                        */
	/* here rather than in the initialization state since the  */
	/* case where it may be passive needs to be handled.               */
```

```
	/* In this case, if window scaling was configured as		*/
	/* passive it has been disabled.						*/
	if ((window_scaling_enabled == TCPC_OPTION_STATUS_DISABLED) && (rcv_buff > TCPC_MAX_WND_SIZE))
		{
		/* Window scaling is disabled but the receive buffer specified */
		/* greater than the value which the 16-bit field can carry.	*/
		/* Truncate the receive buffer specification, and write a		*/
		/* simulation log message.								*/
		tcp_rcvbuff_truncate_log_write (rcv_buff, TCPC_MAX_WND_SIZE);
		rcv_buff = TCPC_MAX_WND_SIZE;
		}

	else if ((window_scaling_enabled == TCPC_OPTION_STATUS_ENABLED) && (rcv_buff > TCPC_MAX_WND_WWS_SIZE))
		{
		tcp_rcvbuff_truncate_log_write (rcv_buff, TCPC_MAX_WND_WWS_SIZE);
		rcv_buff = TCPC_MAX_WND_WWS_SIZE;
		}

	/* Initial receive window size is the full buffer size. */
	rcv_wnd = rcv_buff;

	/* The ssthresh is either set to this value based on TCP/IP Illustrated vol2, pg 835 */
	if (window_scaling_enabled == TCPC_OPTION_STATUS_ENABLED)
		ssthresh = TCPC_MAX_WND_WWS_SIZE;
	else
		ssthresh = TCPC_MAX_WND_SIZE;

	FOUT;
	}

static void
tcp_conn_frfr_reset ()
	{
	/** This function is called when a connection performs a reset **/
	/** and returns to the listen state.  When SACK is enabled,	**/
	/** FR/FR may be enabled even though it was not configured to		**/
	/** as such.  This function will reinitialize the tcp_flavor	**/
	/** SV if sack_enabled is true.							**/

	Boolean					tcp_fast_retransmit_enabled;
	Boolean					tcp_fast_recovery_enabled;

	FIN (tcp_conn_frfr_reset ());

	/* Obtain the values of the FRFR attributes from the PTC memory.		*/
	tcp_fast_retransmit_enabled = tcp_parameter_ptr->fast_retransmit_flag;
	tcp_fast_recovery_enabled		= tcp_parameter_ptr->fast_recovery_mode;

	/* FR/FR may have been enabled, even though it was not configured		*/
	/* to be, if sack_enabled is set to TRUE.						*/
	/* If sack_enabled is true, then re-initialize FR/FR				*/
	if (sack_enabled == TCPC_OPTION_STATUS_ENABLED)
		{
		if (tcp_fast_retransmit_enabled)
			{
			if (tcp_fast_recovery_enabled == TcpC_Fast_Recovery_Reno)
				{
				/* Both Fast Retransmit & Fast Recovery is enabled. This implies*/
				/* the TCP flavor in use to be "TCP Reno".				*/
				tcp_flavor = TcpC_Flavor_Reno;
				}
			else if (tcp_fast_recovery_enabled == TcpC_Fast_Recovery_New_Reno)
				{
				tcp_flavor = TcpC_Flavor_New_Reno;
				}
```

```
				else
					{
					/* Only Fast Retransmit is enabled along with slow start and	*/
					/* congestion avoidance. The TCP flavor thus would be "TCP Tahoe".*/
					tcp_flavor = TcpC_Flavor_Tahoe;
					}
				}
			else
				{
				/* "TCP Reno" only operates over "TCP Tahoe". Forcibly enable			*/
				/* Fast Retransmit even if it is disabled.								*/
				if (tcp_fast_recovery_enabled)
					{
					if (tcp_trace_active || tcp_extns_trace_active)
						{
						op_prg_odb_print_minor ("Fast Retransmit is automatically being enabled",
							"when Fast Retransmit is enabled.", OPC_NIL, OPC_NIL);
						}

					tcp_reno_without_tahoe_log_write ();
					tcp_flavor = TcpC_Flavor_Reno;
					}
				else
					{
					/* Neither Fast Retransmit nor Fast Recovery is enabled.			*/
					tcp_flavor = TcpC_Flavor_Basic;
					}
				}
			}


	FOUT;
	}


static void
tcp_seg_rcvd_stat_write (void)
	{
	double				segment_delay;

	/** Records the following connection statistics after a packet has been received:	**/
	/** TCP Connection.Received Segment Sequence Number		**/
	/** TCP Connection.Segment Delay.								**/
	/** TCP.Segment Delay (local stat)							**/
	/** TCP. Segment Delay (global stat)							**/
	FIN (tcp_seg_rcvd_stat_write ());

	segment_delay = op_sim_time () - op_pk_stamp_time_get (ev_ptr->pk_ptr);
	if (tcb_ptr->tcp_conn_stat_ptr != OPC_NIL)
		{
		if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->rcv_seg_seq_no_stathandle))
			op_stat_write (tcb_ptr->tcp_conn_stat_ptr->rcv_seg_seq_no_stathandle, (double) seg_seq);

		if (op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->seg_ete_delay_stathandle))
			op_stat_write (tcb_ptr->tcp_conn_stat_ptr->seg_ete_delay_stathandle, segment_delay);
		}
	op_stat_write (tcp_seg_delay_handle, segment_delay);
	op_stat_write (tcp_seg_global_delay_handle, segment_delay);

	FOUT;
	}

static int
tcp_conn_option_size_get (void)
	{
```

```
	int							sack_count;
   int						included_sack_count;	/* the number of sack blocks which will go into the option */
	int							option_field_size = 0;

	/** Get the size of option fields (in bytes) in the TCP segment header.	**/
	/** Currently, there are 2 options used: SACK and Timestamp.				**/
	FIN (tcp_conn_option_size_get (void));

	if (conn_supports_ts == TCPC_OPTION_STATUS_ENABLED)
		{
		/* This connection supports timestamp option.						*/
		/* Add this option field size to the total option field size.	*/
		option_field_size = TCPC_SEG_TIMESTAMP_SIZE;
		}

	if (SACK_PERMITTED && (op_prg_list_size (rcv_rec_list) > 0))
		{
		/* Connection supports SACK and there are out-of-order packets.	*/

		/* Get the number of entries in the SACK list.	*/
		sack_count = op_prg_list_size (sacklist_ptr->entries);

		/* The option field can carry maximum of TCPC_SACK_MAX_BLOCKS	*/
		/* blocks. Make sure this will not be exceeded.					*/
		if (conn_supports_ts == TCPC_OPTION_STATUS_ENABLED)
			{
			/* Additionally, if a connection supports timestamp option,	*/
			/* this option will also use the option fields. This will */
			/* further decrease the number of SACK blocks that can be	*/
			/* carried in the header.										*/
			included_sack_count = MIN (sack_count, TCPC_SACK_MAX_BLOCKS - 1);
			}
		else
			{
			included_sack_count = MIN (sack_count, TCPC_SACK_MAX_BLOCKS);
			}

		/* Add size of SACK option field to the total option field size. */
		option_field_size = option_field_size + TCPC_KIND_LENGTH_BLOCK_SIZE + TCPC_SACK_BLOCK_SIZE * in-
cluded_sack_count;
		}

	FRET (option_field_size);
	}


static void
tcp_conn_retrans_stat_write (void)
	{

	/** Writes TCP retransmission statistics.	**/
	FIN (tcp_conn_retrans_stat_write (void));

	op_stat_write (tcp_global_retrans_count_handle, 1.0);
	op_stat_write (retrans_count_handle, 1.0);

	if ((tcb_ptr->tcp_conn_stat_ptr != OPC_NIL) &&
		(op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->retrans_count_stathandle) == OPC_TRUE))
		{
		op_stat_write (tcb_ptr->tcp_conn_stat_ptr->retrans_count_stathandle, 1.0);
		}

	FOUT;
	}
```

```
static void
tcp_conn_rcv_buff_adjust (void)
	{
	int				mss_in_wnd_cnt;

	/** Recalculate receive window according to specified		**/
	/** "Receive Window Adjustment" parameter.				**/
	/** This is to model architecture-specific behavior when**/
	/** receive window is adjusted to the multiples of mss		**/
	/** depending on system architecture (Windows vs.		**/
	/** Solaris.										**/
	FIN (tcp_conn_rcv_buff_adjust ());

	if (tcp_parameter_ptr->rcv_buff_adj == TcpC_Architecture_Windows)
		{
		mss_in_wnd_cnt = ceil (((double) rcv_buff)/snd_mss);

		/* When setting receive window, Windows will round		*/
		/* the configured value to even increments of MSS.	*/
		if (fmod (mss_in_wnd_cnt, 2) == 0)
			{
			/* We already have an even number of increments of MSS.	*/
			rcv_buff = snd_mss * mss_in_wnd_cnt;
			}
		else
			{
			rcv_buff = snd_mss * (mss_in_wnd_cnt + 1);
			}
		}

	else if (tcp_parameter_ptr->rcv_buff_adj == TcpC_Architecture_Solaris)
		{
		/* When setting receive window, Solaris will round	*/
		/* the configured value up to multiples of MSS.		*/
		mss_in_wnd_cnt = ceil (((double) rcv_buff)/snd_mss);

		rcv_buff = snd_mss * mss_in_wnd_cnt;
		}

	FOUT;
	}

static TcpT_Size
tcp_conn_mss_auto_assign (InetT_Address rem_addr)
	{
   IpT_Interface_Info*			ip_interface_ptr;
	IpT_Rte_Module_Data*	ip_module_data_ptr;
	InetT_Addr_Family			addr_family;

	/* Setting the default MSS setting as Ethernet MTU	*/
	TcpT_Size					mss = 1460;
	TcpT_Size					mtu;
	Objid						own_mod_objid, own_node_objid;

	/** This function discover's the IP module in the current node and scans		**/
	/** interface table. If the interface table is non-zero, then it returns the**/
	/** Maximum Transfer Unit (MTU) of the first available interface. If no			**/
	/** interface are configured then the MTU of ethernet is returned.				**/
	/** Based on MTU is then calculates the maximum segment size that can be	**/
	/** transferred over link with the found MTU. Finally it recalculates			**/
	/** variables that depend on MSS - receive buffer and initial window size.	**/
	FIN (tcp_conn_mss_auto_assign (rem_addr));

	/* Find out whether we are dealing with an IPv4 packet or an IPv6 packet.	*/
```

```
	addr_family = inet_address_family_get (&rem_addr);

	/* Get the object ID of the node.      */
	own_mod_objid = op_id_self ();

	/* Obtain the node's objid. */
	own_node_objid = op_topo_parent (own_mod_objid);

	/* Get a pointer to the ip module data of this node.                              */
	ip_module_data_ptr = ip_support_module_data_get (own_node_objid);

/* Flag an error if more than one IP module was found. */
if (OPC_NIL == ip_module_data_ptr)
    {
    /* Having more than one IP module is a serious error.  End simulation. */
    tcp_conn_error ("Unable to locate the ip module in the node.", OPC_NIL, OPC_NIL);
    }
	else
		{
		if (inet_rte_num_interfaces_get (ip_module_data_ptr) > 0)
			{
			ip_interface_ptr = inet_rte_intf_tbl_access (ip_module_data_ptr, 0);

			/* Obtain the Maximum Transfer Unit from the IP interfaces    */
			mtu = inet_rte_intf_mtu_get (ip_interface_ptr, addr_family);

			/* Convert the MTU to the MSS supported by TCP.                   */
			/* This assumes TCP/IP not using any options.                     */
			if (InetC_Addr_Family_v4 == addr_family)
				{
				mss = mtu - (IPC_DGRAM_HEADER_LEN_BYTES + 20);
				}
			else
				{
				mss = mtu - (IPV6C_DGRAM_HEADER_LEN_BYTES + 20);
				}
			}
		else
			{
			tcp_conn_error ("This node does not have any IP interfaces.", OPC_NIL, OPC_NIL);
			}
		}

	/* Also assing snd_mss.     */
	snd_mss = mss;

	/* Set the receive window size -- it is the amount of    */
	/* receive data (in bytes) that can be buffered at one   */
	/* time on a connection. The sending host can send only          */
	/* that amount of data before waiting for an ACK and*/
	/* window update from the receiving host. Check if the           */
	/* receive buffer is set to be computed dynalically.     */
	if (tcp_parameter_ptr->rcv_buff_size == -1)
		{
		/* When set to "Default", this parameter is set to at     */
		/* least four times the negotiated MSS, with a maximum          */
		/* size of 64 KB.                                                    */
		rcv_buff = (4 * snd_mss <= TCPC_MAX_WND_SIZE) ? 4 * snd_mss : TCPC_MAX_WND_SIZE;
		}

	/* Determine the size of the initial window used during slow    */
	/* start. Slow-start occurs in as many as three different ways:   */
	/*  1. start a new connection                                          */
	/*  2. restart connection after a long idle period                  */
	/*  3. restart after an idle timeout                                */
```

```
	/* The initial window size is only used for cases 1. and 2.			*/
	if (tcp_parameter_ptr->slow_start_initial_count == -1)
		{
		/* Set the initial window wise should be set as defined in		*/
		/* RFC-2414.									*/
		initial_window_size = MIN(4*snd_mss, MAX(2*snd_mss, 4380));
		}
	else
		{
		/* Set the initial window size as a multiple of the MSS.		*/
		initial_window_size = snd_mss * tcp_parameter_ptr->slow_start_initial_count;
		}

	cwnd = initial_window_size;

	FRET (mss);
	}

static void
tcp_conn_process_destroy (void)
	{
	/** This function will be called when you are destroying this	**/
	/** process model. It will free the memory associated with it	**/
	/** and destroy the process model						**/
	FIN (tcp_conn_process_destroy (void));

	/*	Call procedure to deallocate memory tcp connection record	*/
	/*	in the llm transport connection table if this node in		*/
	/*	a LAN node.
	*/
	if (my_lanhandle != OPC_NIL)
		{
		llm_trans_conn_record_free (my_lanhandle, sess_wkstn_id, tcb_ptr->rem_port, tcb_ptr->local_port);
		}

	if (op_pro_destroy (op_pro_self ()) == OPC_COMPCODE_FAILURE)
		tcp_conn_error ("Unable to destroy socket process.", OPC_NIL, OPC_NIL);

	/* Deallocate memory related to the SACK Option.				*/
	if (SACK_PERMITTED)
		{
		tcp_sack_memory_free ();
		}

	FOUT;
	}

/* End of Function Block */

/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.			*/
#undef FIN_TRACING
#define FIN_TRACING

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

#if defined (__cplusplus)
extern "C" {
#endif
	void tcp_conn_v3 (OP_SIM_CONTEXT_ARG_OPT);
	VosT_Obtype _op_tcp_conn_v3_init (int * init_block_ptr);
	VosT_Address _op_tcp_conn_v3_alloc (VOS_THREAD_INDEX_ARG_COMMA VosT_Obtype, int);
	void _op_tcp_conn_v3_diag (OP_SIM_CONTEXT_ARG_OPT);
```

```
	void _op_tcp_conn_v3_terminate (OP_SIM_CONTEXT_ARG_OPT);
	void _op_tcp_conn_v3_svar (void *, const char *, void **);


	VosT_Obtype Vos_Define_Object_Prstate (const char * _op_name, unsigned int _op_size);
	VosT_Address Vos_Alloc_Object_MT (VOS_THREAD_INDEX_ARG_COMMA VosT_Obtype _op_ob_hndl);
	VosT_Fun_Status Vos_Poolmem_Dealloc_MT (VOS_THREAD_INDEX_ARG_COMMA VosT_Address _op_ob_ptr);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif




/* Process model interrupt handling procedure */


void
tcp_conn_v3 (OP_SIM_CONTEXT_ARG_OPT)
	{

#if !defined (VOSD_NO_FIN)
	int _op_block_origin = 0;
#endif
	FIN_MT (tcp_conn_v3 ());
		{
		/* Temporary Variables */
		char				str0 [256], str1 [256];
		Ici*				tmp_ici_ptr;
		int					accept_status;
		Objid				id;
		int					status;
		TcpT_Size			rem_rcv_mss;
		TcpT_Seg_Record*		rcv_rec_ptr;
		Evhandle			evh;
		Evhandle			evh_tpal;
		char				rem_addr_str [IPC_ADDR_STR_LEN];
		char				local_addr_str [IPC_ADDR_STR_LEN];
		TcpT_Seg_Fields*		tcp_seg_fd_ptr;
		double				current_time;
		int					attrib_value = 0;
		Boolean				tcp_fast_retransmit_enabled;
		Boolean				tcp_fast_recovery_enabled;
		TcpT_Ptc_Mem*		tcp_ptc_mem_ptr;
		char				msg_string [256];
		TcpT_Seg_Record*		seg_rec_ptr;
		int					i, list_size;
		/* End of Temporary Variables */


		FSM_ENTER ("tcp_conn_v3")

		FSM_BLOCK_SWITCH
			{
			/*---------------------------------------------------------*/
			/** state (init) enter executives **/
			FSM_STATE_ENTER_FORCED_NOLABEL (0, "init", "tcp_conn_v3 [init enter execs]")
				FSM_PROFILE_SECTION_IN ("tcp_conn_v3 [init enter execs]", state0_enter_exec)
				{
				/* Initialize the notification log. */
				tcp_notification_log_init ();

				/* Obtain the encompassing structure containing "tcb_ptr" & "tcp_parms_ptr".	*/
				tcp_ptc_mem_ptr = (TcpT_Ptc_Mem *) op_pro_parmem_access ();
				tcp_parameter_ptr = tcp_ptc_mem_ptr->tcp_conn_params_ptr;
```

```
/* Function which initializes all the state variables from the      */
/* tcp parameter strcuture obtained through parent to child mem.*/
tcp_conn_sv_init ();

/* Get the TCB information from the parent process. */
tcb_ptr = tcp_ptc_mem_ptr->tcb_info_ptr;
if (tcb_ptr == OPC_NIL)
	tcp_conn_error ("Unable to get TCB from TCP manager process.", OPC_NIL, OPC_NIL);

/* Initialize variables for this invocation. */
ev_ptr = tcp_ev_analyze ("INIT");

/* Create the Ici that will be used to communicate with IP.                */
ip_encap_ici_info.ip_encap_req_ici_ptr = op_ici_create ("inet_encap_req");

/* Set the source and destination address fields in the ici to the         */
/* to be pointers to the appropriate fields in the strucure. This way      */
/* to send a packet to IP, all we have to to do is store the dest and      */
/* source addresses in these locations. There is no need to call          */
/* ici_attr_set.
*/
op_ici_attr_set (ip_encap_ici_info.ip_encap_req_ici_ptr, "dest_addr", &(ip_encap_ici_info.dest_addr));
op_ici_attr_set (ip_encap_ici_info.ip_encap_req_ici_ptr, "src_addr",  &(ip_encap_ici_info.src_addr));

/* Set type of service in the ICI. If the session is opened by server      */
/* (passive connection), and server set type of service to be  "As                  */
/* Requested by Client", client's type of service is read and                       */
/* overwrites the value set here. This will be done when the server        */
/* receives a SYN message.
*/
op_ici_attr_set (ip_encap_ici_info.ip_encap_req_ici_ptr, "Type of Service", tcb_ptr->type_of_service);


/* Create a string to be used when tracing this TCP connection.            */
/* The string has been increased in length to accomodate more              */
/* tcp connections otherwise after reaching the 4 digit mark,     */
/* tcp would have started writing onto the heap.                                     */
sprintf (tcp_conn_id_str,   "tcp_conn_id_%d", tcb_ptr->conn_id);

/* Set LAN related information in the TCB.  */
my_lanhandle = tcb_ptr->lan_handle;
sess_svr_id  = tcb_ptr->lan_server_id;
local_dt_key = tcb_ptr->local_key;

/* Get the Objid of this module. */
id = op_id_self ();

/* In trace mode, print the socket information. */
if (tcp_trace_active)
	{
	inet_address_print (rem_addr_str, tcb_ptr->rem_addr);
	sprintf (str0, "remote socket: IP Address (%s), port (%d)",
		rem_addr_str, tcb_ptr->rem_port);
	sprintf (str1, "local port: (%d)", tcb_ptr->local_port);
	op_prg_odb_print_major ("Opening TCP Connection:", str0, str1, OPC_NIL);
	}

/* Set up data buffers. */
snd_buf = op_sar_buf_create  (OPC_SAR_BUF_TYPE_SEGMENT,
OPC_SAR_BUF_OPT_DEFAULT);
una_buf = op_sar_buf_create  (OPC_SAR_BUF_TYPE_RESEGMENT,
OPC_SAR_BUF_OPT_DEFAULT);
rcv_buf = op_sar_buf_create  (OPC_SAR_BUF_TYPE_REASSEMBLY,
OPC_SAR_BUF_OPT_DEFAULT);
```

```
/* Set up list for records of received out-of-order segments. */
rcv_rec_list = op_prg_list_create ();
if (rcv_rec_list == OPC_NIL)
	tcp_conn_error ("Unable to create segment reordering list.", OPC_NIL, OPC_NIL);

/* Initialize the congestion window and slow start threshold values.	*/
/* (refer page 310 in TCP/IP Illustrated Vol. I by W. Richard Stevens).	*/
/* Account for any "slow-start efficiencies" that have been enabled --	*/
/* e.g., if we are allowed to send more than one MSS as the initial		*/
/* number of segments.
*/
cwnd = initial_window_size;

/* Initialize support variables for updating retransmission timeout. */
retrans_rtt = retrans_rto/2;
current_rto = retrans_rto;
retrans_rtt_dev = 0.0;

/* Obtain the values of the FRFR attributes from the PTC memory.		*/
tcp_fast_retransmit_enabled = tcp_parameter_ptr->fast_retransmit_flag;
tcp_fast_recovery_enabled		= tcp_parameter_ptr->fast_recovery_mode;

/* Combine the result of the flags for fast retransmit & fast recovery	*/
/* so that internally they are just reprsented by one variable.		*/
if (tcp_fast_retransmit_enabled)
	{
	if (tcp_fast_recovery_enabled == TcpC_Fast_Recovery_Reno)
		{
		/* Both Fast Retransmit & Fast Recovery is enabled. This implies*/
		/* the TCP flavor in use to be "TCP Reno".
*/
		tcp_flavor = TcpC_Flavor_Reno;
		}
	else if (tcp_fast_recovery_enabled == TcpC_Fast_Recovery_New_Reno)
		{
		tcp_flavor = TcpC_Flavor_New_Reno;
		}
	else
		{
		/* Only Fast Retransmit is enabled along with slow start and	*/
		/* congestion avoidance. The TCP flavor thus would be "TCP Tahoe".*/
		tcp_flavor = TcpC_Flavor_Tahoe;
		}

	}
else
	{
	/* "TCP Reno" only operates over "TCP Tahoe". Forcibly enable		*/
	/* Fast Retransmit even if it is disabled.
*/
	if (tcp_fast_recovery_enabled)
		{
		tcp_conn_warn ("Fast Retransmit is automatically being enabled when Fast Retransmit
is enabled",
			OPC_NIL, OPC_NIL);
		tcp_reno_without_tahoe_log_write ();
		tcp_flavor = TcpC_Flavor_Reno;
		}
	else
		{
		/* Neither Fast Retransmit nor Fast Recovery is enabled.		*/
		tcp_flavor = TcpC_Flavor_Basic;
		}
	}
```

```
				/* Initialize the flags needed to maintained data				*/
				/* transmission activity after a retransmission timer		*/
				/* expires.
			*/
				dup_una_buf_init = OPC_FALSE;

				/* Registers all statistics maintained by this connection.		*/
				/* For active open sessions all the information about the		*/
				/* connection available so rename the per connection TCP		*/
				/* statistics (dimensioned) to a more comprehensible name.	*/
				/* For passive sessions that get converted to active			*/
				/* sessions, the remote address is not yet available - they		*/
				/* perform this registration when a SYN is received (exit		*/
				/* execs of "LISTEN" state.
	*/
				if (OPEN_ACTIVE)
					{
					tcp_connection_statistics_register (OPC_TRUE);
					}

				/*	The TCP process may be running in a LAN node. Initialize	*/
				/*	variables used for storing information regarding LAN level	*/
				/*	models.
				*/
				tcp_conn_info_reg		= OPC_FALSE;
				sess_wkstn_id			= LlmC_Unspec_Wkstn_Id;
				sess_svr_id				= LlmC_Unspec_Wkstn_Id;

				/* Set the maximum number of retransmission and the		*/
				/* retranmission abort time to match the values of a		*/
				/* connect message.						*/
				max_retrans_attempts = max_connect_retrans_attempts;
				max_retrans_interval = max_connect_retrans_interval;
				num_retrans_attempts = 0;
				transmission_start_time = OPC_DBL_INFINITY;
				}
				FSM_PROFILE_SECTION_OUT (state0_enter_exec)

			/** state (init) exit executives **/
			FSM_STATE_EXIT_FORCED (0, "init", "tcp_conn_v3 [init exit execs]")


			/** state (init) transition processing **/
			FSM_PROFILE_SECTION_IN ("tcp_conn_v3 [init trans conditions]", state0_trans_conds)
			FSM_INIT_COND (OPEN_PASSIVE)
			FSM_TEST_COND (OPEN_ACTIVE)
			FSM_TEST_LOGIC ("init")
			FSM_PROFILE_SECTION_OUT (state0_trans_conds)

			FSM_TRANSIT_SWITCH
				{
				FSM_CASE_TRANSIT (0, 1, state1_enter_exec, ;, "OPEN_PASSIVE", "", "init", "LISTEN",
"tcp_conn_v3 [init -> LISTEN : OPEN_PASSIVE / ]")
				FSM_CASE_TRANSIT (1, 3, state3_enter_exec, SEND_SYN;, "OPEN_ACTIVE", "SEND_SYN",
"init", "SYN_SENT", "tcp_conn_v3 [init -> SYN_SENT : OPEN_ACTIVE / SEND_SYN]")
				}
				/*---------------------------------------------------------*/



			/** state (LISTEN) enter executives **/
			FSM_STATE_ENTER_UNFORCED (1, "LISTEN", state1_enter_exec, "tcp_conn_v3 [LISTEN enter execs]")
				FSM_PROFILE_SECTION_IN ("tcp_conn_v3 [LISTEN enter execs]", state1_enter_exec)
				{
```

```
                    /* Set the name of the current state in the TCB.          */
                    strcpy (tcb_ptr->state_name, "LISTEN");

                    /* Set the flag to indicate that this end of the */
                    /* connection was set up by a PASSIVE OPEN.                          */
                    passive = 1;

                    /* Initialize the retransmission backoff factor.          */
                    retrans_backoff = 1;
                    current_rto = retrans_rto;

                    /* Set the maximum number of retransmission and the        */
                    /* retranmission abort time to match the values of a       */
                    /* connect message.                                 */
                    max_retrans_attempts = max_connect_retrans_attempts;
                    max_retrans_interval = max_connect_retrans_interval;
                    num_retrans_attempts = 0;
                    transmission_start_time = OPC_DBL_INFINITY;

                    /* Remember the initially specified values, in case      */
                    /* a connection aborts after SYN is received but         */
                    /* before entering the ESTAB state, so that we can       */
                    /* reenter LISTEN with the initial socket data.                  */
                    passive_rem_addr = inet_address_copy (tcb_ptr->rem_addr);
                    passive_rem_port = tcb_ptr->rem_port;

                    /* Indicate that the ECN status is still unset.  */
                    tcb_ptr->ecn_status = TcpC_Ecn_Not_Supported;

                    /* The connection cannot be created unless a local port was specified. */
                    if (tcb_ptr->local_port == TCPC_PORT_UNSPEC)
                              {
                              tcp_conn_warn ("An OPEN PASSIVE command was issued,",
                                        "but no local port was specified.", "Aborting connection.");
                              ev_ptr->event = TCPC_EV_ABORT;
                              }

                    if (ev_ptr->event == TCPC_EV_ABORT || ev_ptr->event == TCPC_EV_ABORT_NO_RST)
                              tcp_conn_abort ();
                    else if (ev_ptr->event == TCPC_EV_PASSIVE_RESET)
                              {
                              /* Send an indication ICI to the application layer.                    */
                              /* Application will then close the session corresponding          */
                              /* to this session, that has been aborted.                                      */
                              tcp_conn_app_indication (tcb_ptr->state_name, TCPC_IND_ABORTED);
                              }

                    /* Tag this connection process for debugging purposes.          */
                    if (op_sim_debug () == OPC_TRUE)
                              {
                              inet_address_print (local_addr_str, tcb_ptr->local_addr);
                              inet_address_print (rem_addr_str,   tcb_ptr->rem_addr);
                              sprintf (msg_string, "Passive [%s (%d) -> %s (%d); conn id %d]",
                                        local_addr_str, tcb_ptr->local_port, rem_addr_str, tcb_ptr->rem_port,  tcb_ptr-
>conn_id);

                              op_pro_tag_set (op_pro_self (), msg_string);
                              }
                    }
                    FSM_PROFILE_SECTION_OUT (state1_enter_exec)

          /** blocking after enter executives of unforced state. **/
          FSM_EXIT (3,"tcp_conn_v3")


          /** state (LISTEN) exit executives **/
```

```
				FSM_STATE_EXIT_UNFORCED (1, "LISTEN", "tcp_conn_v3 [LISTEN exit execs]")
					FSM_PROFILE_SECTION_IN ("tcp_conn_v3 [LISTEN exit execs]", state1_exit_exec)
					{
					/* Perform standard event analysis. */
					ev_ptr = tcp_ev_analyze ("LISTEN");

					switch (ev_ptr->event)
						{
						case TCPC_EV_OPEN_ACTIVE:
							break;

						case TCPC_EV_SEND:
							{
							tcp_command_send (ev_ptr->pk_ptr, ev_ptr->flags);
							break;
							}

						case TCPC_EV_RECEIVE:
							{
							tcp_command_receive (ev_ptr->num_pks);
							break;
							}

						case TCPC_EV_CLOSE:
							{
							if (tcp_trace_active)
								op_prg_odb_print_major ("TCP received command: CLOSE", OPC_NIL);
							break;
							}

						case TCPC_EV_ABORT:
							break;

						case TCPC_EV_SEG_ARRIVAL:
							{
							/*	Access the fields structure from the			*/
							/*	packet for getting infromation like ack num,*/
							/*	etc depending upon segment type.			*/
							op_pk_nfd_access (ev_ptr->pk_ptr, "fields", &tcp_seg_fd_ptr);

							if (ev_ptr->flags & TCPC_FLAG_RST)
								{
								/* Ignore RST segments. */
								if (tcp_trace_active)
									op_prg_odb_print_minor ("Received RST; ignored.", OPC_NIL);

								op_pk_destroy (ev_ptr->pk_ptr);
								}
							else if (ev_ptr->flags & TCPC_FLAG_ACK)
								{
								/* Any ACKs are invalid; respond with RST. */
								if (tcp_trace_active)
									op_prg_odb_print_minor ("Received unexpected ACK; replying with
RST.", OPC_NIL);


								/* Register/rename the statistics. */
								tcp_connection_statistics_register (OPC_FALSE);

								/* The sequence number of the RST is the ACK number of the received seg-
ment. */

								seg_ack = tcp_seg_fd_ptr->ack_num;
								tcp_seg_send (TCPC_DATA_NONE, seg_ack, TCPC_FLAG_RST);

								op_pk_destroy (ev_ptr->pk_ptr);
								}
```

```
else if (ev_ptr->flags & TCPC_FLAG_SYN)
        {
        /* Initialize receive sequence variables.  Obtain the     */
        /* sequence number of the SYN and add one to get the            */
        /* sequence number of the next expected octet.                          */
        syn_rcvd = 1;
        seg_seq = tcp_seg_fd_ptr->seq_num;
        irs = seg_seq;
        rcv_buf_seq = rcv_nxt = seg_seq + 1;

        /* In case the user has opted for the MSS to be Auto-Assigned, then */
        /* the manager would read the first ip interface MTU size and store */
        /* it as the maximum segment size.                          */
        if (snd_mss == TCPC_MSS_AUTO_ASSIGNED)
                {
                /* Scans the first configured IP interface on this node and     */
                /* returns its MTU size.                           */
                tcp_parameter_ptr->max_seg_size  = tcp_conn_mss_auto_assign
(tcb_ptr->rem_addr);

                }

        /* Check if this end supports ECN.  */
        if ((tcp_parameter_ptr->ecn_capability ==
TCPC_OPTION_STATUS_ENABLED) ||

                (tcp_parameter_ptr->ecn_capability ==
TCPC_OPTION_STATUS_PASSIVE))

                {
                /* Check if ECN-related flags are set. For the SYN-sending      */
                /* side to be ECN-capable, it should have set both ECE and      */
                /* the CWR flag in the received SYN segment.
                        */

                if ((ev_ptr->flags & TCPC_FLAG_ECE) && (ev_ptr->flags &
TCPC_FLAG_CWR))

                        {
                        /* This connection can support ECN on both sides.     */
                        tcb_ptr->ecn_status |= TcpC_Ecn_Supported;
                        }
                }

        /* If ECN was configured as passive, then send a trace message             */
        /* detailing the change to enabled or disabled.
                        */

        if ((tcp_parameter_ptr->ecn_capability ==
TCPC_OPTION_STATUS_PASSIVE) && (tcp_trace_active || tcp_extns_trace_active))
                {
                if ((ev_ptr->flags & TCPC_FLAG_ECE) && (ev_ptr->flags &
TCPC_FLAG_CWR))

                        {
                        op_prg_odb_print_minor ("Enabled ECN for the connec-
tion.",

                                "after SYN message with a supported ECN",
                                "option has been received.", OPC_NIL);
                        }
                else
                        {
                        op_prg_odb_print_minor ("Disabled ECN for the connec-
tion.",

                                "after SYN message with an unsupported ECN",
                                "option has been received.", OPC_NIL);
                        }
                }

        /* Register/rename the statistics. */
        tcp_connection_statistics_register (OPC_FALSE);
```

```
/* Let the application know that a SYN has been received.       */
/* Create the Open Indication ICI that will be sent      to the */
/* higher layer. This ICI will contain the IP address of        */
/* the remote node that is attempting to establish a            */
/* connection with this node.
*/
accept_status = tcp_conn_app_open_indicate ();

if (accept_status == TCPC_IND_CONNXN_REJECT)
  {
  /* Set the current event to be NONE and break.  */
  ev_ptr->event = TCPC_EV_NONE;

        /* Send a RESET to the other side indicating*/
        /* that the connection has been rejected.           */
  tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_RST);

        /* Break out from further processing.                   */
      break;
        }

/* Process received window scaling option, if enabled */
/* NOTE: if window_scaling_enabled is passive, then            */
/* it may end up being set as enabled, so call the              */
/* function if it is passive as well.                                   */
if ((window_scaling_enabled == TCPC_OPTION_STATUS_ENABLED) ||
        (window_scaling_enabled ==
TCPC_OPTION_STATUS_PASSIVE))

        /* This function needs to be called before the MSS     */
        /* Option is checked, because it uses the rcv_buff      */
        /* variable which may be updated in this function       */
        tcp_window_scale_option_process (ev_ptr->pk_ptr);
else
        {
        /* Configuration of window scaling needs to be done
        */

        /* here rather than in the initialization state since the   */
        /* case where it may be passive needs to be handled.              */
        /* In this case, if window scaling was configured as              */
        /* passive it has been disabled.
*/

        if (rcv_buff > TCPC_MAX_WND_SIZE)
                {
                /* Window scaling is disabled but the receive buffer speci-
fied         */

                /* greater than the value which the 16-bit field can carry.
        */

                /* Truncate the receive buffer specification, and write a
        */

                /* simulation log message.
*/

                tcp_rcvbuff_truncate_log_write (rcv_buff,
TCPC_MAX_WND_SIZE);

                rcv_buff = TCPC_MAX_WND_SIZE;
                }

        /* Initial receive window size is the full buffer size. */
        rcv_wnd = rcv_buff;

        /* The ssthresh is either set to this value based on TCP/IP Illustrated
vol2, pg 835 */

        if (window_scaling_enabled ==
TCPC_OPTION_STATUS_ENABLED)

                ssthresh = TCPC_MAX_WND_WWS_SIZE;
        else
```

```
                    ssthresh = TCPC_MAX_WND_SIZE;

                }

        /* Check for the MSS Option in the received SYN.     */
        /* Only use it if smaller than our current MSS.                    */
        if (op_pk_nfd_is_set (ev_ptr->pk_ptr, "MSS Option"))
                {
                if (op_pk_nfd_get (ev_ptr->pk_ptr, "MSS Option", &rem_rcv_mss)
== OPC_COMPCODE_FAILURE)
                        {
                        tcp_conn_warn ("Unable to get MSS option from SYN seg-
ment.", "Ignoring option.", OPC_NIL);
                        }

                /* Determine the send MSS and receive window size (the         */
                /* amount of receives data (bytes) that can be buffered             */
                /* at one time on a connection. A sending host can send           */
                /* only that amount of data before waiting for an ACK &          */
                /* window update from the receiving host. Check if the            */
                /* receive buffer is set to be computed dynamically.    */
                if (tcp_parameter_ptr->rcv_buff_size == -1)
                        {
                        /* Set the MSS to be the lower of this process' and      */
                        /* peer process' MSS configuration.
                  */

                        snd_mss = MIN (rem_rcv_mss, snd_mss);

                        /* When set to "Default", this parameter is set to at      */
                        /* least four times the negotiated MSS, with a maximum
          */

                        /* size of 64 KB.
                                                 */

                        rcv_buff = (4 * snd_mss <= TCPC_MAX_WND_SIZE) ? 4
* snd_mss : TCPC_MAX_WND_SIZE;

                        }
                else
                        {
                        /* An explicit value has been specified for the receive*/
                        /* buffer for this process.
                                        */


                        /* Set the MSS to be the lower of this process' and      */
                        /* peer process' MSS configuration. The MSS is also  */
                        /* constrained by this process' receiver buffer.           */
                        snd_mss = (rcv_buff < rem_rcv_mss) ? MIN (rcv_buff,
snd_mss) : MIN (rem_rcv_mss, snd_mss);


                        /* Once MSS has been negotiated, adjust receive */
                        /* window to multiples of mss if specified so.            */
                        tcp_conn_rcv_buff_adjust ();
                        }
                }

        /* Read the Timestamp option field and determine whether       */
        /* the new connection supports Time stamp option.                   */
        if (((conn_supports_ts == TCPC_OPTION_STATUS_ENABLED) ||
                (conn_supports_ts == TCPC_OPTION_STATUS_PASSIVE)) &&
                (op_pk_nfd_is_set (ev_ptr->pk_ptr, "Timestamp Option") ==
OPC_TRUE))

                {
                /* The fact that the field is set means that the*/
                /* time stamp option is supported by both sides.         */

                /* If conn_supports_ts was set as passive, then a          */
```

```
					/* trace message detailing the change to enabled		*/
					/* needs to be created.
				*/

					if (conn_supports_ts == TCPC_OPTION_STATUS_PASSIVE)
						{
						if (tcp_trace_active || tcp_extns_trace_active)
							{
							op_prg_odb_print_minor ("Enabled timestamp for
the connection.",

								"after SYN message with a supported time-
stamp",

								"option has been received.", OPC_NIL);
							}
						}

					/* if conn_supports_ts was set as passive, it			*/
					/* should now be set as enabled.
		*/

					conn_supports_ts = TCPC_OPTION_STATUS_ENABLED;
					}
				else
					{
					/* If conn_supports_ts was set as passive, then a		*/
					/* trace message detailing the change to disabled		*/
					/* needs to be created.
				*/

					if (conn_supports_ts == TCPC_OPTION_STATUS_PASSIVE)
						{
						if (tcp_trace_active || tcp_extns_trace_active)
							{
							op_prg_odb_print_minor ("Disabled timestamp for
the connection.",

								"after SYN message with an unsupported
timestamp",

								"option has been received.", OPC_NIL);
							}
						}

					/* The remote side does not support Timestamps. Reset		*/
					/* the local process' status for this option.			*/
					conn_supports_ts = TCPC_OPTION_STATUS_DISABLED;
					}

				/* Get the size of the remote receive window. */
				tcp_send_window_update (seg_seq, seg_ack, tcp_seg_fd_ptr->rcv_win);

				/* Look for the SACK-Permitted Option, if enabled. */
				if (((sack_enabled == TCPC_OPTION_STATUS_ENABLED) ||
					(sack_enabled == TCPC_OPTION_STATUS_PASSIVE)) &&
					(op_pk_nfd_is_set (ev_ptr->pk_ptr, "SACK-Permitted Option")))
					{
					/* If sack_enabled was set as passive, then a trace		*/
					/* message detailing the change to enabled needs to	*/
					/* be created.
					*/

					if (sack_enabled == TCPC_OPTION_STATUS_PASSIVE)
						{
						if (tcp_trace_active || tcp_extns_trace_active)
							{
							op_prg_odb_print_minor ("Enabled SACK for the
connection.",

								"after SYN message with a supported
SACK",

								"option has been received.", OPC_NIL);
							}
```

```
				}

			/* if sack_enabled was set as passive, it should		*/
			/* now be enabled.						*/

			sack_enabled = TCPC_OPTION_STATUS_ENABLED;

			/* FR/FR must be enabled for SACK to work properly */
			/* NOTE: This will have been checked only if			*/
			/* sack enabled was not set to passive.				*/

			if ((tcp_flavor != TcpC_Flavor_Reno) && (tcp_flavor != TcpC_Flavor_New_Reno))
				{
				if (tcp_trace_active || tcp_extns_trace_active)
					{
					op_prg_odb_print_minor ("Fast Retransmit/Fast Recovery is automatically",
						"being enabled for SACK.", OPC_NIL, OPC_NIL);
					}

				tcp_sack_without_frfr_log_write ();
				tcp_flavor = TcpC_Flavor_Reno;
				}

			sack_permit_rcvd = OPC_TRUE;
			}
		else
			{
			/* If sack_enabled was set as passive, then a trace	*/
			/* message detailing the change to disabled needs	*/
			/* to be created.							*/

			if (sack_enabled == TCPC_OPTION_STATUS_PASSIVE)
				{
				if (tcp_trace_active || tcp_extns_trace_active)
					{
					op_prg_odb_print_minor ("Enabled SACK for the connection.",
						"after SYN message with a supported SACK",
						"option has been received.", OPC_NIL);
					}
				}

			/* SACK will be set to disabled.				*/

			sack_enabled = TCPC_OPTION_STATUS_DISABLED;
			}

		/* Record advertized receive window statistics. */
		if ((tcb_ptr->tcp_conn_stat_ptr != OPC_NIL)&&
			(op_stat_valid (tcb_ptr->tcp_conn_stat_ptr->remote_rcv_win_stathandle) == OPC_TRUE))
			{
				op_stat_write (tcb_ptr->tcp_conn_stat_ptr->remote_rcv_win_stathandle, rcv_wnd);
			}

		/* Receord receive sequence number and end-to-end delay statistics.	*/
		tcp_seg_rcvd_stat_write ();

		/* If any data is in the segment, queue it for later processing. */
		if (op_pk_nfd_is_set (ev_ptr->pk_ptr, "data"))
```

```
									tcp_seg_receive (ev_ptr->pk_ptr, ev_ptr->flags);

								/* If a trace is active, print the new socket information. */
								if (tcp_trace_active)
									{
									inet_address_print (rem_addr_str, tcb_ptr->rem_addr);
									sprintf (str0, "SYN received; remote socket: IP Address (%s), port (%
d)",

										rem_addr_str, tcb_ptr->rem_port);
									sprintf (str1, "Initial Receive Sequence Number: (%u); Negotiated
MSS: (%u) bytes", seg_seq, snd_mss);

									op_prg_odb_print_major (str0, str1, OPC_NIL);
									}

								/* If a receive window is less than twice MSS, the application will
	*/
								/* experience high response time. This will be because a new segment	*/
								/* will be sent only when ACK timer expires. Write a log message.
	*/
								if ((snd_wnd < 2 * snd_mss) && (max_ack_delay > 0.0) && (tcb_ptr-
>local_port != TCPC_BGP_RESERVED_PORT) &&
									(tcb_ptr->rem_port	!= TCPC_BGP_RESERVED_PORT))
									{
									tcp_rcv_wnd_low_log_add (tcb_ptr, snd_wnd, snd_mss);
									}

								/* Discard the segment. */
								op_pk_destroy (ev_ptr->pk_ptr);

								/* Indicate that a SYN has been received. */
								ev_ptr->event = TCPC_EV_RCV_SYN;
								}
							else
								{
								/* Anything but RST, ACK, or SYN should be discarded. */
								if (tcp_trace_active)
									{
									tcp_conn_warn ("In LISTEN state, received a non-SYN segment.",
										"Segment will be ignored.", OPC_NIL);
									}
								op_pk_destroy (ev_ptr->pk_ptr);
								}

							break;
							}

						default:
							{
							tcp_ev_error (ev_ptr->event, "LISTEN");
							break;
							}
						}
					}
				FSM_PROFILE_SECTION_OUT (state1_exit_exec)


			/** state (LISTEN) transition processing **/
			FSM_PROFILE_SECTION_IN ("tcp_conn_v3 [LISTEN trans conditions]", state1_trans_conds)
			FSM_INIT_COND (RCV_SYN)
			FSM_TEST_COND (SEND)
			FSM_TEST_COND (CLOSE)
			FSM_TEST_COND (OPEN_ACTIVE)
			FSM_DFLT_COND
			FSM_TEST_LOGIC ("LISTEN")
			FSM_PROFILE_SECTION_OUT (state1_trans_conds)
```

```
			FSM_TRANSIT_SWITCH
				{
				FSM_CASE_TRANSIT (0, 2, state2_enter_exec, SEND_SYN_ACK;, "RCV_SYN", 
"SEND_SYN_ACK", "LISTEN", "SYN_RCVD", "tcp_conn_v3 [LISTEN -> SYN_RCVD : RCV_SYN / SEND_SYN_ACK]")
				FSM_CASE_TRANSIT (1, 3, state3_enter_exec, SEND_SYN;, "SEND", "SEND_SYN", "LISTEN", 
"SYN_SENT", "tcp_conn_v3 [LISTEN -> SYN_SENT : SEND / SEND_SYN]")
				FSM_CASE_TRANSIT (2, 11, state11_enter_exec, ;, "CLOSE", "", "LISTEN", "CLOSED", 
"tcp_conn_v3 [LISTEN -> CLOSED : CLOSE / ]")
				FSM_CASE_TRANSIT (3, 3, state3_enter_exec, SEND_SYN;, "OPEN_ACTIVE", "SEND_SYN", 
"LISTEN", "SYN_SENT", "tcp_conn_v3 [LISTEN -> SYN_SENT : OPEN_ACTIVE / SEND_SYN]")
				FSM_CASE_TRANSIT (4, 1, state1_enter_exec, ;, "default", "", "LISTEN", "LISTEN", "tcp_conn_v3 
[LISTEN -> LISTEN : default / ]")
				}
				/*---------------------------------------------------------*/



			/** state (SYN_RCVD) enter executives **/
			FSM_STATE_ENTER_UNFORCED (2, "SYN_RCVD", state2_enter_exec, "tcp_conn_v3 [SYN_RCVD enter 
execs]")
				FSM_PROFILE_SECTION_IN ("tcp_conn_v3 [SYN_RCVD enter execs]", state2_enter_exec)
				{
				strcpy (tcb_ptr->state_name, "SYN_RCVD");

				/*	Register TCP connection information in			*/
				/*	lan handle if this is a LAN node.			*/
				if (my_lanhandle != OPC_NIL)
					{
					/*	Check if tcp connection information			*/
					/*	(wkstn id, remote port and local port)		*/
					/*	is already registered in the llm			*/
					/*	transport connection table.				*/
					if (tcp_conn_info_reg == OPC_FALSE)
						{
						/*	This connection is opened on the	*/
						/*	server.							*/
						sess_wkstn_id = sess_svr_id;

						/*	Register information about the			*/
						/*	new TCP connection process.				*/
						if (llm_trans_new_session (my_lanhandle, &sess_wkstn_id, tcb_ptr->rem_port, tcb_ptr-
>local_port) == OPC_COMPCODE_FAILURE)
							{
							tcp_conn_warn ("Unable to create tcp connection record in the tcp connection 
table", OPC_NIL, OPC_NIL);
							}

						/*	Set the flag that tcp connection	*/
						/*	information is registered.			*/
						tcp_conn_info_reg = OPC_TRUE;
						}
					}

				if (ev_ptr->event == TCPC_EV_ABORT)
					{
					tcp_seg_send (TCPC_DATA_NONE, snd_nxt, TCPC_FLAG_RST);
					tcp_conn_abort ();
					}
				else if (ev_ptr->event == TCPC_EV_ABORT_NO_RST)
					tcp_conn_abort ();
				}
				FSM_PROFILE_SECTION_OUT (state2_enter_exec)

			/** blocking after enter executives of unforced state. **/
```

```
FSM_EXIT (5,"tcp_conn_v3")


/** state (SYN_RCVD) exit executives **/
FSM_STATE_EXIT_UNFORCED (2, "SYN_RCVD", "tcp_conn_v3 [SYN_RCVD exit execs]")
	FSM_PROFILE_SECTION_IN ("tcp_conn_v3 [SYN_RCVD exit execs]", state2_exit_exec)
	{
	ev_ptr = tcp_ev_analyze ("SYN_RCVD");

	switch (ev_ptr->event)
		{
		case TCPC_EV_SEND:
			{
			tcp_command_send (ev_ptr->pk_ptr, ev_ptr->flags);
			break;
			}

		case TCPC_EV_RECEIVE:
			{
			tcp_command_receive (ev_ptr->num_pks);
			break;
			}

		case TCPC_EV_CLOSE:
			{
			if (tcp_trace_active)
				op_prg_odb_print_major ("TCP received command: CLOSE", OPC_NIL);
			break;
			}

		case TCPC_EV_ABORT:
			break;

		case TCPC_EV_SEG_ARRIVAL:
			{
			/* Check sequence number of received segment. */
			if (tcp_seq_check ())
				{
				if (ev_ptr->flags & TCPC_FLAG_RST)
					{
					if (passive)
						{
						/* Abandon this connection, return to LISTEN state. */
						if (tcp_trace_active)
							op_prg_odb_print_minor ("RST received; returning
to LISTEN state.", OPC_NIL);

						ev_ptr->event = TCPC_EV_PASSIVE_RESET;

						/* Destroy possible prevoious allocation to rem_addr.*/
						inet_address_destroy (tcb_ptr->rem_addr);

						/* Restore the initially specified socket information. */
						tcb_ptr->rem_addr = inet_address_copy (passive_rem_addr);
						tcb_ptr->rem_port = passive_rem_port;


						/* Reinitialize the keys.                    */
						oms_dt_item_remove (tcb_ptr->dt_handle, tcb_ptr-
>local_key);

						tcb_ptr->local_key  = oms_dt_item_insert (tcb_ptr-
>dt_handle, tcb_ptr);

						local_dt_key = tcb_ptr->local_key;
						tcb_ptr->remote_key = OmsC_Dt_Key_Undefined;

						/* Clean out all data buffers. */
```

```
op_sar_buf_destroy (snd_buf);
op_sar_buf_destroy (una_buf);
op_sar_buf_destroy (rcv_buf);
snd_buf = op_sar_buf_create
(OPC_SAR_BUF_TYPE_SEGMENT, OPC_SAR_BUF_OPT_DEFAULT);
una_buf = op_sar_buf_create
(OPC_SAR_BUF_TYPE_RESEGMENT, OPC_SAR_BUF_OPT_DEFAULT);
rcv_buf = op_sar_buf_create
(OPC_SAR_BUF_TYPE_REASSEMBLY, OPC_SAR_BUF_OPT_DEFAULT);

/* Flush all information about received packets. */

op_prg_list_free (rcv_rec_list);

/* Cancel all pending timers. */
if (op_ev_valid (ack_evh) && op_ev_pending (ack_evh))
	if (op_ev_cancel (ack_evh) ==
OPC_COMPCODE_FAILURE)
		tcp_conn_warn ("Unable to cancel pending
acknowledgement timeout.",
			OPC_NIL, OPC_NIL);
if (op_ev_valid (retrans_evh) && op_ev_pending
(retrans_evh))
	if (op_ev_cancel (retrans_evh) ==
OPC_COMPCODE_FAILURE)
		tcp_conn_warn ("Unable to cancel pending
retransmission timeout.",
			OPC_NIL, OPC_NIL);

if (op_ev_valid (max_retrans_evh) && op_ev_pending
(max_retrans_evh))
{
if (op_ev_cancel (max_retrans_evh) == OPC_COMPCODE_FAILURE)
{
tcp_conn_warn ("Unable to cancel old retransmission timeout.",
"Spurious resets might be sent out.", OPC_NIL);
}
}

/* Reinitialize FR/FR setting                 */
tcp_conn_frfr_reset ();

/* Reinitialize state variables.        */
tcp_conn_sv_init ();

/* Set the name of the current state in the TCB.         */
strcpy (tcb_ptr->state_name, "LISTEN");

/* Reset the application stream index. It will be set once
        */
/* the application sends OPEN request.
                                  */
tcb_ptr->strm_index = TCPC_UNINIT_STREAM;

/* Initial receive window size is the full buffer size. */
rcv_wnd = rcv_buff;

/* The ssthresh is either set to this value based on TCP/IP
Illustrated vol2, pg 835 */
/* If window_scaling_enabled is still passive then we should
still assume that it            */
/* will eventually be enabled.  The case when passive gets
set to disabled will be        */
/* handled later.
                                   */
```

```
if ((window_scaling_enabled ==
TCPC_OPTION_STATUS_ENABLED) ||
        (window_scaling_enabled ==
TCPC_OPTION_STATUS_PASSIVE))
        ssthresh = TCPC_MAX_WND_WWS_SIZE;
else
        ssthresh = TCPC_MAX_WND_SIZE;

/* Initialize the congestion window and slow start threshold
values.   */
/* (refer page 310 in TCP/IP Illustrated Vol. I by W. Richard
Stevens).            */
/* Account for any "slow-start efficiencies" that have been
enabled --           */
/* e.g., if we are allowed to send more than one MSS as the
initial              */
/* number of segments.
*/
cwnd = initial_window_size;

/* Initialize support variables for updating retransmission
timeout. */
retrans_rtt = retrans_rto/2;
current_rto = retrans_rto;
retrans_rtt_dev = 0.0;

/* Initialize the flags needed to maintained data
        */
/* transmission activity after a retransmission timer
    */
/* expires.
*/
dup_una_buf_init = OPC_FALSE;

num_retrans_attempts       = 0;
transmission_start_time = OPC_DBL_INFINITY;
}
else
        ev_ptr->event = TCPC_EV_ABORT;

op_pk_destroy (ev_ptr->pk_ptr);
break;
}

if (tcp_syn_check ())
        {
        /* If this is an ACK of the SYN sent by this process, establish connec-
tion fully. */
        if (ev_ptr->flags & TCPC_FLAG_ACK)
                {


/* End of Temporary Variables */

/* Diagnostic Block */


BINIT
{
printf ("Connection State: %s\n", tcb_ptr->state_name);
printf ("Type of Service:  %s\n", ip_qos_tos_value_to_tos_name_convert ((OmsT_Qm_Tos) tcb_ptr->type_of_service));

inet_address_print (rem_addr_str, tcb_ptr->rem_addr);
inet_address_print (local_addr_str, tcb_ptr->local_addr);
printf ("\nLocal Port: (%4d)\tRemote Port: (%4d)\n", tcb_ptr->local_port, tcb_ptr->rem_port);
```

```
printf ("Remote Address: (%s)  Advertised Local IP Address: (%s)\n", rem_addr_str, local_addr_str);
printf ("Local ECN Status: (%s) Connection ECN Status: (%s)\n",
        tcp_parameter_ptr->ecn_capability == OPC_TRUE ? "Enabled" : "Disabled",
        tcb_ptr->ecn_status & TcpC_Ecn_Supported ? "Enabled" : "Disabled");
printf ("Time Stamp Option Status: (%s)\n",
        tcp_parameter_ptr->timestamp_flag == OPC_TRUE ? "Enabled" : "Disabled");

printf ("\nSend Sequence Variables          Receive Sequence Variables\n");
printf ("-----------------------          --------------------------\n");
printf (" SND.UNA: %12u\n", snd_una);
printf (" SND.MAX: %12u\n", snd_max);
printf (" SND.NXT: %12u             RCV.NXT: %12u\n", snd_nxt, rcv_nxt);
printf (" SND.WND: %12u             RCV.WND: %12u\n", snd_wnd, rcv_wnd);
printf (" SND.WL1: %12u\n", snd_wl1);
printf (" SND.WL2: %12u\n", snd_wl2);
printf ("     ISS: %12u                 IRS: %12u\n", iss, irs);

printf ("\nCongestion Control Variables\n");
printf ("----------------------------\n");
printf ("       cwnd: %12u\n", cwnd);
printf ("   ssthresh: %12u\n", ssthresh);

printf ("\nRetransmission Timeout Variables\n");
printf ("--------------------------------\n");
printf ("          RTO: %7.3f sec\n", current_rto);
printf (" smoothed RTT: %7.3f sec\n", retrans_rtt);
printf (" mean RTT dev: %7.3f sec\n", retrans_rtt_dev);
printf ("rtt base time: %7.3f sec\n", rtt_base_time);

printf ("\nReceive requests pending: %d\n", num_pks_req);
printf ("Sequence number of next packet to be forwarded: %u\n", rcv_buf_seq);

if (snd_up_valid)
        printf ("Last octet of outgoing URGENT data: %u\n", snd_up);
if (rcv_up_valid)
        printf ("Last octet of incoming URGENT data: %u\n", rcv_up);

/* Print information on TCP buffers. */
if (op_prg_odb_ltrace_active ("tcp_buffers"))
        {
        /* Print the complete buffer dumps only if the user explicitly asks. */
        printf ("\nUnsent data buffer:");
        op_sar_buf_print (snd_buf);
        printf ("\nUnacknowledged data buffer (retransmission buffer):");
        op_sar_buf_print (una_buf);
        printf ("\nReassembly/Socket buffer:");
        op_sar_buf_print (rcv_buf);
        }
else
        {
        printf ("\nUnsent data buffer contains "OPC_SAR_SIZE_FMT" bytes.\n", (OpT_Sar_Size)op_sar_buf_size
(snd_buf) / 8);
        printf ("Retransmission buffer contains "OPC_SAR_SIZE_FMT" bytes.\n", (OpT_Sar_Size)op_sar_buf_size
(una_buf) / 8);
        printf ("Reassembly/Socket buffer contains "OPC_SAR_SIZE_FMT" bytes (%d complete packets).\n",
                        (OpT_Sar_Size)op_sar_buf_size (rcv_buf) / 8, op_sar_rsmbuf_pk_count
(rcv_buf));
        }

/* Printf information on out-of-order segment list.       */
list_size = op_prg_list_size (rcv_rec_list);
printf ("\nOut-of-order receive list contains %d packets.\n", list_size);

if (list_size > 0 && op_prg_odb_ltrace_active ("tcp_ooo_list"))
        {
```

```
				for (i = 0; i < list_size; i++)
					{
					/* Access the i_th_segment.		*/
					seg_rec_ptr = (TcpT_Seg_Record *) op_prg_list_access (rcv_rec_list, i);
					if (seg_rec_ptr == OPC_NIL)
						{
						tcp_conn_warn ("Unable to get segment record from reordering list.",
										"Skipping to next record in list.", OPC_NIL);
						continue;
						}
					else
						{
						printf ("\t%2d. Sequence Number: %u\t size: "OPC_PACKET_SIZE_FMT" bytes\n",
							i+1, seg_rec_ptr->seq, (OpT_Packet_Size) (op_pk_total_size_get (seg_rec_ptr-
>data_ptr) / 8.0));
						}
					}
				}
			if (sack_enabled == OPC_TRUE)
				{
				printf (" pipe: %12u\n", pipe);

				tcp_scoreboard_print ();
				}

			}

			/* End of Diagnostic Block */

			}

		FOUT
#endif /* OPD_ALLOW_ODB */
		}




void
_op_tcp_conn_v3_terminate (OP_SIM_CONTEXT_ARG_OPT)
		{

#if !defined (VOSD_NO_FIN)
		int _op_block_origin = __LINE__;
#endif

		FIN_MT (_op_tcp_conn_v3_terminate ())

		if (1)
			{
			/* Temporary Variables */
			char					str0 [256], str1 [256];
			Ici*					tmp_ici_ptr;
			int						accept_status;
			Objid					id;
			int						status;
			TcpT_Size				rem_rcv_mss;
			TcpT_Seg_Record*		rcv_rec_ptr;
			Evhandle				evh;
			Evhandle				evh_tpal;
			char					rem_addr_str [IPC_ADDR_STR_LEN];
			char					local_addr_str [IPC_ADDR_STR_LEN];
			TcpT_Seg_Fields*		tcp_seg_fd_ptr;
			double					current_time;
			int						attrib_value = 0;
```

```
Boolean                                 tcp_fast_retransmit_enabled;
Boolean                                 tcp_fast_recovery_enabled;
TcpT_Ptc_Mem*           tcp_ptc_mem_ptr;
char                                    msg_string [256];
TcpT_Seg_Record*        seg_rec_ptr;
int                                             i, list_size;
/* End of Temporary Variables */

/* Termination Block */


BINIT
{
/* Generate diagnostic message if tracing is on. */
if (tcp_trace_active)
        {
        op_prg_odb_print_major ("Connection closed.", OPC_NIL);
        }

/* Free the "key" index used by this process. This       */
/* index is maintained for fast matching of TCP                  */
/* connections when a packet comes to the manager.   */
oms_dt_item_remove (tcb_ptr->dt_handle, local_dt_key);

/* Destroy the net_ici_ptr.                                          */
op_ici_destroy (ip_encap_ici_info.ip_encap_req_ici_ptr);

if (ev_ptr->event == TCPC_EV_ABORT || ev_ptr->event == TCPC_EV_ABORT_NO_RST)
        status = TCPC_IND_ABORTED;
else
        status = TCPC_IND_CLOSED;

/* Alert the TCP manager of the status change. */
sprintf (tcb_ptr->state_name, "CLOSED");
tmp_ici_ptr = op_ici_create ("tcp_status_ind");
if (tmp_ici_ptr == OPC_NIL ||
        op_ici_attr_set (tmp_ici_ptr, "conn_id", tcb_ptr->conn_id) == OPC_COMPCODE_FAILURE ||
        op_ici_attr_set (tmp_ici_ptr, "status", status) == OPC_COMPCODE_FAILURE)
        {
        tcp_conn_warn ("Unable to create or initialize status indication ICI.",
                "TCP manager process will not be notified that this connection has closed.",
                OPC_NIL);
        }
else
        {
        op_ici_install (tmp_ici_ptr);

        /* Send a status Indication message to the higher layer and tcp_manager          */
        /* to clean up the record for the current TCP connection.                                */
        evh = op_intrpt_schedule_remote (op_sim_time (), TCPC_COMMAND_STATUS_IND, op_id_self ());

        if (op_ev_valid (evh) == OPC_FALSE)
                {
                tcp_conn_warn ("Unable to schedule remote interrupt to TCP manager.",
                        "TCP manager process will not be notified that this connection has closed.",
                        OPC_NIL);
                }
        }

/* Inform the application that this session has been closed.       */
tcp_conn_app_notify_conn_close (tcb_ptr->state_name, status);

/* Free all data buffers. */
op_sar_buf_destroy (snd_buf);
op_sar_buf_destroy (una_buf);
```

```
			if (dup_una_buf_init == OPC_TRUE)
				op_sar_buf_destroy (dup_una_buf);
			op_sar_buf_destroy (rcv_buf);

			/* Free the out-of-order segment records. */
			while (op_prg_list_size (rcv_rec_list) > 0)
				{
				rcv_rec_ptr = (TcpT_Seg_Record*) op_prg_list_remove (rcv_rec_list, OPC_LISTPOS_HEAD);
				if (rcv_rec_ptr != OPC_NIL)
					{
					if (rcv_rec_ptr->data_ptr != OPC_NIL)
						op_pk_destroy (rcv_rec_ptr->data_ptr);
					op_prg_mem_free (rcv_rec_ptr);
					}
				}
			op_prg_mem_free (rcv_rec_list);

			/* Free the SACK-related memory (scoreboard, sacklist). */
			tcp_sack_memory_free ();

			/* Clear all timeouts. */
			tcp_timeout_ev_cancel ();
			}

			/* End of Termination Block */

			}
		Vos_Poolmem_Dealloc_MT (OP_SIM_CONTEXT_THREAD_INDEX_COMMA pr_state_ptr);

		FOUT
		}


/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in _op_tcp_conn_v3_svar function. */
#undef tcb_ptr
#undef tcp_parameter_ptr
#undef ev_ptr
#undef conn_start_time
#undef ack_evh
#undef retrans_evh
#undef persist_evh
#undef time_wait_evh
#undef snd_buf
#undef una_buf
#undef rcv_buf
#undef rcv_rec_list
#undef snd_mss
#undef initial_window_size
#undef rcv_buf_usage_thresh
#undef sess_svr_id
#undef tcp_conn_info_reg
#undef my_lanhandle
#undef sess_wkstn_id
#undef retrans_rtt
#undef retrans_rtt_dev
#undef rtt_gain
#undef rtt_dev_gain
#undef rtt_dev_coef
#undef rtt_active
#undef rtt_base_time
#undef rtt_seq
#undef rto_min
#undef rto_max
```

```
#undef retrans_rto
#undef retrans_backoff
#undef current_rto
#undef snd_una
#undef snd_max
#undef snd_nxt
#undef snd_wnd
#undef snd_up
#undef snd_wl1
#undef snd_wl2
#undef iss
#undef snd_fin_seq
#undef snd_fin_valid
#undef snd_up_valid
#undef rcv_nxt
#undef rcv_up
#undef irs
#undef rcv_fin_seq
#undef rcv_fin_valid
#undef rcv_up_valid
#undef rcv_buf_seq
#undef seg_seq
#undef seg_ack
#undef seg_len
#undef rcv_buff
#undef rcv_wnd
#undef cwnd
#undef ssthresh
#undef tcp_trace_active
#undef tcp_retransmission_trace_active
#undef tcp_extns_trace_active
#undef nagle_support
#undef max_ack_delay
#undef timer_gran
#undef persist_timeout
#undef karns_algo_enabled
#undef max_retrans_seq
#undef syn_rcvd
#undef conn_estab
#undef num_pks_req
#undef last_snd_time
#undef dup_una_buf_init
#undef dup_una_buf
#undef passive
#undef passive_rem_addr
#undef passive_rem_port
#undef tcp_app_notified_for_conn_closed
#undef packet_thru_handle
#undef byte_thru_handle
#undef packet_sec_thru_handle
#undef byte_sec_thru_handle
#undef tcp_seg_delay_handle
#undef tcp_seg_global_delay_handle
#undef tcp_delay_handle
#undef tcp_global_delay_handle
#undef tcp_global_retrans_count_handle
#undef retrans_count_handle
#undef tcp_del_ack_scheme
#undef tcp_segments_rcvd_without_sending_ack
#undef wnd_scale_sent
#undef wnd_scale_rcvd
#undef snd_scale
#undef rcv_scale
#undef requested_snd_scale
#undef requested_rcv_scale
```

```
#undef window_scaling_enabled
#undef tcp_flavor
#undef dup_ack_cnt
#undef sack_enabled
#undef sack_permit_rcvd
#undef sack_permit_sent
#undef scoreboard_ptr
#undef sacklist_ptr
#undef pipe
#undef fast_retransmit_occurring
#undef scoreboard_entry_pmh
#undef sackblock_pmh
#undef tcp_scoreboard_and_sacklist_ptr_valid
#undef tcp_conn_id_str
#undef max_retrans_mode
#undef max_connect_retrans_attempts
#undef max_connect_retrans_interval
#undef max_data_retrans_attempts
#undef max_data_retrans_interval
#undef max_retrans_attempts
#undef max_retrans_interval
#undef max_retrans_evh
#undef num_retrans_attempts
#undef transmission_start_time
#undef close_indicated_to_app
#undef local_dt_key
#undef fin_segment_sent
#undef push_seq
#undef nagle_limit_time
#undef snd_wnd_limit_time
#undef cwnd_limit_time
#undef tcp_app_notified_for_close_rcvd
#undef snd_recover
#undef timestamp_info
#undef conn_supports_ts
#undef max_recvd_wnd
#undef ip_encap_ici_info

#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

VosT_Obtype
_op_tcp_conn_v3_init (int * init_block_ptr)
	{

#if !defined (VOSD_NO_FIN)
	int _op_block_origin = 0;
#endif
	VosT_Obtype obtype = OPC_NIL;
	FIN_MT (_op_tcp_conn_v3_init (init_block_ptr))

	obtype = Vos_Define_Object_Prstate ("proc state vars (tcp_conn_v3)",
		sizeof (tcp_conn_v3_state));
	*init_block_ptr = 0;

	FRET (obtype)
	}

VosT_Address
_op_tcp_conn_v3_alloc (VOS_THREAD_INDEX_ARG_COMMA VosT_Obtype obtype, int init_block)
	{
```

```
#if !defined (VOSD_NO_FIN)
	int _op_block_origin = 0;
#endif
	tcp_conn_v3_state * ptr;
	FIN_MT (_op_tcp_conn_v3_alloc (obtype))

	ptr = (tcp_conn_v3_state *)Vos_Alloc_Object_MT (VOS_THREAD_INDEX_COMMA obtype);
	if (ptr != OPC_NIL)
		{
		ptr->_op_current_block = init_block;
#if defined (OPD_ALLOW_ODB)
		ptr->_op_current_state = "tcp_conn_v3 [init enter execs]";
#endif
		}
	FRET ((VosT_Address)ptr)
	}



void
_op_tcp_conn_v3_svar (void * gen_ptr, const char * var_name, void ** var_p_ptr)
	{
	tcp_conn_v3_state		*prs_ptr;

	FIN_MT (_op_tcp_conn_v3_svar (gen_ptr, var_name, var_p_ptr))

	if (var_name == OPC_NIL)
		{
		*var_p_ptr = (void *)OPC_NIL;
		FOUT
		}
	prs_ptr = (tcp_conn_v3_state *)gen_ptr;

	if (strcmp ("tcb_ptr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcb_ptr);
		FOUT
		}
	if (strcmp ("tcp_parameter_ptr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_parameter_ptr);
		FOUT
		}
	if (strcmp ("ev_ptr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->ev_ptr);
		FOUT
		}
	if (strcmp ("conn_start_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->conn_start_time);
		FOUT
		}
	if (strcmp ("ack_evh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->ack_evh);
		FOUT
		}
	if (strcmp ("retrans_evh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->retrans_evh);
		FOUT
		}
	if (strcmp ("persist_evh" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->persist_evh);
		FOUT
		}
	if (strcmp ("time_wait_evh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->time_wait_evh);
		FOUT
		}
	if (strcmp ("snd_buf" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_buf);
		FOUT
		}
	if (strcmp ("una_buf" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->una_buf);
		FOUT
		}
	if (strcmp ("rcv_buf" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_buf);
		FOUT
		}
	if (strcmp ("rcv_rec_list" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_rec_list);
		FOUT
		}
	if (strcmp ("snd_mss" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_mss);
		FOUT
		}
	if (strcmp ("initial_window_size" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->initial_window_size);
		FOUT
		}
	if (strcmp ("rcv_buf_usage_thresh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_buf_usage_thresh);
		FOUT
		}
	if (strcmp ("sess_svr_id" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->sess_svr_id);
		FOUT
		}
	if (strcmp ("tcp_conn_info_reg" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_conn_info_reg);
		FOUT
		}
	if (strcmp ("my_lanhandle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->my_lanhandle);
		FOUT
		}
	if (strcmp ("sess_wkstn_id" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->sess_wkstn_id);
		FOUT
		}
	if (strcmp ("retrans_rtt" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->retrans_rtt);
		FOUT
		}
	if (strcmp ("retrans_rtt_dev" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->retrans_rtt_dev);
		FOUT
		}
	if (strcmp ("rtt_gain" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rtt_gain);
		FOUT
		}
	if (strcmp ("rtt_dev_gain" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rtt_dev_gain);
		FOUT
		}
	if (strcmp ("rtt_dev_coef" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rtt_dev_coef);
		FOUT
		}
	if (strcmp ("rtt_active" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rtt_active);
		FOUT
		}
	if (strcmp ("rtt_base_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rtt_base_time);
		FOUT
		}
	if (strcmp ("rtt_seq" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rtt_seq);
		FOUT
		}
	if (strcmp ("rto_min" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rto_min);
		FOUT
		}
	if (strcmp ("rto_max" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rto_max);
		FOUT
		}
	if (strcmp ("retrans_rto" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->retrans_rto);
		FOUT
		}
	if (strcmp ("retrans_backoff" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->retrans_backoff);
		FOUT
		}
	if (strcmp ("current_rto" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->current_rto);
		FOUT
		}
	if (strcmp ("snd_una" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->snd_una);
		FOUT
		}
	if (strcmp ("snd_max" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_max);
		FOUT
		}
	if (strcmp ("snd_nxt" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_nxt);
		FOUT
		}
	if (strcmp ("snd_wnd" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_wnd);
		FOUT
		}
	if (strcmp ("snd_up" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_up);
		FOUT
		}
	if (strcmp ("snd_wl1" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_wl1);
		FOUT
		}
	if (strcmp ("snd_wl2" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_wl2);
		FOUT
		}
	if (strcmp ("iss" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->iss);
		FOUT
		}
	if (strcmp ("snd_fin_seq" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_fin_seq);
		FOUT
		}
	if (strcmp ("snd_fin_valid" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_fin_valid);
		FOUT
		}
	if (strcmp ("snd_up_valid" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_up_valid);
		FOUT
		}
	if (strcmp ("rcv_nxt" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_nxt);
		FOUT
		}
	if (strcmp ("rcv_up" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_up);
		FOUT
		}
	if (strcmp ("irs" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->irs);
		FOUT
		}
	if (strcmp ("rcv_fin_seq" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_fin_seq);
		FOUT
		}
	if (strcmp ("rcv_fin_valid" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_fin_valid);
		FOUT
		}
	if (strcmp ("rcv_up_valid" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_up_valid);
		FOUT
		}
	if (strcmp ("rcv_buf_seq" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_buf_seq);
		FOUT
		}
	if (strcmp ("seg_seq" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->seg_seq);
		FOUT
		}
	if (strcmp ("seg_ack" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->seg_ack);
		FOUT
		}
	if (strcmp ("seg_len" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->seg_len);
		FOUT
		}
	if (strcmp ("rcv_buff" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_buff);
		FOUT
		}
	if (strcmp ("rcv_wnd" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_wnd);
		FOUT
		}
	if (strcmp ("cwnd" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->cwnd);
		FOUT
		}
	if (strcmp ("ssthresh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->ssthresh);
		FOUT
		}
	if (strcmp ("tcp_trace_active" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_trace_active);
		FOUT
		}
	if (strcmp ("tcp_retransmission_trace_active" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->tcp_retransmission_trace_active);
		FOUT
		}
	if (strcmp ("tcp_extns_trace_active" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_extns_trace_active);
		FOUT
		}
	if (strcmp ("nagle_support" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->nagle_support);
		FOUT
		}
	if (strcmp ("max_ack_delay" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_ack_delay);
		FOUT
		}
	if (strcmp ("timer_gran" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->timer_gran);
		FOUT
		}
	if (strcmp ("persist_timeout" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->persist_timeout);
		FOUT
		}
	if (strcmp ("karns_algo_enabled" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->karns_algo_enabled);
		FOUT
		}
	if (strcmp ("max_retrans_seq" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_retrans_seq);
		FOUT
		}
	if (strcmp ("syn_rcvd" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->syn_rcvd);
		FOUT
		}
	if (strcmp ("conn_estab" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->conn_estab);
		FOUT
		}
	if (strcmp ("num_pks_req" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->num_pks_req);
		FOUT
		}
	if (strcmp ("last_snd_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->last_snd_time);
		FOUT
		}
	if (strcmp ("dup_una_buf_init" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->dup_una_buf_init);
		FOUT
		}
	if (strcmp ("dup_una_buf" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->dup_una_buf);
		FOUT
		}
	if (strcmp ("passive" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->passive);
		FOUT
		}
	if (strcmp ("passive_rem_addr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->passive_rem_addr);
		FOUT
		}
	if (strcmp ("passive_rem_port" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->passive_rem_port);
		FOUT
		}
	if (strcmp ("tcp_app_notified_for_conn_closed" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_app_notified_for_conn_closed);
		FOUT
		}
	if (strcmp ("packet_thru_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->packet_thru_handle);
		FOUT
		}
	if (strcmp ("byte_thru_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->byte_thru_handle);
		FOUT
		}
	if (strcmp ("packet_sec_thru_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->packet_sec_thru_handle);
		FOUT
		}
	if (strcmp ("byte_sec_thru_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->byte_sec_thru_handle);
		FOUT
		}
	if (strcmp ("tcp_seg_delay_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_seg_delay_handle);
		FOUT
		}
	if (strcmp ("tcp_seg_global_delay_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_seg_global_delay_handle);
		FOUT
		}
	if (strcmp ("tcp_delay_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_delay_handle);
		FOUT
		}
	if (strcmp ("tcp_global_delay_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_global_delay_handle);
		FOUT
		}
	if (strcmp ("tcp_global_retrans_count_handle" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->tcp_global_retrans_count_handle);
		FOUT
		}
	if (strcmp ("retrans_count_handle" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->retrans_count_handle);
		FOUT
		}
	if (strcmp ("tcp_del_ack_scheme" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_del_ack_scheme);
		FOUT
		}
	if (strcmp ("tcp_segments_rcvd_without_sending_ack" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_segments_rcvd_without_sending_ack);
		FOUT
		}
	if (strcmp ("wnd_scale_sent" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->wnd_scale_sent);
		FOUT
		}
	if (strcmp ("wnd_scale_rcvd" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->wnd_scale_rcvd);
		FOUT
		}
	if (strcmp ("snd_scale" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_scale);
		FOUT
		}
	if (strcmp ("rcv_scale" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->rcv_scale);
		FOUT
		}
	if (strcmp ("requested_snd_scale" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->requested_snd_scale);
		FOUT
		}
	if (strcmp ("requested_rcv_scale" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->requested_rcv_scale);
		FOUT
		}
	if (strcmp ("window_scaling_enabled" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->window_scaling_enabled);
		FOUT
		}
	if (strcmp ("tcp_flavor" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_flavor);
		FOUT
		}
	if (strcmp ("dup_ack_cnt" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->dup_ack_cnt);
		FOUT
		}
	if (strcmp ("sack_enabled" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->sack_enabled);
		FOUT
		}
	if (strcmp ("sack_permit_rcvd" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->sack_permit_rcvd);
		FOUT
		}
	if (strcmp ("sack_permit_sent" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->sack_permit_sent);
		FOUT
		}
	if (strcmp ("scoreboard_ptr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->scoreboard_ptr);
		FOUT
		}
	if (strcmp ("sacklist_ptr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->sacklist_ptr);
		FOUT
		}
	if (strcmp ("pipe" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->pipe);
		FOUT
		}
	if (strcmp ("fast_retransmit_occurring" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->fast_retransmit_occurring);
		FOUT
		}
	if (strcmp ("scoreboard_entry_pmh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->scoreboard_entry_pmh);
		FOUT
		}
	if (strcmp ("sackblock_pmh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->sackblock_pmh);
		FOUT
		}
	if (strcmp ("tcp_scoreboard_and_sacklist_ptr_valid" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_scoreboard_and_sacklist_ptr_valid);
		FOUT
		}
	if (strcmp ("tcp_conn_id_str" , var_name) == 0)
		{
		*var_p_ptr = (void *) (prs_ptr->tcp_conn_id_str);
		FOUT
		}
	if (strcmp ("max_retrans_mode" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_retrans_mode);
		FOUT
		}
	if (strcmp ("max_connect_retrans_attempts" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_connect_retrans_attempts);
		FOUT
		}
	if (strcmp ("max_connect_retrans_interval" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->max_connect_retrans_interval);
		FOUT
		}
	if (strcmp ("max_data_retrans_attempts" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_data_retrans_attempts);
		FOUT
		}
	if (strcmp ("max_data_retrans_interval" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_data_retrans_interval);
		FOUT
		}
	if (strcmp ("max_retrans_attempts" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_retrans_attempts);
		FOUT
		}
	if (strcmp ("max_retrans_interval" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_retrans_interval);
		FOUT
		}
	if (strcmp ("max_retrans_evh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_retrans_evh);
		FOUT
		}
	if (strcmp ("num_retrans_attempts" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->num_retrans_attempts);
		FOUT
		}
	if (strcmp ("transmission_start_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->transmission_start_time);
		FOUT
		}
	if (strcmp ("close_indicated_to_app" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->close_indicated_to_app);
		FOUT
		}
	if (strcmp ("local_dt_key" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->local_dt_key);
		FOUT
		}
	if (strcmp ("fin_segment_sent" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->fin_segment_sent);
		FOUT
		}
	if (strcmp ("push_seq" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->push_seq);
		FOUT
		}
	if (strcmp ("nagle_limit_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->nagle_limit_time);
		FOUT
		}
	if (strcmp ("snd_wnd_limit_time" , var_name) == 0)
		{
```

```
		*var_p_ptr = (void *) (&prs_ptr->snd_wnd_limit_time);
		FOUT
		}
	if (strcmp ("cwnd_limit_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->cwnd_limit_time);
		FOUT
		}
	if (strcmp ("tcp_app_notified_for_close_rcvd" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->tcp_app_notified_for_close_rcvd);
		FOUT
		}
	if (strcmp ("snd_recover" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->snd_recover);
		FOUT
		}
	if (strcmp ("timestamp_info" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->timestamp_info);
		FOUT
		}
	if (strcmp ("conn_supports_ts" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->conn_supports_ts);
		FOUT
		}
	if (strcmp ("max_recvd_wnd" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->max_recvd_wnd);
		FOUT
		}
	if (strcmp ("ip_encap_ici_info" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->ip_encap_ici_info);
		FOUT
		}
	*var_p_ptr = (void *)OPC_NIL;

	FOUT
	}
```