Improving TCP Performance with Periodic Disconnections over Wireless Links

M-TCP Performance Evaluation Using OPNET

Meihua Judy Zhan (mjzhan@cs.sfu.ca)
Wan Gang Zeng (wgzeng@cs.sfu.ca)
Zhiwen Lin (zlin@sfu.ca)

CMPT885, Spring 2002
Supervised by Dr. Ljiljana Trajkovic

Department of Computing Science, Simon Fraser University

Table of Contents

Abstract

TCP has been most widely existing transport protocol in the Internet for many years, but problems were introduced while it's being applied into the wireless networks. It does not address the distinct features of wireless mobile computing, or very unreliable networks with high bit error rate (BER) in general. As a result, regular TCP suffers poor performance in the connection between mobile hosts and wired networks. In this paper, we are going to introduce a solution proposed by Kevin Brown and Suresh Singh, M-TCP. It has two significant advantages over other solutions: it maintains end-to-end TCP semantics, and it delivers excellent performance for environment where the mobile encounters periods of disconnection [3]. Then we describe our work on the performance evaluation of M-TCP using OPNET. And the result of the comparison with other TCP protocols is presented as well.

1. Introduction

In theory, transport protocols should be independent of the technology of the underlying network layer. In particular, TCP should not care whether IP is running over fiber or over radio. In practice, it does matter because most TCP implementations have been carefully optimized based on assumptions that are true for wired networks but which fail for wireless network.

The principle problem is the congestion control algorithm. Nearly all TCP implementations nowadays assume that timeouts are caused by congestion, not by lost packets or disconnection. Consequently, when a timer goes off, TCP slows down and sends less vigorously (e.g., Jacobson's slow start algorithm). First the sender retransmits the segment, then exponentially backs off its retransmit timer for the next retransmission, and closes its congestion window to one segment. Repeated errors will ensure that the congestion window at the sender remains small resulting in low throughput.

Nowadays, a cellar network infrastructure is typically used to connect mobile users to the Internet. A geographical region, such as a highway or campus, is divided into cells each of which contains a base station that provides a connection end-point for roaming mobiles.

It is likely that in order to provide high-bandwidth wireless connections, cell sizes will have to be reduced. Small cell sizes unfortunately result in small cell latencies that, in turn, cause frequent disconnections as a user roams. More serious problem caused by small cell latencies and frequent disconnections is that of serial timeouts at the TCP sender. A serial timeout is a condition wherein multiple consecutive retransmissions of the same segment are transmitted to the mobile while it is disconnected. All these retransmissions are thus lost. Since the retransmission timer at the sender is doubled with each unsuccessful retransmission attempt (until it reaches 64 sec), several consecutive failures can lead to inactivity lasting several minutes. Thus, even when the mobile is reconnected no data is successfully transmitted for as

long as 1 minute! Serial timeouts at the TCP sender can prove to be even more harmful to overall throughput than losses due to bit errors or small congestion windows.

M-TCP protocol was designed to work well in the presence of frequent disconnection events and over low bit-rate wireless links subject to dynamically changing bandwidth. In addition, it maintains end-to-end TCP semantics. The main idea of it is to force the TCP sender into persistent mode (by setting sender's window size to zero) while disconnection happens, and allow the sender to transmit at full speed while the mobile host is reconnected.

In this CMPT885 project, we implement the M-TCP protocol in a WLAN network using OpNet. We compare the performance difference of a network running with plain old TCP protocol against the same network running with the M-TCP protocol. This report examines our implementation of the M-TCP protocol and the performance improvement.

2. Technical Overview

2.1 TCP in General

TCP was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. TCP uses multiple timers to do its work. The most important of these is the retransmission timer, which is calculated based on the estimate of round-trip delay. The second timer is the persistence timer.

To make congestion control, TCP maintains two windows: the windows the receiver has granted and a second window, the congestion window. Each reflects the number of bytes the sender may transmit. The number of bytes that may be sent is the minimum of the two windows. When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection. It then sends out maximum segment. If this segment is acknowledged before the timer goes off, it adds one segment worth of bytes to the congestion window and sends two segments. If these two segments are acknowledged, the congestion window is increased by 2 maximum segments. In this way, the congestion window size grows exponentially until either a timeout occurs or the receiver's window is reached.

If the retransmission timer goes off before the acknowledgement comes in, the sender retransmits the segment and reduces the window size by half. From that point on, the source increases its window size by one unit every average round trip time. When timeout happens again, the window size is halved and the algorithm repeats.

The persistence timer is used when the receiver sends an acknowledgement with a window size of 0, telling the sender to wait, then sender go into persistence mode and set the persistence timer. If persistence timeout, the sender will send a one byte segment as a probe to the receiver. The response to the probe gives window size. If it is still zero, the persistence timer is set again and the cycle repeats. If it is nonzero, sender will go out of persistence mode and new data can now be sent. If no persistence timer and the packet from the receiver for updating the window is lost, then both receiver and sender will wait for each other to do something forever.

2.2 M-TCP Protocol

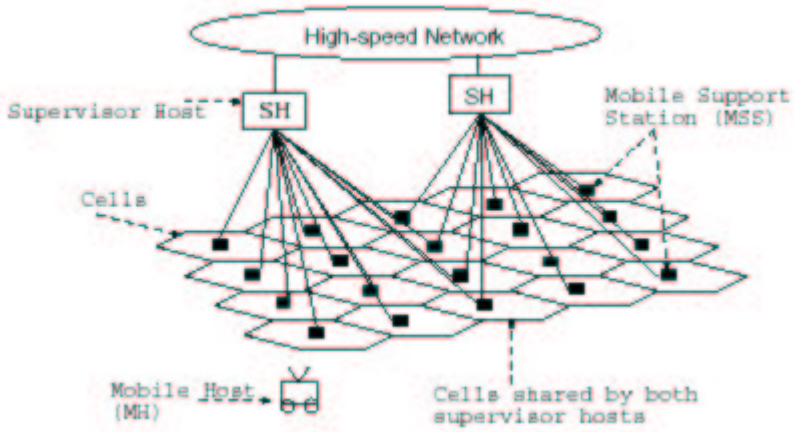

Figure 1: Proposed Architecture

Figure 1 shows our architecture, which is a three level hierarchy. At the lowest level are the mobile hosts (MH) who communicate with MSS nodes in each cell. Several MSSs are controlled by a machine called the Supervisor Host (SH). The SH is connected to the wired network and it handles most of the routing and other protocol details for the mobile users. In addition it maintains connections for mobile users, handles flow-control and is responsible for maintaining the negotiated quality of service. These SHs thus serve the function of gateways.

According to the hierarchy above, we simplify the model as figure 2. M-TCP split TCP connection in two at the SH. The TCP client at the SH (called SH-TCP) receives segments transmitted by the sender on the fixed network and it passes these segments to the M-TCP client for delivery to the MH. ACKs received by M-TCP at the SH are forwarded to SH-TCP for delivery to the TCP sender.

When SH-TCP receives a segment from TCP sender, it passes the segment to the M-TCP client. However, it does not ACK this data until the MH does. It is easy to see that this behavior ensures that end-to-end TCP semantics are maintained. However, how do we ensure that the sender does not go into congestion control when ACKs do not arrive (because the MH was temporarily disconnected and did not receive the data, or could not send the ACKs)?

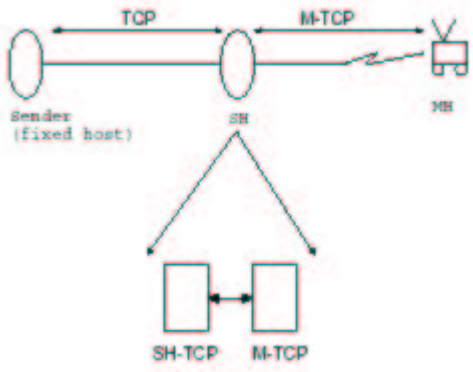

Figure 2: Setting up a TCP connection

When SH_TCP forward ACK from MH to FH, it holds the last byte ACK, it means it just ACK the sequence number up to Ack – 1 to FH in normal way and one last byte is always left unacknowledged. When timeout happens, SH-TCP sends this last byte ACK to FH. This ACK will also contain a TCP window size update that sets the sender's window size to zero. When FH receives this ACK and is forced into persist mode. While in this state, it will not suffer from retransmit timeouts and will not exponentially back off its retransmission timer, nor will it close its congestion window. This state of the sender does not change no matter how long the disconnection period lasts. Because only new ACK packet can force the host into persistence mode according to the normal TCP protocol, this is the reason why we keep the last byte ACK.

When MH is reconnected, MH will send a reconnection ACK to M-TCP, then M-TCP checks the packet and knows that MH is reconnected, therefore it will retransmit all the packets which have not been acknowledged to MH and forward the ACK packet to SH-TCP and FH so that SH-TCP and FH can resume their full window size and go out of persistence mode, begin sending data again. Since the sender never timed out, it never performed congestion control or slow-start. Thus the sender can resume transmitting at full-speed!

We ignore the high BER, because we believe that in most mobile environments link layer solution will ensure that the bit error seen at the TCP layer is small.

Even if the mobile environment does have high BER, the M-TCP sender will come out of persist mode quickly and resume regular data transmissions. This is because, when in persist mode, M-TCP will send persist packets to the MH who will be forced to respond when it receives these packets. Currently, the persist packets are generated at increasing intervals starting at 5 seconds and doubling until the interval becomes 60 seconds. We could change the interval to be equal to the RTT between the SH and the MH to ensure earlier recovery (we have not implemented this simply because of our assumption of low BER).

3. Performance Evaluation

The performance evaluation of M-TCP is done in OPNET. OPNET provides sets of models for network modeling and simulation. With carefully chosen models, one can model most of existing network components with/without even small modifications.

3.1 Model Setup in OPNET

We following the model setup described in [3] for our evaluation, also see in Figure 1. Three node models are selected from two OPNET standard model sets, Wireless LAN and Ethernet, to build our simulation model.

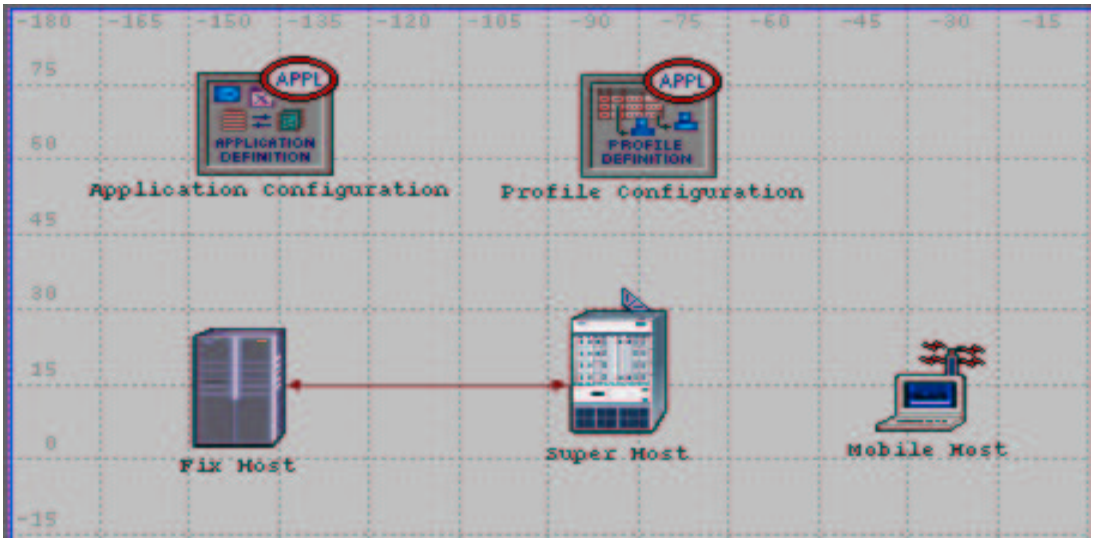

Figure 3: Model Setup in OPNET

The standard model wlan_ethernet_router, a wireless LAN ethernet router, is used to model the supervisor host in our topology; wlan_wkstn_adv, a wireless LAN workstation, is used for mobile host, and ethernet_server is used for fixed host.

These nodes are chosen for various reasons. Ethernet_server is an ethernet server model with server applications running over TCP/IP and UDP/IP. And it has an interface to 1 Ethernet connection at 10 Mbps, 100 Mbps, or 1000 Mbps. These meet our requirements of having TCP connection to supervisor host through ethernet. Then wlan_wkstn_adv is chosen for mobile host for the similar reason. It represents a workstation with applications running over TCP/IP and UDP/IP. It supports 1 underlying Wlan connection at 1Mbps, 5.5 Mbps, and 11 Mbps. These properties enable us to connect it with supervisor host. Also, we use the mobile version of the station to meet the requirement of having frequently handoff to occur. Lastly, the trickiest part of this is to choose the proper model for supervisor host. We use the wireless LAN based router, wlan_ethernet_router, to have the connection between the fixed host and the mobile host, since it has the capability of making connections to both Ethernet (1 Ethernet port available) and wireless LAN (one wireless LAN port available). The other most important feature is that it also comes with TCP protocol implemented. Because of this, it is an ideal place to have M-TCP implemented inside this node.

3.2 Model Modification

Even though these OPNET standard models provide the basic connections, we still need to make modifications to implement M-TCP in our model.

As we described in section 2.2, there's no need of any modification in the wired LAN domain, including fixed host. In fact, this is one of the most important features of M-TCP, as well as most of the mobile TCP protocol, which doesn't requires the modification of the existing internet components. Then the TCP on mobile host is modified into M-TCP client. And the most of the work is required in supervisor host, where the original TCP is modified to two layers, SH-TCP and M-TCP. The SH-TCP on supervisor host is used to make the communication between the supervisor host and the wired lab, the fixed host in the Ethernet.

The M-TCP on supervisor host is used to make communication between the supervisor host and the wireless LAN, the mobile host in the wireless LAN. In the following two sections, we will detail the modifications for this two nodes.

3.3 Modifications of Mobile Host

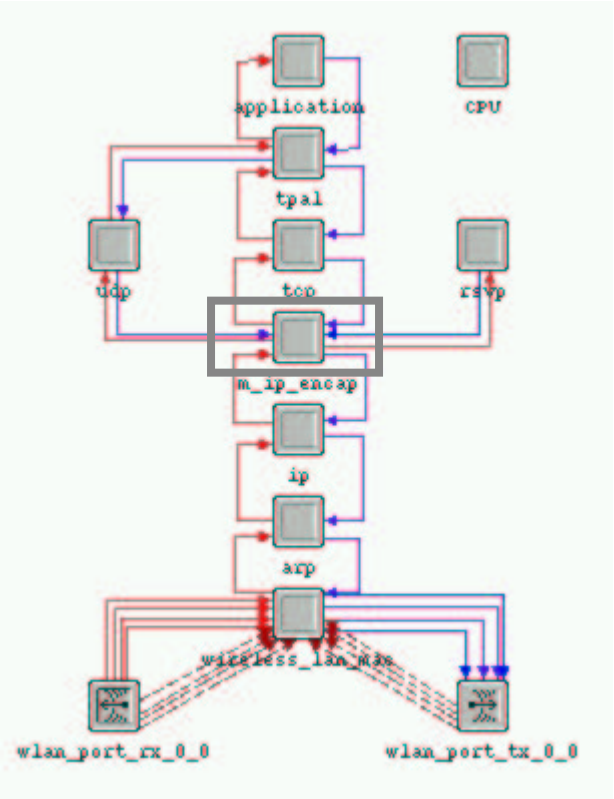

Figure 4: Structure of Mobile Host

Figure 4 shows the structure of Mobile Host Node. We modify the *ip_encap* process for two purposes:

1) Cutting the packet stream periodically.

This will allow we simulate hand-offs in the mobile host. Theoretically, any processes along the data path between MH and SH can break the packet stream. We make modification to *ip_encap* is for implementation considerations. This process is relatively simple: it is responsible for decapsulating IP packets to higher layer and encapsulate packets from higher layer into IP packets. In our experiments, we intercept the network traffic both in decap and encap states. This is shown in the modified state diagram of *ip_encap* (Figure 5).

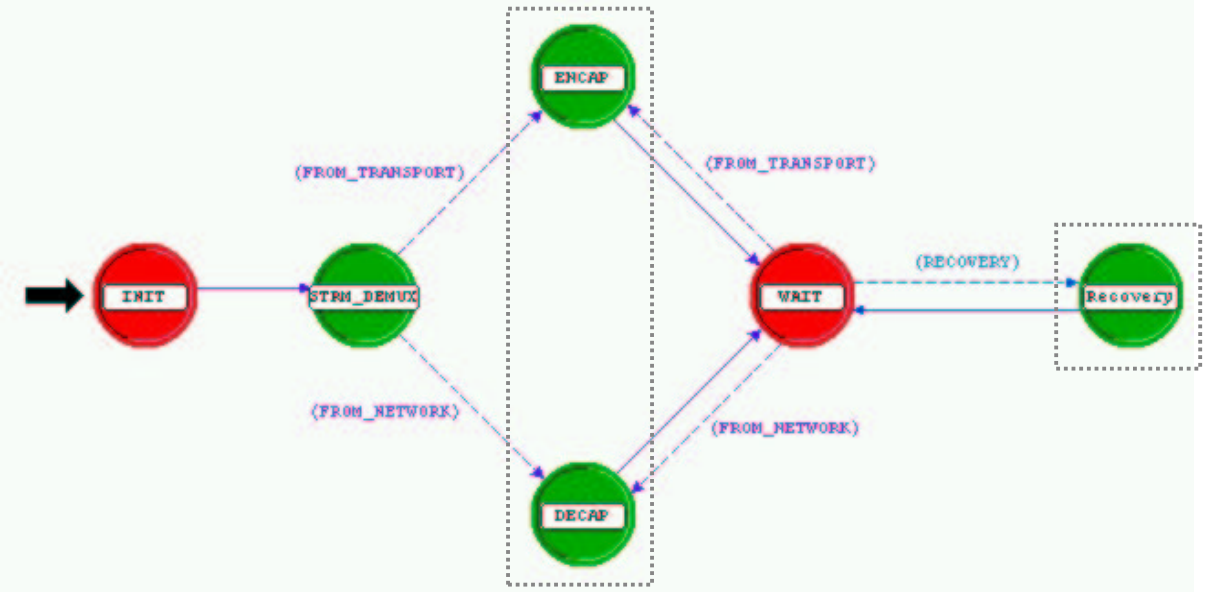

Figure 5: Modified *ip_encap* State Diagram

We add 4 attributes in the process module:

`BrokenEnable:` if is set to 1, the data stream will be broken cyclically. This increases the flexibility of our simulations.

`CycleTime:` this attribute indicates the time in seconds between two breakdowns

`BrokenStartTime:` in each breaking cycle, the beginning time of breaking

`BrokenEndTime:` in each breaking cycle, the ending time of breaking

Figure 6 shows a typical setup of these attributes in our simulation, we chose cycleTime=300, brokenStartTime=270, brokenEndTime=300. This generates a 30-second broken link every 5 minutes.

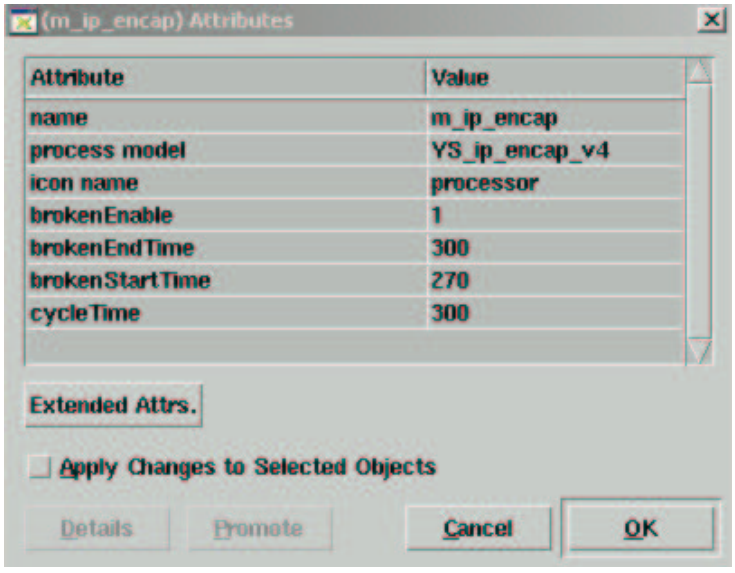

Figure 6: A typical setup of simulation

The C code for a cyclically breaking in DECAP state is listed below(similar code in ENCAP state):

```
/*****************************************************************/
/*                                                               */
/* if it is time to break the link, then intercept the packets and   */
/* destroy it . Otherwise forward it to next layer.              */
/*                                                               */
/*****************************************************************/
if ( isBroken() )
{
   // Schedule a time to send recovery ACK packet
   if ( initScheduledRecovery == 0 )
   { op_intrpt_schedule_self( getRecoveryTime(), 0 );

     // set schedule flag
     initScheduledRecovery = 1;
    }

   printTrace("***** link is broken now.");
   op_pk_destroy( pkptr );
}
else
       { op_pk_send (pkptr, output_strm);

         // reset schedule flag
         initScheduledRecovery = 0;
       }
```

2) Sending recovery packet to the Fix Host when the connection is restored.

This is an essential part of M-TCP protocol. When the mobile recovers from the hand-off, it will send an ACK packet to FH to restore the connection. This ACK packet is the last ACK packet sent to FH by MH with a normal receive window size (>0).

A RECOVERY state has been added to the *ip_encap* diagram. This is used to send the recovery packet to the Fixed Host. The recovery interrupt is scheduled when the link is broken (See above code). When the time it recovers, the interrupt will force the process into RECOVERY state. The core processing code in Recovery state is like this:

```
   ..........
// copy from the sample pacekt
seg_ptr = op_pk_copy ( sample_MH_pkt );

if (op_pk_nfd_access (seg_ptr, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
  { printTrace ("Unable to get data from TCP Segment to get persist tcp pkt.\n");
  }

// setting the tcp packet  fields
tcp_seg_fd_ptr->ack_num = fh_last_seq_num;
tcp_seg_fd_ptr->seq_num = mh_last_seq_num ;
tcp_seg_fd_ptr->rcv_win = 8760;          // original default receive window size

printTrace( "structure of persist ACK pkt is:");

// send the packet to ip layer
op_pk_send( myEncap(seg_ptr), outstrm_to_network );
  ........
```

3.4 Modifications of Supervisor Host

Since we use a wireless/LAN router as Supervisor Host, which doesn't have any data stream to layers higher than IP, we need to adjust the IP layer code so that packets can be directed to higher layers such as M-TCP. We modified *ip* and *ip_encap* to meet our need. Figure 7 shows the modified structure of SH node.

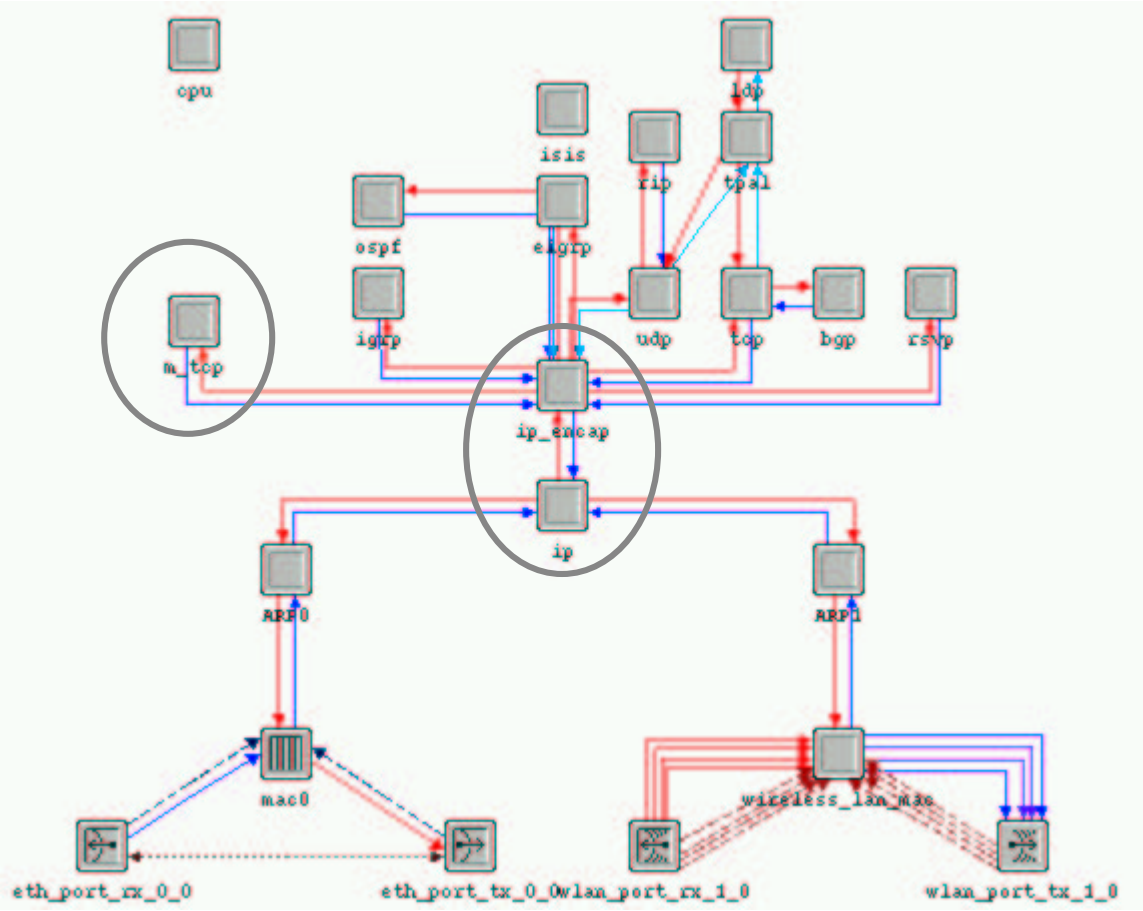

Figure 7: Modified Structure of Supervisor Host

3.4.1 IP Routing Process

Routing is performed in the *ip* process. The standard routing module has the structures shown in Figure 8. In this figure, we can see that routing task is carried out in the *ip_dispatch* and *ip_rte_central_cpu.* If the packet has reached its destination node, the routing part will forward it to higher layer such as TCP and UDP. In a router, however, the packets are routed directly through *ip_rte_central_cpu* module. We modified the *ip_rte_central_cpu*, and let all packets go through *ip_encap* to reach M-TCP process. The core code to re-route packets are listed below:

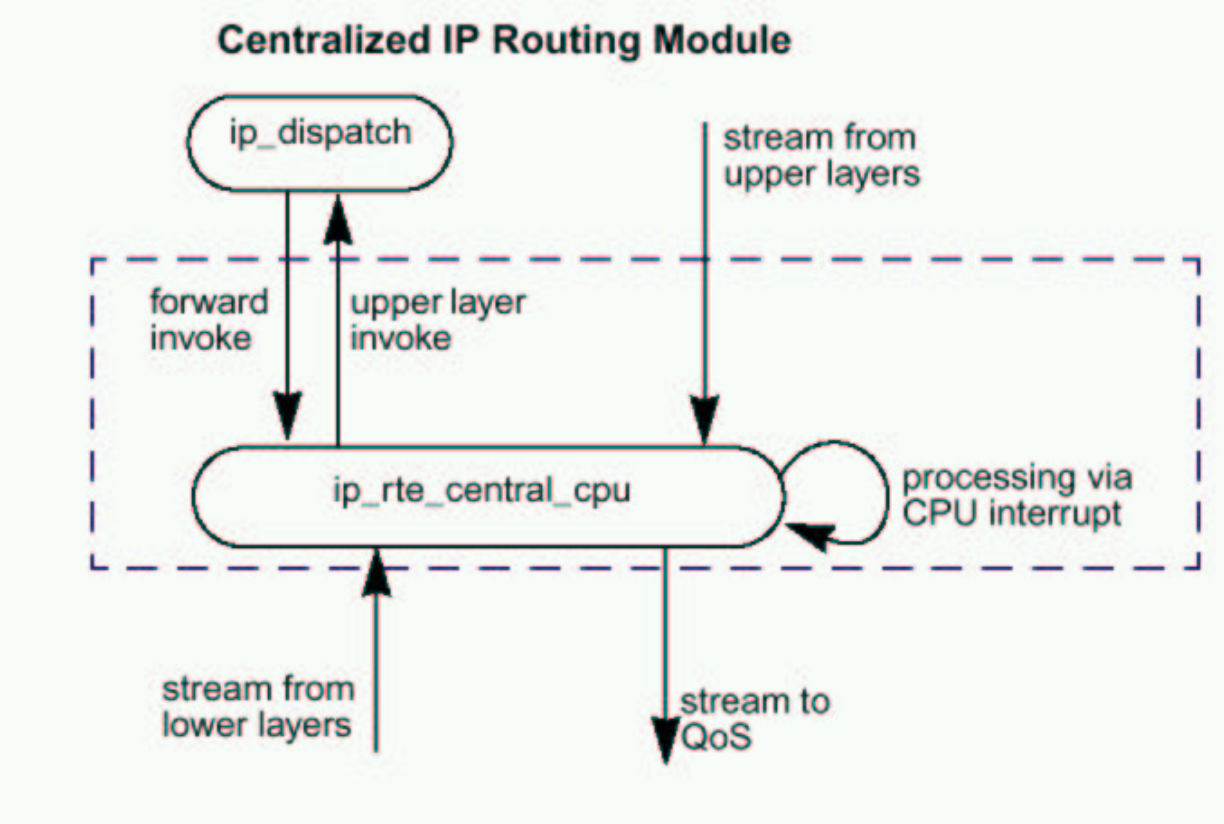

Figure 8: IP routing architecture

```
static void ip_rte_central_cpu_packet_arrival (void)
  {
        . . . . . .

        /* Packet coming from some stream */
        instrm = op_intrpt_strm ();

        pkptr = op_pk_get (instrm);
        if (pkptr == OPC_NIL)
            ip_rte_cpu_error ("Unable to get packet from input stream.");

        if ( ( (instrm== 1 )      // from lan arp0
             || ( instrm == 2 ))  // from wireless arp1
             && ( op_id_self() == 723 ) ) // is router
        { printTrace( "Directly sending packet to ip-encap" );

          // send to ip_encap stream
          op_pk_send( pkptr, 0 );
          FOUT;
        }

       .  .  .  .  .
  }
```

3.4.2 IP Encap Process

The original *ip_encap* process is able to to decap IP packets and encap packets from higher layers. Since our M-TCP is only a TCP-like layer above it, we bypass most of the functions in

ip_encap and let the packets flow through unchanged. Here is the code in Encap state enter executive:

```
        . . . . . .
/* Obtain the packet arriving from a higher protocol layer. */
printTrace( "Entering router ip encap");

input_strm = op_intrpt_strm();

pkptr = op_pk_get (input_strm);
if (pkptr == OPC_NIL)
   ip_encap_error ("Unable to get packet from transport layer.");

if ( input_strm == INSTRM_FROM_MTCP )
{
 printTrace("ENCAP To send packet to ip layer");

 // send the packet to IP layer
 op_pk_send_forced( pkptr, outstrm_to_network );
 goto en1;
}
      . . . . .
```

3.4.3 M-TCP Process

We added a process M-TCP to replace the TCP layer in the router. This is our implementation of the M-TCP/SH-TCP protocol. When a packet arrives, we check its origin address. If it is from FH, we will put a copy of this packet into a local queue and forward it to MH. If the packet fails to reach MH, we will retransmit the copy in the queue. If the incoming packet is from MH, we change its ACK number in the packet, let ack_num=ack_num-1. This will give us a remaining byte of un-Acked data. We then forward the modified packet to FH. If the M-TCP detects that there is a broken link between SH and MH, it will send an ACK packet of the last un-Acked byte with a receiver window size 0. This is called a persistent ACK pacekt. The FH will then turn into persist mode.

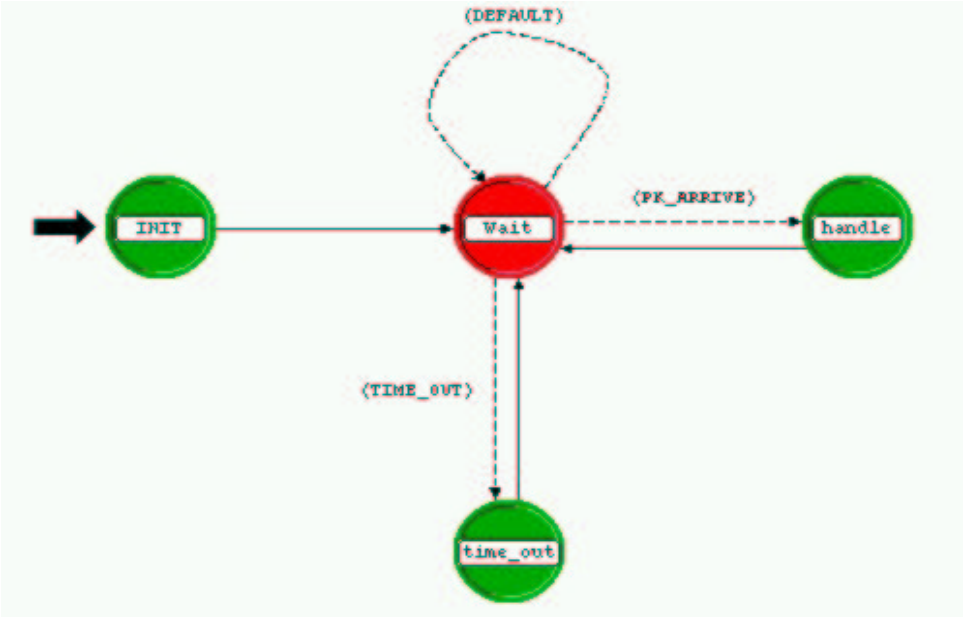

Figure 9: State diagram of M-TCP

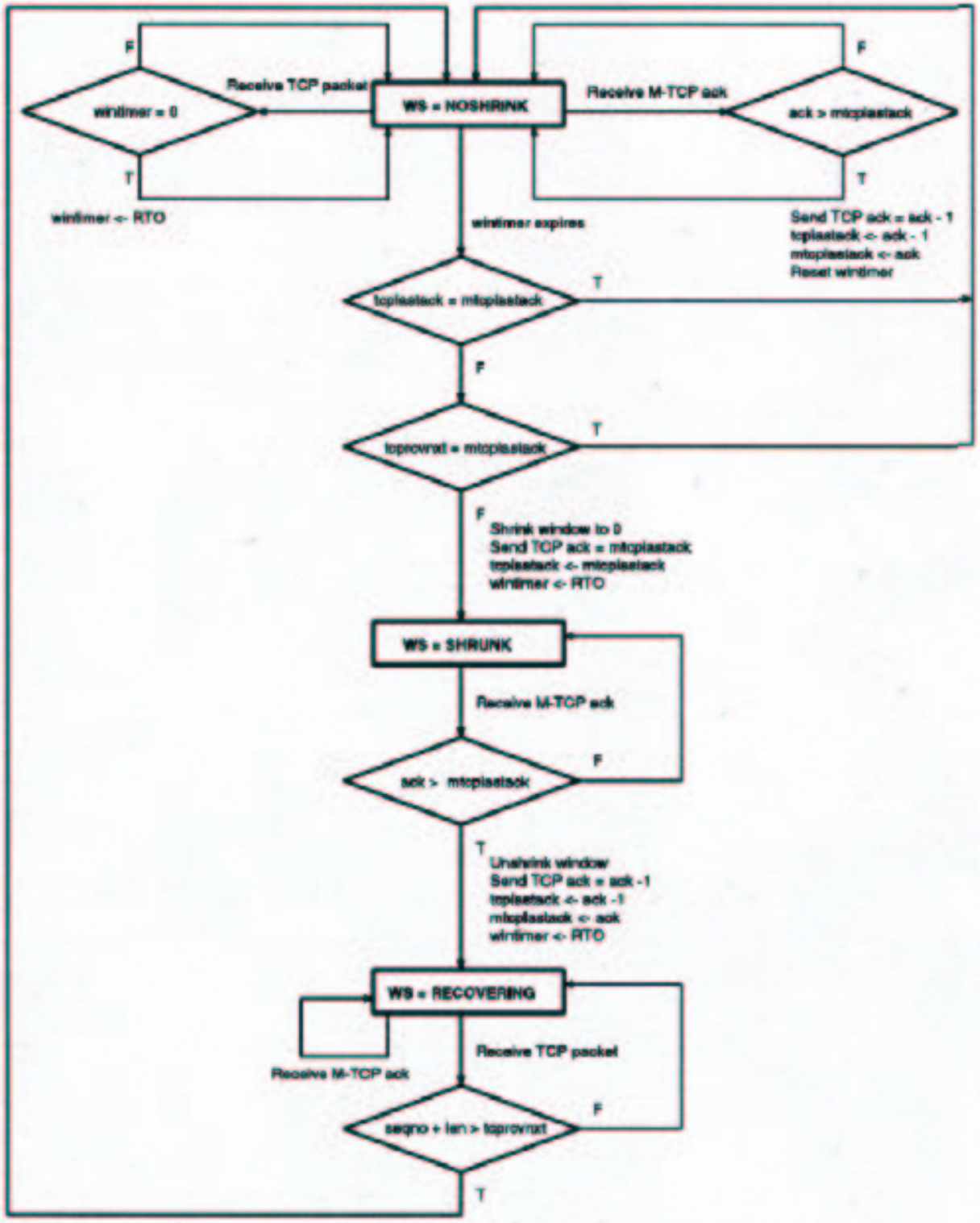

Figure 10: M-TCP Flowchart (a)

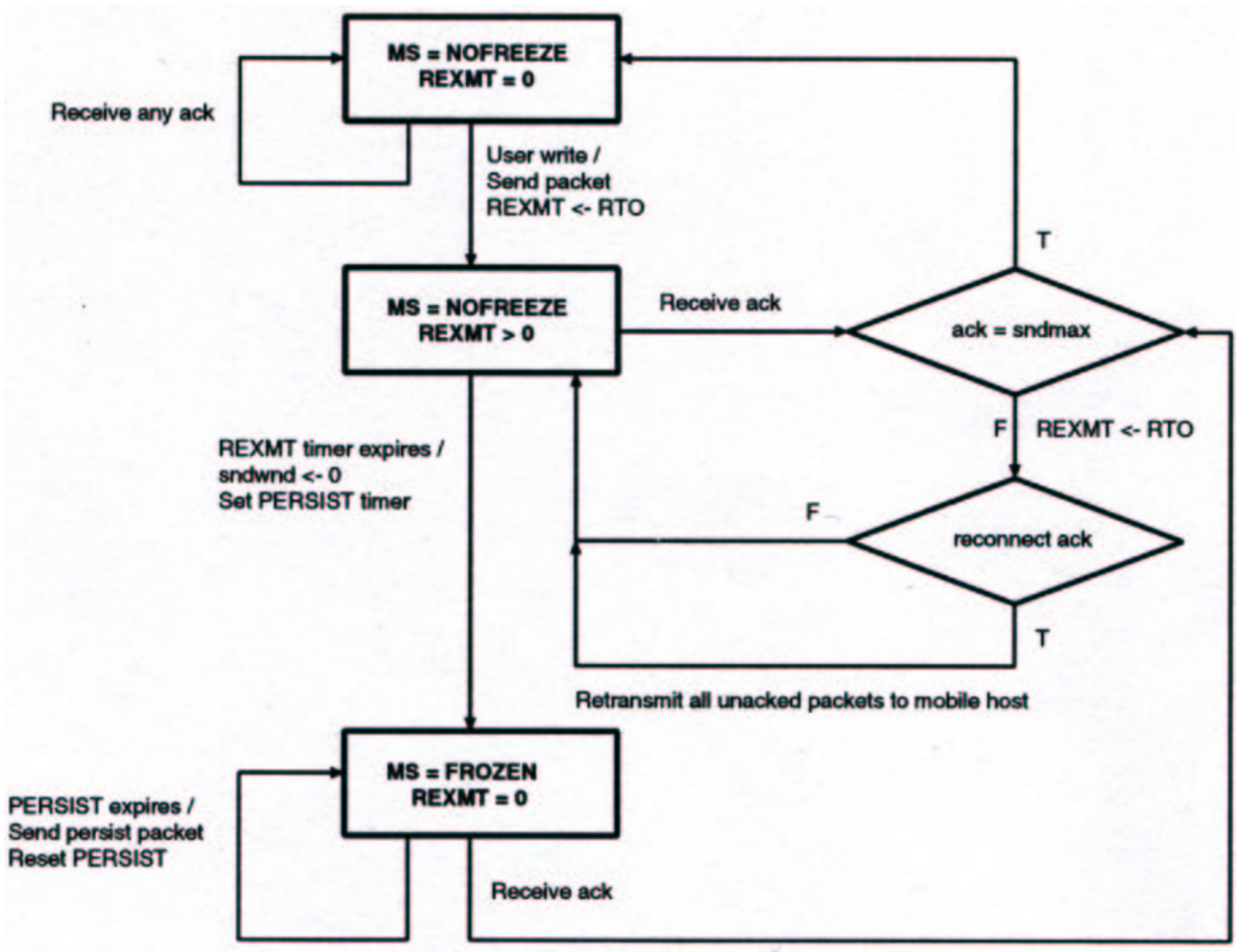

Figure 11: M-TCP Flowchart (b)

Figure 9 depicts the state diagram of M-TCP. After initialization, the MTCP process stays at *wait* state. When a packet comes in, it will deal with it in the *handle* state. If a timeout occurs (MH disconnects), it will enter *time_out* state and generate a persistent packet to the Fixed Host.

The main handling logic in *handle* is shown here:

```
    . . . . . .
analyse_incoming_pkt( ip_pkptr );

if ( isFromFixHost() )
    handle_Fix_Host_Pkt ( ip_pkptr );

if ( isFromMobileHost() )
    handle_Mobile_Host_Pkt( ip_pkptr );

    . . . . .
```

The complete flowcharts are shown in Figure 10 and Figure 11.

3.5 Performance Comparison

3.5.1 Scenario Setup

We setup three scenarios to evaluate the performance of M-TCP. Since our focus is on the connection-specific congestion window, in our simulation, we only establish a connection between Mobile Host and Fixed Host. The application using this connection is FTP. We investigate the ftp downloading performance under three scenarios.

Scenario 1: TCP with no disconnections
This is the ideal network situation without any disconnections between MH and FH. It serves as the baseline of comparison.

Scenario 2: Normal TCP with disconnections
Normal TCP protocol is used in all hosts and router. Disconnections are scheduled every 5 minutes. Each breakdown time is 30 seconds. The reason to choose 30 seconds is based on the fact we found out: If the disconnection time is greater than 30 seconds, the connection will be closed by the FH. In order to provide comparable result, we pick 30 seconds as the breaking duration.

Scenario 3: M-TCP with disconnections
M-TCP is deployed on SH. Disconnections are scheduled every 5 minutes. Each breakdown time is 30 seconds.

The same data stream was fed into the network by FH in these 3 scenarios. The disconnection period is the last 30 seconds of every 5-minute interval.

3.5.2 Result Comparison

a) **Congestion Window Size**

Figure 12 shows the comparison of the congestion window size (*cwnd*) on FH. We can see the M-TCP line is quite close to the TCP on ideal network condition. We can notice that there are a few horizontal segments in the M-TCP line, which indicate the period when FH was in persist mode (during which *cwnd* is frozen). Contrarily, the *cwnd* in normal TCP scenario dropped drastically during the disconnection period and never rose as high as M-TCP. This suggests that M-TCP does help preventing the *cwnd* from decreasing during hand-offs.

b) **Throughput**

Figure 13 illustrates the throughput of three scenarios. The top line is the ideal TCP. The second is M-TCP and the last one is TCP with disconnections. We observe that M-TCP had a higher throughput than TCP in a network with disconnections. This is expected because M-TCP tends to have higher congestion window size, which allows it to transmit more data once. Moreover, eventhough each disconnection period is only 30 seconds, the performance difference is still obvious. In the real mobile network environment, the hand-off time can range

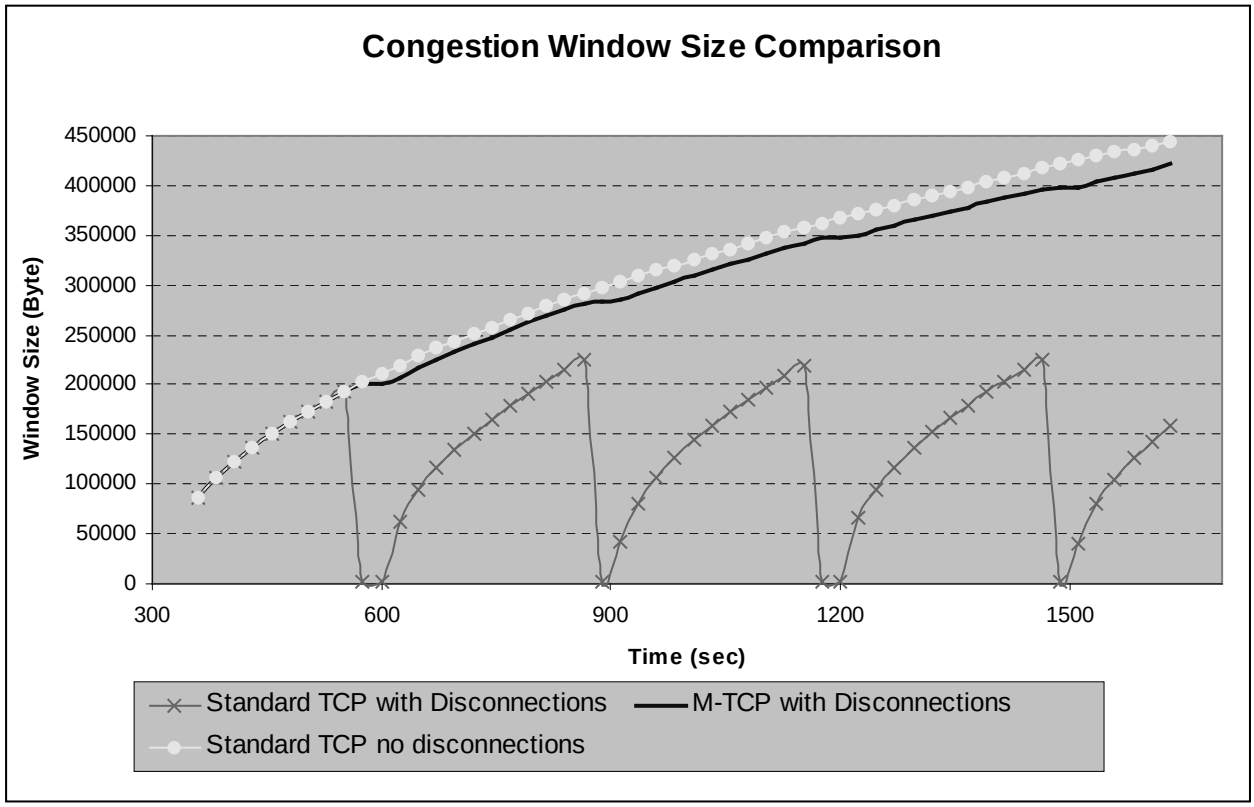

Figure 12: Congestion Window Size Comparison

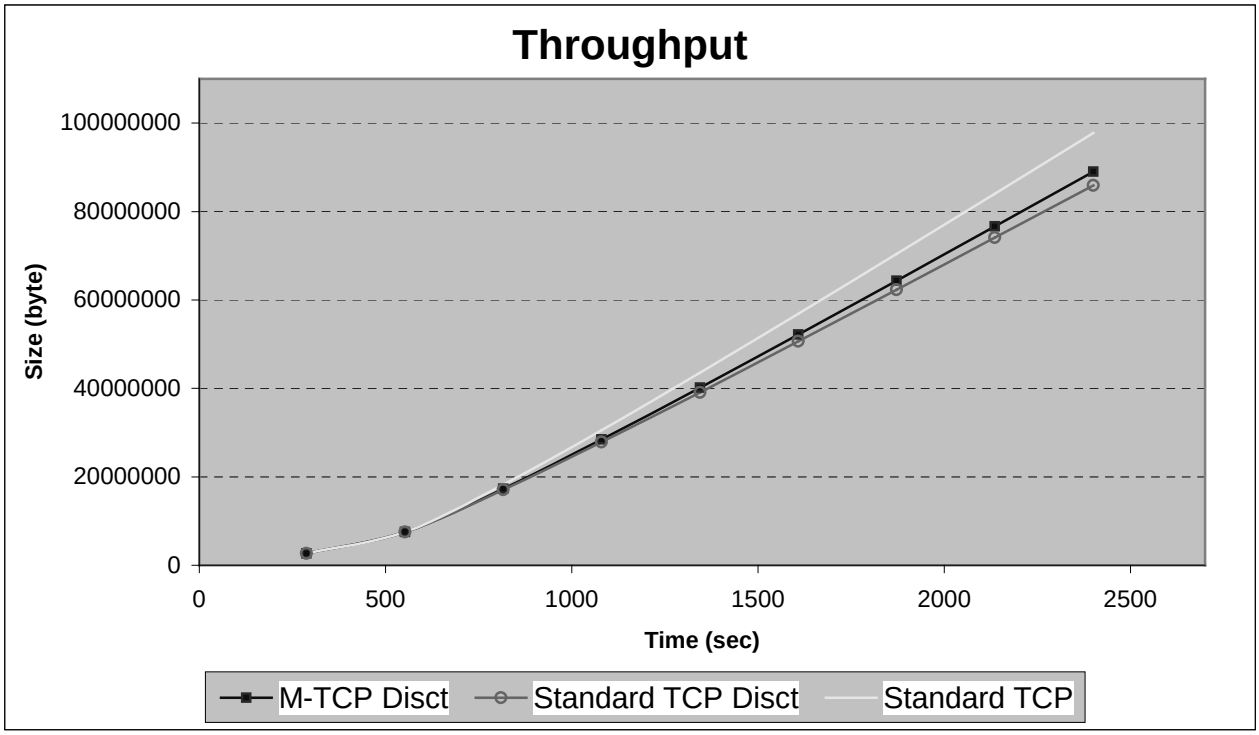

Figure 13: Throughput Comparison

from 10 seconds to a few minutes [3]. In this sense, the difference in throughput will be much greater.

c) **RTO (Retransmission Timeout)**
Figure 14 indicates the difference in RTO over three scenrios. M-TCP has a similar RTO with ideal TCP while the TCP with disconnections has a much higher RTO. This is due to the fact that MH is "really" disconnected from FH during normal TCP disconnections, while in M-TCP's case, the FH is shielded from knowing the disconnections.

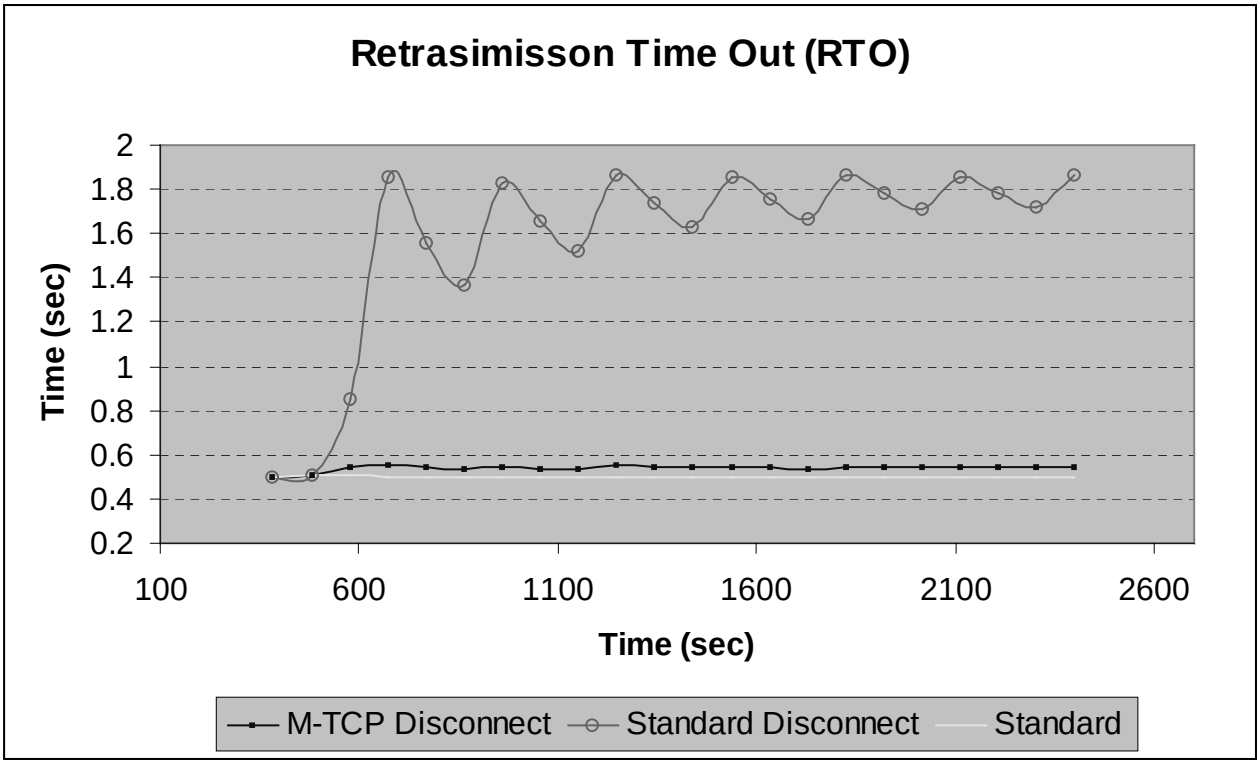

Figure 14: RTO comparison

4. Conclusions

In this project, we implemented a M-TCP protocol for use in mobile networks. This protocol is based on the observation that, in a mobile computing environment, frequent disconnections(caused by signal fades, handoff or lack of bandwidth) should be dealt with properly. We setup three scenarios to evaluate the performance of M-TCP in mobile networks. Our simulation results show that M-TCP outperforms normal TCP in terms of congestion window size, throughput and RTO. Its performance is very close to the non-disconnected network. Therefore M-TCP may be nice candidate for addressing the disconnection problems in mobile netoworks.

5. Role of Group Members

Meihua Judy Zhan	Responsible for design and setup of the simulation scenarios, presentation and demo. Implemented Mobile Host disconnection and its recovery, modification of Supervisor Host routing module, part of M-TCP kernel code. Wrote part of the final report.
Wan Gang Zeng	Responsible for part of Supervisor Host M-TCP kernel implementation, presentation and demo, part of setup simulation scenarios and interim report.
Zhiwen Lin	Responsible for part of Supervisor Host M-TCP kernel implementation, part of final report.

References

[1] J. Border, M. Kojo, J. Griner, G. Montenegro, Z. Shelby. "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations". RFC 3135, June 2001. http://www.ietf.org/rfc/rfc3135.txt (Last visit on Feb. 5, 2002)

[2] G. Montenegro, S. Dawkins, M. Kojo, V. Magret, N. Vaidya. "Long Thin Networks". RFC 2725, January 2000. http://www.ietf.org/rfc/rfc2757.txt (Last visit on Feb. 5, 2002)

[3] Kevin Brown, Suresh Singh. "M-TCP: TCP for Mobile Cellular Networks". ACM, July 1997. http://www.acm.org/sigcomm/ccr/archive/1997/oct97/ccr-9710-brown.pdf (Last visit on Feb. 30, 2002)

[4] Kevin Brown, Suresh Singh. "A Network Architecture for Mobile Computing". Proc. IEEE INFOCOMM'96, S.F. CA, March 1996. http://www.it.kth.se/~jiang/doc/literature/bibdata/Bro96.pdf (Last visit on Feb. 5, 2002)

[5] S. Singh. "Quality of Service Guarantees in Mobile Computing". J. Computer Communications, Vol. 19, pp. 359-371, 1996.

[6] K. Seal and S. Singh. "Loss profiles: A Quality of Service Measure in Mobile Computing". Journal of Wireless Networks, vol. 2, no. 1, pp. 45-61, 1996.

[7] Ajay Bakre and B.R. Badrinath. "I-TCP: Indirect TCP for Mobile Hosts". In Proc. of 15th Int'l Conf. on Distributed Computing Systems (ICDCS), May 1995. http://users.ece.gatech.edu/~siva/ECE4894/list/7.pdf. (Last visit on Feb. 24, 2002)

Appendix A: Code

M-TCP kernel:

```
/* Process model C form file: mtcp_kernel.pr.c */
/* Portions of this file copyright 1992-2001 by OPNET Technologies, Inc. */


/* This variable carries the header into the object file */
static const char mtcp_kernel_pr_c [] = "MIL_3_Tfile_Hdr_ 80C 30A opnet 7 3CBA8F4E 3CBA8F4E 1
payette mjzhan 0 0 none none 0 0 none 0 0 0 0 0 0
";
#include <string.h>


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

#if defined (__cplusplus)
extern "C" {
#endif
FSM_EXT_DECS
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif


/* Header Block */

#include <opnet.h>

/* Header Block */

#include <ip_addr_v4.h>
#include <oms_dt.h>
#include <tcp_api_v3.h>
#include <tcp_v3.h>
#include <oms_pr.h>
#include <oms_qm.h>
#include <oms_tan.h>
#include <tcp_seg_sup.h>
#include <ip_notif_log_support.h>
#include <ip_rte_v4.h>
#include <ip_higher_layer_proto_reg_sup.h>
#include <tcp_seg_sup.h>

#include "/cs/grad1/mjzhan/op_models/msg.h"

static Ici * iciptr;
static Packet * pkptr;

#define PK_ARRIVE      ( op_intrpt_type() ==  OPC_INTRPT_STRM )
#define TIME_OUT       ( (op_intrpt_type() == OPC_INTRPT_REMOTE )|| (op_intrpt_type() ==
OPC_INTRPT_SELF ))
#define DEFAULT        ( (op_intrpt_type() != OPC_INTRPT_STRM) && (op_intrpt_type() !=
OPC_INTRPT_REMOTE ) && (op_intrpt_type() != OPC_INTRPT_SELF) )

#define FROM_ENCAP_STRM     0
#define TO_ENCAP_STRM     0


#define FIX_HOST_ADDR     0xc0000009
#define MOBILE_HOST_ADDR  0xc0000101

static int isInit =0;

static unsigned fix_host_port ;
```

```
static unsigned mobile_host_port;

static int      fh_data_len;
unsigned fh_last_seq_num;
unsigned fh_last_ack_num;

// the ack from Mobile Host, seen by router, which is
// ack seen by Fix host plus 1
unsigned mh_last_ack_num;
unsigned mh_last_seq_num;

// shared variables for incoming ip pkt
static  IpT_Address                  org_addr;
static  IpT_Address                  dest_addr;
static  int                                 type_of_service;
static   int                         protocol_type ;

// to store a sample of the Mobile host packet for later use
Packet * sample_MH_pkt;


int transMode=0;
long transBeginCycle=0;

// rto for MH
double MH_rto=0.6;

#define MH_DISCONNECT_TIME_OUT  100

extern long cycle;
extern int isBrokenPeek(void );



/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef  BIN
#undef  BOUT
#define BIN             FIN_LOCAL_FIELD(last_line_passed) = __LINE__ - _block_origin;
#define BOUT    BIN
#define BINIT   FIN_LOCAL_FIELD(last_line_passed) = 0; _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
        } mtcp_kernel_state;

#define pr_state_ptr                        ((mtcp_kernel_state*) SimI_Mod_State_Ptr)

/* This macro definition 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
#define FIN_PREAMBLE   mtcp_kernel_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };
typedef struct
{
 Packet * pkt;
```

```
 unsigned seq_num;
 unsigned data_len;
 double   arr_time;
 double   time_out;

} QueueItem ;

#define MAXQUEUESIZE 5000

QueueItem queue[MAXQUEUESIZE];

int queueHead=0; /* point to first empty item */
int queueTail=0; /* point to last item */

int nextQueueIndex( int n )
{
  if ( n==MAXQUEUESIZE-1 ) n=0;
   else n++;

  return n;
}

int isQueueEmpty()
{ return ( queueHead == queueTail );
}

int hasReachedEnd( int n )
{ return ( n == queueTail );
}


void putInQueue( Packet * pkt, unsigned seq_num, unsigned  data_len )
{
       if ( nextQueueIndex(queueTail) != queueHead )  /* the queue is not full  */
       { queueTail=nextQueueIndex(queueTail);

         queue[queueTail].pkt = pkt;
         queue[queueTail].seq_num = seq_num;
         queue[queueTail].data_len = data_len;
         queue[queueTail].arr_time = op_sim_time();
       }
}

// purge packets that have been ACKed by lastAck number
void purgeFromQueue( unsigned lastAck )
{
  int nextItem ;

  //  nothing to purge
  while ( !isQueueEmpty() )
  {
    nextItem = nextQueueIndex( queueHead );

       // purge ACKed packet(s)
    if ( queue[nextItem].seq_num + queue[nextItem].data_len <= lastAck )
       { queueHead = nextItem;
         op_pk_destroy( queue[nextItem].pkt );
       }
       else
         break;
  } // while

}


// re-send all packets that hasn't reached the destination
void resendPackets(  )
{ int first;
  int n=0;
  Packet * pkt;

  if ( isQueueEmpty () ) return ;
```

```
  first = queueHead ;

  while ( !hasReachedEnd( first) )
  { first = nextQueueIndex( queueHead );

    pkt = op_pk_copy( queue[first].pkt );
    op_pk_send( pkt, TO_ENCAP_STRM ); // sequentially re-send pkts
        n++;
  }
}

int queueSize()
{ int size = queueTail - queueHead;
  if ( size<0 ) size += MAXQUEUESIZE ;

  return size;
}
//---------- end of queue functions  ---------

static int isFromFixHost()
{ return  (org_addr == FIX_HOST_ADDR ) ;
}

static int isFromMobileHost()
{ return  (org_addr == MOBILE_HOST_ADDR ) ;
}


// for testing
extern int brokenStartTime;
extern int brokenEndTime;
extern int cycleTime;
extern long cycle;

// this is for debug
// test if the link is broken
static int isBroken()
{
  double t=op_sim_time();
  long tt =  t;
  cycle=tt%cycleTime;

  if ( (cycle>brokenStartTime) && (cycle<brokenEndTime ))
          return 1;

  return 0;
}



/* This function re-calculate the RTO based on the arrival time of ACK packet */
/* When an ACK comes in, we get the send time of the packet from the queue and*/
/* decide the RTT of it                                                       */
static double calculateRTO( unsigned ack_num, double arrive_time )
{
  int nextItem =queueHead;
  double rtt;


  //  try to find the paket in queue
  while ( !hasReachedEnd( nextItem) )
  {
    nextItem = nextQueueIndex( queueHead );

        // find the ACKed packet(s)
    if ( queue[nextItem].seq_num + queue[nextItem].data_len == ack_num )
        { // found
          rtt = arrive_time - queue[nextItem].arr_time;
          break;
        }
  } // while
```

```
  /* calculate RTO base on RTT   */

  /* DOES NOT WORK NOW */

  /* 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;


     return 0.45;   // we return the fix RTO at this moment
}


// analyse an imcoming pacekt, record the src/dest address,
// type of service, and protocol
static void analyse_incoming_pkt( Packet * ip_pkptr )
{   IpT_Dgram_Fields* ip_dgram_fd_ptr;

    /*  Get the fields structure from the packet.                   */
        op_pk_nfd_access (ip_pkptr, "fields", &ip_dgram_fd_ptr);

        /* Determine the source address of the data.                */
        org_addr = ip_dgram_fd_ptr->src_addr;

        /* Determine the destination address of the data.           */
        dest_addr = ip_dgram_fd_ptr->dest_addr;

        /* Determine the type of service. This attribute needs to be passed to tcp        */
        /* layer (for tcp connectios),^^or to tpal (for udp connections).                 */
        type_of_service = ip_dgram_fd_ptr->tos;

        /* Determine the protocol number.                                          */
        protocol_type = ip_dgram_fd_ptr->protocol;

}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
// modify ip pkt from MH, change ack_num  to the value of the input parameter, //
// other fields are untouched                                             //
//                                                                        //
////////////////////////////////////////////////////////////////////////////
static void modify_MH_pkt( Packet * ip_pkptr, unsigned ack_num )
{ Packet * orginal_pkt;
  Packet * ipcopy_pkt;

  TcpT_Seg_Fields *    tcp_seg_fd_ptr;


  // copy a new instance of the packet
  ipcopy_pkt = op_pk_copy( ip_pkptr );

  // extract the tcp packet out
  if (op_pk_nfd_get (ipcopy_pkt, "data", &orginal_pkt) == OPC_COMPCODE_FAILURE)
  { printTrace ("modify MH: Unable to get tcp pkt from IP datagram again.\n");
        return ;
        }

  // delete this packet
  op_pk_destroy( ipcopy_pkt );

  // put back the tcp packet.
```

```
  if (op_pk_nfd_access (orginal_pkt, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
  { printTrace ("modify MH: Unable to get data from TCP Segment to modify tcp pkt.\n");
  }

  tcp_seg_fd_ptr->ack_num = ack_num;

}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
// Generate an ACK packet with rcv_wnd=0, this packet is used to force    //
// the sender into persist mode                                           //
//                                                                        //
////////////////////////////////////////////////////////////////////////////
static Packet * generate_persist_ACK( )
{
       Packet *                          seg_ptr;
    TcpT_Seg_Fields * tcp_seg_fd_ptr;

       // duplicate the sample ACK packet
       seg_ptr = op_pk_copy ( sample_MH_pkt );

    // get the tcp control fields
       if (op_pk_nfd_access (seg_ptr, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
    { printTrace ("Unable to get data from TCP Segment to get persist tcp pkt.\n");
    }

       // set the tcp control fields and make sure rcv_win=0
    tcp_seg_fd_ptr->ack_num = fh_last_seq_num;
       tcp_seg_fd_ptr->seq_num = mh_last_seq_num ;
       tcp_seg_fd_ptr->rcv_win = 0;

       // for debug
    printTrace( "structure of persist ACK pkt is:");
    tcp_seg_fdstruct_print(tcp_seg_fd_ptr);


       return seg_ptr;
}


/***************************************************************************/
/*                                                                         */
/* This is the modified version of ip_encap to encap a tcp pkt into ip pkt.  */
/* The original ip_encap is for general purpose and it can encap TCP, UDP,   */
/* RSVP etc protocol packets. Since we only need tcp for our purpose, we     */
/* simplfy this process here.                                              */
/*                                                                         */
/***************************************************************************/
Packet * myEncap( Packet * pkptr)
{   Packet * ip_pkptr;
       IpT_Dgram_Fields*     ip_dgram_fd_ptr;
       int                   data_len;
       char    error_string[500];

                             /* Obtain the length of the encapsulated packet (in bytes)  */
                             /* (Note: up to 7 bits may be unmodeled for packets which   */
                             /* do not contain an integral number of bytes.              */
                             data_len = op_pk_total_size_get (pkptr) / 8;

                             /* Create an IP packet and encapsulate the new arrival      */
                             /* within its data field. The data field has a modeled size */
                             /* of zero in order to allow breaking of the packet into    */
                             /* pre-determined arbitrarily small sizes. The bulk size    */
                             /* attribute of the IP packet will instead be used to model */
                             /* the size of the encapsulated data.                       */
                             ip_pkptr = op_pk_create_fmt ("ip_dgram_v4");
                             if (ip_pkptr == OPC_NIL)
                                    printTrace("Unable to create IP datagram.");

                             if (op_pk_nfd_set (ip_pkptr, "data", pkptr) ==
OPC_COMPCODE_FAILURE)
```

```
                                    printTrace ("Unable to set data in IP datagram.");

                                /* set protocol to be tcp
                                                */
                                protocol_type = IpC_Protocol_Tcp; // tcp protocol

                                /* In case of an invalid protocol number, generate error.   */
                                if (protocol_type == IpC_Protocol_Unspec)
                                    {
                                    sprintf (error_string, "Simulation aborted; error in
object (%d)", op_id_self ());
                                    op_sim_end (error_string, "ip_encap: protocol is invalid
or unregistered", " ", " ");
                                    }

                                /* Set the bulk size of the IP packet to model the space    */
                                /* occupied by the encapsulated data.                       */
                                op_pk_bulk_size_set (ip_pkptr, data_len * 8);


                                /*      Create fields data structure that contains orig_len, */
                                /*      ident, frag_len, ttl, src_addr, dest_addr, frag,     */
                                /*      connection class, src and dest internal addresses.   */
                                ip_dgram_fd_ptr = ip_dgram_fdstruct_create ();

                                /*      Assign values to members of the field structure.     */
                                ip_dgram_fd_ptr->src_addr                   = MOBILE_HOST_ADDR;
                                ip_dgram_fd_ptr->dest_addr                  = FIX_HOST_ADDR;
                                ip_dgram_fd_ptr->connection_class    = 0;
                                ip_dgram_fd_ptr->orig_len                   = data_len;
                                ip_dgram_fd_ptr->frag_len                   = data_len;
                                ip_dgram_fd_ptr->ttl                        = 32;
                                ip_dgram_fd_ptr->protocol                   = protocol_type;
                                ip_dgram_fd_ptr->tos                        = 0;
                                ip_dgram_fd_ptr->compression_method = IpC_No_Compression;
                                ip_dgram_fd_ptr->original_size      = 160 + op_pk_total_size_get
(ip_pkptr);

                                /* Indicate that the packet is not yet fragmented.   */
                                ip_dgram_fd_ptr->frag                       = 0;

                                /*      Set the fields structure inside the ip datagram.     */
                                op_pk_nfd_set (ip_pkptr, "fields", ip_dgram_fd_ptr,
ip_dgram_fdstruct_copy, ip_dgram_fdstruct_destroy, sizeof (IpT_Dgram_Fields));



    return ip_pkptr;
}


/*************************************************************************/
/*                                                                       */
/* This function handle all the packets from Fix Host.                   */
/* For each arriving packet, do the following:                           */
/*   a) put it into a queue,                                             */
/*   b) record its seq_num and data_len                                  */
/*   c) schedule a timeout intrrupt                                      */
/*   d) forward it to Mobile Host, so that it "thinks" the pkt is from FH   */
/*                                                                       */
/*************************************************************************/
static void handle_Fix_Host_Pkt( Packet * ip_pkptr )
{ Packet * copy_pkptr;
  Packet * copy_pkptr2;
  Packet * tcp_pkptr;

  TcpT_Seg_Fields *    tcp_seg_fd_ptr;

  copy_pkptr = op_pk_copy(ip_pkptr );   // for extracting tcp info
  copy_pkptr2 = op_pk_copy(ip_pkptr );  // to put into queue

  if (op_pk_nfd_get (copy_pkptr, "data", &tcp_pkptr) == OPC_COMPCODE_FAILURE)
  { printTrace ("Unable to get tcp pkt from IP datagram.\n");
```

```
    return ;
  }


  if (op_pk_nfd_get (tcp_pkptr, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
  {    printTrace ("Unable to get fields from TCP Segment.\n");
    return;
  }

  // for debug
  tcp_seg_fdstruct_print(tcp_seg_fd_ptr);

  // if this is a new packet, we
  // mark down the last seq number from FH
  if ( fh_last_seq_num < tcp_seg_fd_ptr->seq_num + tcp_seg_fd_ptr->data_len )
   {
     fh_last_seq_num = tcp_seg_fd_ptr->seq_num + tcp_seg_fd_ptr->data_len;

         // put this pkt into Queue
         putInQueue( copy_pkptr2, tcp_seg_fd_ptr->seq_num, tcp_seg_fd_ptr->data_len );

         // schedule a self interrupt to detect MH disconnection
         op_intrpt_schedule_self( op_sim_time() + MH_rto, 0 );

         // for debug
         printf("After putting into queue, queue size is [%d], head=[%d], tail=[%d]\n",
            queueSize(), queueHead, queueTail );

       }


  // if this is a new ack_num, update our record
  if ( fh_last_ack_num < tcp_seg_fd_ptr->ack_num)
    fh_last_ack_num = tcp_seg_fd_ptr->ack_num;


  printf("MTCP: receive server pkt seq:[%d], ack[%d]\n", fh_last_seq_num, fh_last_ack_num );

  printTrace("MTCP to re-send server pk to ip_encap");
  op_pk_send(ip_pkptr, TO_ENCAP_STRM );       // SEND PK to ip_encap
}


/*************************************************************************/
/*                                                                       */
/*  This function handle all the packets from Mobile Host.               */
/*  For each arriving packet, do the following:                          */
/*   a) get the ack_num from the pkt, record this number                 */
/*   b) record its seq_num and data_len                                  */
/*   c) in the packet, set ack_num = ack_num -1                          */
/*   d) forward it to Fix Host, so that it "thinks" the packet is from MH*/
/*                                                                       */
/*************************************************************************/
static void handle_Mobile_Host_Pkt( Packet * ip_pkptr )
{ Packet * copy_pkptr;
  Packet * copy_pkptr2;
  Packet * tcp_pkptr;
  Packet * modified_pkptr;

  TcpT_Seg_Fields *    tcp_seg_fd_ptr;

  copy_pkptr = op_pk_copy(ip_pkptr );   // for extracting tcp info

  // get tcp packet from ip packet
  if (op_pk_nfd_get (copy_pkptr, "data", &tcp_pkptr) == OPC_COMPCODE_FAILURE)
  { printTrace ("Unable to get tcp pkt from IP datagram.\n");
    return ;
  }


  // get control fields from tcp packet
  if (op_pk_nfd_get (tcp_pkptr, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
  {    printTrace ("Unable to get fields from TCP Segment.\n");
```

```
    return;
  }

  // for debug
  tcp_seg_fdstruct_print(tcp_seg_fd_ptr);

  // if this is a new seq_num, record this new number
  if ( mh_last_seq_num < tcp_seg_fd_ptr->seq_num )
    mh_last_seq_num = tcp_seg_fd_ptr->seq_num;


  // if this is a new ack
  if ( mh_last_ack_num < tcp_seg_fd_ptr->ack_num)
    mh_last_ack_num = tcp_seg_fd_ptr->ack_num;


  /* adjust the RTO of MH, currently not fully implemented */
  /* This part doesn't work now, we set it as 0.6 for the MH_rto  */

  // MH_rto =  calculateRTO ( mh_last_ack_num, op_sim_time() );


  // remove ACKed packets from queue
  purgeFromQueue( mh_last_ack_num );


  // for debug
  printf("After purging, queue size is [%d], head=[%d], tail=[%d]\n",
           queueSize(), queueHead, queueTail );


  // Obtain the sample ACK pkt from mobile host in order to duplicate ACK pkt later
  // This only need to be done once at the very beginning of the connection
  if (  !isInit )            // not init for client pkt setup
   if ( !tcp_seg_fd_ptr->syn ) // not SYN pkt
        if ( tcp_seg_fd_ptr->data_len == 0 )  // double check
   { printTrace( "init client sample pkt" );
     isInit=1;

        // get tcp packet from duplicated packet.
        copy_pkptr2 = op_pk_copy( ip_pkptr );
     if (op_pk_nfd_get (copy_pkptr2, "data", &tcp_pkptr) == OPC_COMPCODE_FAILURE)
         { printTrace ("Unable to get tcp pkt from IP datagram again.\n");
              return ;
              }

     sample_MH_pkt = op_pk_copy ( tcp_pkptr );  // make a sample of MH tcp pkt

     op_pk_destroy( copy_pkptr2 );
       }


  // for dubug
  printf("MTCP: receive client pkt seq:[%d], ack[%d]\n", mh_last_seq_num, mh_last_ack_num );
  printTrace("MTCP to re-send Mobile Host pk to ip_encap");


  // check to see if the link is broken
  if ( isBroken() )
    {
        // first time in persist mode
        if ( transMode == 0 )
    {
          // this is the flag indicating the transition into persist mode
          transMode = 1;

          // which cycle do we start the persist mode, cycle refer to brokenBeginCycle and
brokenEndCycle
          transBeginCycle=cycle;

          // for debug
       fprintf( stderr, "send ACK for persist mode at %lf*********\n", op_sim_time() );
          printTrace("send ACK for persist mode");
```

```
            // generate persist mode packet
                Packet* persist_ACK = generate_persist_ACK();

          // send persist mode packet
                op_pk_send( myEncap( persist_ACK), TO_ENCAP_STRM );
            }
            else printTrace("pk not send for persist mode" );
          }
    else
          {
            // not in persist mode
            transMode=0;

            // reduce the ACK number by 1
            modify_MH_pkt( ip_pkptr, mh_last_ack_num-1 );

            // send the modified  ACK packet to FH
        op_pk_send(ip_pkptr, TO_ENCAP_STRM );
      }
}



/*****************************************************************************/
/*                                                                           */
/*   Main dispatcher of handling packets                                     */
/*   packets will be dispatched to proper handling functions                 */
/*                                                                           */
/*****************************************************************************/
static void myTest2( Packet * ip_pkptr)
{
   // get info from pkt
   analyse_incoming_pkt( ip_pkptr );

   // handle fix host packet
   if ( isFromFixHost() )
          handle_Fix_Host_Pkt ( ip_pkptr );

   // handle mobile host pacekt
   if ( isFromMobileHost() )
          handle_Mobile_Host_Pkt( ip_pkptr );


}


/* 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 mtcp_kernel (void);
       Compcode mtcp_kernel_init (void **);
       void mtcp_kernel_diag (void);
       void mtcp_kernel_terminate (void);
       void mtcp_kernel_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif
```

```
/* Process model interrupt handling procedure */


void
mtcp_kernel (void)
	{
	int _block_origin = 0;
	FIN (mtcp_kernel ());
	if (1)
		{


		FSM_ENTER (mtcp_kernel)

		FSM_BLOCK_SWITCH
			{
			/*---------------------------------------------------------*/
			/** state (INIT) enter executives **/
			FSM_STATE_ENTER_FORCED_NOLABEL (0, "INIT", "mtcp_kernel () [INIT enter
execs]")
				{
				}


			/** state (INIT) exit executives **/
			FSM_STATE_EXIT_FORCED (0, "INIT", "mtcp_kernel () [INIT exit execs]")
				{
				}


			/** state (INIT) transition processing **/
			FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "INIT",
"Wait")
				/*---------------------------------------------------------*/



			/** state (Wait) enter executives **/
			FSM_STATE_ENTER_UNFORCED (1, state1_enter_exec, "Wait", "mtcp_kernel ()
[Wait enter execs]")
				{
				//printTrace("entering MTCP Wait..");
				}


			/** blocking after enter executives of unforced state. **/
			FSM_EXIT (3,mtcp_kernel)


			/** state (Wait) exit executives **/
			FSM_STATE_EXIT_UNFORCED (1, "Wait", "mtcp_kernel () [Wait exit execs]")
				{
				//printTrace( "exit MTCP wait");
				}


			/** state (Wait) transition processing **/
			FSM_INIT_COND (PK_ARRIVE)
			FSM_TEST_COND (DEFAULT)
			FSM_TEST_COND (TIME_OUT)
			FSM_TEST_LOGIC ("Wait")

			FSM_TRANSIT_SWITCH
				{
				FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;, "PK_ARRIVE", "",
"Wait", "handle")
				FSM_CASE_TRANSIT (1, 1, state1_enter_exec, ;, "DEFAULT", "",
"Wait", "Wait")
				FSM_CASE_TRANSIT (2, 3, state3_enter_exec, ;, "TIME_OUT", "",
"Wait", "time_out")
				}
				/*---------------------------------------------------------*/
```

```
                        /** state (handle) enter executives **/
                        FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "handle", "mtcp_kernel ()
[handle enter execs]")
                                {
                                printTrace( "MTCP get pk from ip_encap ");

                                // obtain incomming packet
                                pkptr=op_pk_get(FROM_ENCAP_STRM); // get pk from ip_encap
                                if ( pkptr==OPC_NIL )
                                 puts("empty pkt from ip_encap " );

                                // process the packet
                                myTest2( pkptr );

                                printTrace("MTCP send pk to ip_encap");
                                }


                        /** state (handle) exit executives **/
                        FSM_STATE_EXIT_FORCED (2, "handle", "mtcp_kernel () [handle exit execs]")
                                {
                                }


                        /** state (handle) transition processing **/
                        FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "handle",
"Wait")
                                /*---------------------------------------------------------*/



                        /** state (time_out) enter executives **/
                        FSM_STATE_ENTER_FORCED (3, state3_enter_exec, "time_out", "mtcp_kernel ()
[time_out enter execs]")
                                {


                                /* send packet to force FH into persist mode                    */
                                /* This state occur when there is time out for packets we send */
                                if ( op_intrpt_code() == MH_DISCONNECT_TIME_OUT )
                                 {
                                    // MH is disconnected this moment
                                       printTrace("send ACK for persist mode");

                                       /* Generate persist mode AcK pkt, which is rcv_wind=0  */
                                       Packet* persist_ACK = generate_persist_ACK();

                                       // force FH into persist mode
                                       op_pk_send( myEncap( persist_ACK), TO_ENCAP_STRM );
                                  }




                                }


                        /** state (time_out) exit executives **/
                        FSM_STATE_EXIT_FORCED (3, "time_out", "mtcp_kernel () [time_out exit
execs]")
                                {
                                }


                        /** state (time_out) transition processing **/
                        FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "time_out",
"Wait")
                                /*---------------------------------------------------------*/
```

```
                        }


                FSM_EXIT (0,mtcp_kernel)
                }
        }

#if defined (__cplusplus)
        extern "C" {
#endif
        extern VosT_Fun_Status Vos_Catmem_Register (const char * , int , VosT_Void_Null_Proc,
VosT_Address *);
        extern VosT_Address Vos_Catmem_Alloc (VosT_Address, size_t);
        extern VosT_Fun_Status Vos_Catmem_Dealloc (VosT_Address);
#if defined (__cplusplus)
        }
#endif


Compcode
mtcp_kernel_init (void ** gen_state_pptr)
        {
        int _block_origin = 0;
        static VosT_Address   obtype = OPC_NIL;

        FIN (mtcp_kernel_init (gen_state_pptr))

        if (obtype == OPC_NIL)
                {
                /* Initialize memory management */
                if (Vos_Catmem_Register ("proc state vars (mtcp_kernel)",
                        sizeof (mtcp_kernel_state), Vos_Vnop, &obtype) == VOSC_FAILURE)
                        {
                        FRET (OPC_COMPCODE_FAILURE)
                        }
                }

        *gen_state_pptr = Vos_Catmem_Alloc (obtype, 1);
        if (*gen_state_pptr == OPC_NIL)
                {
                FRET (OPC_COMPCODE_FAILURE)
                }
        else
                {
                /* Initialize FSM handling */
                ((mtcp_kernel_state *)(*gen_state_pptr))->current_block = 0;

                FRET (OPC_COMPCODE_SUCCESS)
                }
        }



void
mtcp_kernel_diag (void)
        {
        /* No Diagnostic Block */
        }




void
mtcp_kernel_terminate (void)
        {
        int _block_origin = __LINE__;

        FIN (mtcp_kernel_terminate (void))

        Vos_Catmem_Dealloc (pr_state_ptr);

        FOUT;
        }
```

```
void
mtcp_kernel_svar (void * gen_ptr, const char * var_name, char ** var_p_ptr)
	{

	FIN (mtcp_kernel_svar (gen_ptr, var_name, var_p_ptr))

	*var_p_ptr = (char *)OPC_NIL;

	FOUT;
	}
```

IP_ENCAP in SH:

```
/* Process model C form file: mtcp_kernel.pr.c */
/* Portions of this file copyright 1992-2001 by OPNET Technologies, Inc. */



/* This variable carries the header into the object file */
static const char mtcp_kernel_pr_c [] = "MIL_3_Tfile_Hdr_ 80C 30A opnet 7 3CBA8F4E 3CBA8F4E 1
payette mjzhan 0 0 none none 0 0 none 0 0 0 0 0 0
";
#include <string.h>



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

#if defined (__cplusplus)
extern "C" {
#endif
FSM_EXT_DECS
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif


/* Header Block */

#include <opnet.h>

/* Header Block */

#include <ip_addr_v4.h>
#include <oms_dt.h>
#include <tcp_api_v3.h>
#include <tcp_v3.h>
#include <oms_pr.h>
#include <oms_qm.h>
#include <oms_tan.h>
#include <tcp_seg_sup.h>
#include <ip_notif_log_support.h>
#include <ip_rte_v4.h>
#include <ip_higher_layer_proto_reg_sup.h>
#include <tcp_seg_sup.h>

#include "/cs/grad1/mjzhan/op_models/msg.h"

static Ici * iciptr;
static Packet * pkptr;

#define PK_ARRIVE      ( op_intrpt_type() ==  OPC_INTRPT_STRM )
#define TIME_OUT       ( (op_intrpt_type() == OPC_INTRPT_REMOTE )|| (op_intrpt_type() ==
OPC_INTRPT_SELF ))
#define DEFAULT        ( (op_intrpt_type() != OPC_INTRPT_STRM) && (op_intrpt_type() !=
OPC_INTRPT_REMOTE ) && (op_intrpt_type() != OPC_INTRPT_SELF) )
```

```
#define FROM_ENCAP_STRM     0
#define TO_ENCAP_STRM     0


#define FIX_HOST_ADDR     0xc0000009
#define MOBILE_HOST_ADDR  0xc0000101

static int isInit =0;

static unsigned fix_host_port ;
static unsigned mobile_host_port;

static int      fh_data_len;
unsigned fh_last_seq_num;
unsigned fh_last_ack_num;

// the ack from Mobile Host, seen by router, which is
// ack seen by Fix host plus 1
unsigned mh_last_ack_num;
unsigned mh_last_seq_num;

// shared variables for incoming ip pkt
static  IpT_Address                  org_addr;
static  IpT_Address                  dest_addr;
static  int                                 type_of_service;
static   int                         protocol_type ;

// to store a sample of the Mobile host packet for later use
Packet * sample_MH_pkt;


int transMode=0;
long transBeginCycle=0;

// rto for MH
double MH_rto=0.6;

#define MH_DISCONNECT_TIME_OUT  100

extern long cycle;
extern int isBrokenPeek(void );



/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef  BIN
#undef  BOUT
#define BIN            FIN_LOCAL_FIELD(last_line_passed) = __LINE__ - _block_origin;
#define BOUT    BIN
#define BINIT   FIN_LOCAL_FIELD(last_line_passed) = 0; _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
       } mtcp_kernel_state;

#define pr_state_ptr                        ((mtcp_kernel_state*) SimI_Mod_State_Ptr)

/* This macro definition 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
#define FIN_PREAMBLE   mtcp_kernel_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };
typedef struct
{
 Packet * pkt;

 unsigned seq_num;
 unsigned data_len;
 double   arr_time;
 double   time_out;

} QueueItem ;

#define MAXQUEUESIZE 5000

QueueItem queue[MAXQUEUESIZE];

int queueHead=0; /* point to first empty item */
int queueTail=0; /* point to last item */

int nextQueueIndex( int n )
{
  if ( n==MAXQUEUESIZE-1 ) n=0;
   else n++;

  return n;
}

int isQueueEmpty()
{ return ( queueHead == queueTail );
}

int hasReachedEnd( int n )
{ return ( n == queueTail );
}


void putInQueue( Packet * pkt, unsigned seq_num, unsigned  data_len )
{
       if ( nextQueueIndex(queueTail) != queueHead )  /* the queue is not full  */
       { queueTail=nextQueueIndex(queueTail);

         queue[queueTail].pkt = pkt;
         queue[queueTail].seq_num = seq_num;
         queue[queueTail].data_len = data_len;
         queue[queueTail].arr_time = op_sim_time();
       }
}

// purge packets that have been ACKed by lastAck number
void purgeFromQueue( unsigned lastAck )
{
  int nextItem ;

  //  nothing to purge
  while ( !isQueueEmpty() )
  {
    nextItem = nextQueueIndex( queueHead );

       // purge ACKed packet(s)
    if ( queue[nextItem].seq_num + queue[nextItem].data_len <= lastAck )
       { queueHead = nextItem;
         op_pk_destroy( queue[nextItem].pkt );
       }
       else
         break;
  } // while
```

```
}


// re-send all packets that hasn't reached the destination
void resendPackets(  )
{ int first;
  int n=0;
  Packet * pkt;

  if ( isQueueEmpty () ) return ;

  first = queueHead ;

  while ( !hasReachedEnd( first) )
  { first = nextQueueIndex( queueHead );

    pkt = op_pk_copy( queue[first].pkt );
    op_pk_send( pkt, TO_ENCAP_STRM ); // sequentially re-send pkts
        n++;
  }
}

int queueSize()
{ int size = queueTail - queueHead;
  if ( size<0 ) size += MAXQUEUESIZE ;

  return size;
}
//---------- end of queue functions  ---------

static int isFromFixHost()
{ return  (org_addr == FIX_HOST_ADDR ) ;
}

static int isFromMobileHost()
{ return  (org_addr == MOBILE_HOST_ADDR ) ;
}


// for testing
extern int brokenStartTime;
extern int brokenEndTime;
extern int cycleTime;
extern long cycle;

// this is for debug
// test if the link is broken
static int isBroken()
{
  double t=op_sim_time();
  long tt =  t;
  cycle=tt%cycleTime;

  if ( (cycle>brokenStartTime) && (cycle<brokenEndTime ))
          return 1;

  return 0;
}



/* This function re-calculate the RTO based on the arrival time of ACK packet */
/* When an ACK comes in, we get the send time of the packet from the queue and*/
/* decide the RTT of it                                                       */
static double calculateRTO( unsigned ack_num, double arrive_time )
{
  int nextItem =queueHead;
  double rtt;


  //  try to find the paket in queue
  while ( !hasReachedEnd( nextItem) )
  {
```

```
    nextItem = nextQueueIndex( queueHead );

        // find the ACKed packet(s)
    if ( queue[nextItem].seq_num + queue[nextItem].data_len == ack_num )
        { // found
          rtt = arrive_time - queue[nextItem].arr_time;
          break;
        }
  } // while


  /* calculate RTO base on RTT   */

  /* DOES NOT WORK NOW */

  /* 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;


     return 0.45;   // we return the fix RTO at this moment
}


// analyse an imcoming pacekt, record the src/dest address,
// type of service, and protocol
static void analyse_incoming_pkt( Packet * ip_pkptr )
{   IpT_Dgram_Fields* ip_dgram_fd_ptr;

    /*  Get the fields structure from the packet.                   */
        op_pk_nfd_access (ip_pkptr, "fields", &ip_dgram_fd_ptr);

        /* Determine the source address of the data.                */
        org_addr = ip_dgram_fd_ptr->src_addr;

        /* Determine the destination address of the data.           */
        dest_addr = ip_dgram_fd_ptr->dest_addr;

        /* Determine the type of service. This attribute needs to be passed to tcp        */
        /* layer (for tcp connectios),^^or to tpal (for udp connections).                 */
        type_of_service = ip_dgram_fd_ptr->tos;

        /* Determine the protocol number.                                          */
        protocol_type = ip_dgram_fd_ptr->protocol;

}


//////////////////////////////////////////////////////////////////////////////
//                                                                          //
// modify ip pkt from MH, change ack_num  to the value of the input parameter, //
// other fields are untouched                                               //
//                                                                          //
//////////////////////////////////////////////////////////////////////////////
static void modify_MH_pkt( Packet * ip_pkptr, unsigned ack_num )
{ Packet * orginal_pkt;
  Packet * ipcopy_pkt;

  TcpT_Seg_Fields *    tcp_seg_fd_ptr;


  // copy a new instance of the packet
  ipcopy_pkt = op_pk_copy( ip_pkptr );
```

```
  // extract the tcp packet out
  if (op_pk_nfd_get (ipcopy_pkt, "data", &orginal_pkt) == OPC_COMPCODE_FAILURE)
  { printTrace ("modify MH: Unable to get tcp pkt from IP datagram again.\n");
        return ;
        }

  // delete this packet
  op_pk_destroy( ipcopy_pkt );

  // put back the tcp packet.
  if (op_pk_nfd_access (orginal_pkt, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
  { printTrace ("modify MH: Unable to get data from TCP Segment to modify tcp pkt.\n");
  }

  tcp_seg_fd_ptr->ack_num = ack_num;

}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
// Generate an ACK packet with rcv_wnd=0, this packet is used to force    //
// the sender into persist mode                                           //
//                                                                        //
////////////////////////////////////////////////////////////////////////////
static Packet * generate_persist_ACK( )
{
       Packet *                          seg_ptr;
    TcpT_Seg_Fields * tcp_seg_fd_ptr;

       // duplicate the sample ACK packet
       seg_ptr = op_pk_copy ( sample_MH_pkt );

    // get the tcp control fields
       if (op_pk_nfd_access (seg_ptr, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
    { printTrace ("Unable to get data from TCP Segment to get persist tcp pkt.\n");
    }

       // set the tcp control fields and make sure rcv_win=0
    tcp_seg_fd_ptr->ack_num = fh_last_seq_num;
       tcp_seg_fd_ptr->seq_num = mh_last_seq_num ;
       tcp_seg_fd_ptr->rcv_win = 0;

       // for debug
    printTrace( "structure of persist ACK pkt is:");
    tcp_seg_fdstruct_print(tcp_seg_fd_ptr);


       return seg_ptr;
}


/***************************************************************************/
/*                                                                         */
/* This is the modified version of ip_encap to encap a tcp pkt into ip pkt.  */
/* The original ip_encap is for general purpose and it can encap TCP, UDP,   */
/* RSVP etc protocol packets. Since we only need tcp for our purpose, we     */
/* simplfy this process here.                                              */
/*                                                                         */
/***************************************************************************/
Packet * myEncap( Packet * pkptr)
{   Packet * ip_pkptr;
       IpT_Dgram_Fields*     ip_dgram_fd_ptr;
       int                   data_len;
       char    error_string[500];

                             /* Obtain the length of the encapsulated packet (in bytes)  */
                             /* (Note: up to 7 bits may be unmodeled for packets which   */
                             /* do not contain an integral number of bytes.
              */
                             data_len = op_pk_total_size_get (pkptr) / 8;
```

```
                            /* Create an IP packet and encapsulate the new arrival
        */
                            /* within its data field. The data field has a modeled size */
                            /* of zero in order to allow breaking of the packet into    */
                            /* pre-determined arbitrarily small sizes. The bulk size    */
                            /* attribute of the IP packet will instead be used to model */
                            /* the size of the encapsulated data.
                */
                            ip_pkptr = op_pk_create_fmt ("ip_dgram_v4");
                            if (ip_pkptr == OPC_NIL)
                                    printTrace("Unable to create IP datagram.");

                            if (op_pk_nfd_set (ip_pkptr, "data", pkptr) ==
OPC_COMPCODE_FAILURE)
                                    printTrace ("Unable to set data in IP datagram.");

                            /* set protocol to be tcp
                                        */
                            protocol_type = IpC_Protocol_Tcp; // tcp protocol

                            /* In case of an invalid protocol number, generate error.   */
                            if (protocol_type == IpC_Protocol_Unspec)
                                    {
                                    sprintf (error_string, "Simulation aborted; error in
object (%d)", op_id_self ());
                                    op_sim_end (error_string, "ip_encap: protocol is invalid
or unregistered", " ", " ");
                                    }

                            /* Set the bulk size of the IP packet to model the space    */
                            /* occupied by the encapsulated data.
                */
                            op_pk_bulk_size_set (ip_pkptr, data_len * 8);


                            /*      Create fields data structure that contains orig_len,
        */
                            /*      ident, frag_len, ttl, src_addr, dest_addr, frag,
        */
                            /*      connection class, src and dest internal addresses.
        */
                            ip_dgram_fd_ptr = ip_dgram_fdstruct_create ();

                            /*      Assign values to members of the field structure.     */
                            ip_dgram_fd_ptr->src_addr                   = MOBILE_HOST_ADDR;
                            ip_dgram_fd_ptr->dest_addr                  = FIX_HOST_ADDR;
                            ip_dgram_fd_ptr->connection_class    = 0;
                            ip_dgram_fd_ptr->orig_len                   = data_len;
                            ip_dgram_fd_ptr->frag_len                   = data_len;
                            ip_dgram_fd_ptr->ttl                        = 32;
                            ip_dgram_fd_ptr->protocol                   = protocol_type;
                            ip_dgram_fd_ptr->tos                        = 0;
                            ip_dgram_fd_ptr->compression_method = IpC_No_Compression;
                            ip_dgram_fd_ptr->original_size      = 160 + op_pk_total_size_get
(ip_pkptr);

                            /* Indicate that the packet is not yet fragmented.   */
                            ip_dgram_fd_ptr->frag                       = 0;

                            /*      Set the fields structure inside the ip datagram.     */
                            op_pk_nfd_set (ip_pkptr, "fields", ip_dgram_fd_ptr,
ip_dgram_fdstruct_copy, ip_dgram_fdstruct_destroy, sizeof (IpT_Dgram_Fields));



    return ip_pkptr;
}


/*****************************************************************************/
/*                                                                           */
/* This function handle all the packets from Fix Host.                       */
/* For each arriving packet, do the following:                               */
```

```
/*   a) put it into a queue,                                                */
/*   b) record its seq_num and data_len                                     */
/*   c) schedule a timeout intrrupt                                         */
/*   d) forward it to Mobile Host, so that it "thinks" the pkt is from FH   */
/*                                                                          */
/****************************************************************************/
static void handle_Fix_Host_Pkt( Packet * ip_pkptr )
{ Packet * copy_pkptr;
  Packet * copy_pkptr2;
  Packet * tcp_pkptr;

  TcpT_Seg_Fields *    tcp_seg_fd_ptr;

  copy_pkptr = op_pk_copy(ip_pkptr );   // for extracting tcp info
  copy_pkptr2 = op_pk_copy(ip_pkptr );  // to put into queue

  if (op_pk_nfd_get (copy_pkptr, "data", &tcp_pkptr) == OPC_COMPCODE_FAILURE)
  { printTrace ("Unable to get tcp pkt from IP datagram.\n");
    return ;
  }


  if (op_pk_nfd_get (tcp_pkptr, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
  {     printTrace ("Unable to get fields from TCP Segment.\n");
    return;
  }

  // for debug
  tcp_seg_fdstruct_print(tcp_seg_fd_ptr);

  // if this is a new packet, we
  // mark down the last seq number from FH
  if ( fh_last_seq_num < tcp_seg_fd_ptr->seq_num + tcp_seg_fd_ptr->data_len )
   {
     fh_last_seq_num = tcp_seg_fd_ptr->seq_num + tcp_seg_fd_ptr->data_len;

         // put this pkt into Queue
         putInQueue( copy_pkptr2, tcp_seg_fd_ptr->seq_num, tcp_seg_fd_ptr->data_len );

         // schedule a self interrupt to detect MH disconnection
         op_intrpt_schedule_self( op_sim_time() + MH_rto, 0 );

         // for debug
         printf("After putting into queue, queue size is [%d], head=[%d], tail=[%d]\n",
            queueSize(), queueHead, queueTail );

        }


  // if this is a new ack_num, update our record
  if ( fh_last_ack_num < tcp_seg_fd_ptr->ack_num)
    fh_last_ack_num = tcp_seg_fd_ptr->ack_num;


  printf("MTCP: receive server pkt seq:[%d], ack[%d]\n", fh_last_seq_num, fh_last_ack_num );

  printTrace("MTCP to re-send server pk to ip_encap");
  op_pk_send(ip_pkptr, TO_ENCAP_STRM );       // SEND PK to ip_encap
}


/****************************************************************************/
/*                                                                          */
/*  This function handle all the packets from Mobile Host.                  */
/*  For each arriving packet, do the following:                             */
/*   a) get the ack_num from the pkt, record this number                    */
/*   b) record its seq_num and data_len                                     */
/*   c) in the packet, set ack_num = ack_num -1                             */
/*   d) forward it to Fix Host, so that it "thinks" the packet is from MH   */
/*                                                                          */
/****************************************************************************/
static void handle_Mobile_Host_Pkt( Packet * ip_pkptr )
{ Packet * copy_pkptr;
```

```
Packet * copy_pkptr2;
Packet * tcp_pkptr;
Packet * modified_pkptr;

TcpT_Seg_Fields *    tcp_seg_fd_ptr;

copy_pkptr = op_pk_copy(ip_pkptr );   // for extracting tcp info

// get tcp packet from ip packet
if (op_pk_nfd_get (copy_pkptr, "data", &tcp_pkptr) == OPC_COMPCODE_FAILURE)
{ printTrace ("Unable to get tcp pkt from IP datagram.\n");
  return ;
}


// get control fields from tcp packet
if (op_pk_nfd_get (tcp_pkptr, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
{    printTrace ("Unable to get fields from TCP Segment.\n");
  return;
}

// for debug
tcp_seg_fdstruct_print(tcp_seg_fd_ptr);

// if this is a new seq_num, record this new number
if ( mh_last_seq_num < tcp_seg_fd_ptr->seq_num )
  mh_last_seq_num = tcp_seg_fd_ptr->seq_num;


// if this is a new ack
if ( mh_last_ack_num < tcp_seg_fd_ptr->ack_num)
  mh_last_ack_num = tcp_seg_fd_ptr->ack_num;


/* adjust the RTO of MH, currently not fully implemented */
/* This part doesn't work now, we set it as 0.6 for the MH_rto  */

// MH_rto =  calculateRTO ( mh_last_ack_num, op_sim_time() );


// remove ACKed packets from queue
purgeFromQueue( mh_last_ack_num );


// for debug
printf("After purging, queue size is [%d], head=[%d], tail=[%d]\n",
         queueSize(), queueHead, queueTail );


// Obtain the sample ACK pkt from mobile host in order to duplicate ACK pkt later
// This only need to be done once at the very beginning of the connection
if (  !isInit )            // not init for client pkt setup
 if ( !tcp_seg_fd_ptr->syn ) // not SYN pkt
     if ( tcp_seg_fd_ptr->data_len == 0 )  // double check
 { printTrace( "init client sample pkt" );
   isInit=1;

       // get tcp packet from duplicated packet.
       copy_pkptr2 = op_pk_copy( ip_pkptr );
   if (op_pk_nfd_get (copy_pkptr2, "data", &tcp_pkptr) == OPC_COMPCODE_FAILURE)
        { printTrace ("Unable to get tcp pkt from IP datagram again.\n");
             return ;
             }

   sample_MH_pkt = op_pk_copy ( tcp_pkptr );  // make a sample of MH tcp pkt

   op_pk_destroy( copy_pkptr2 );
     }


// for dubug
printf("MTCP: receive client pkt seq:[%d], ack[%d]\n", mh_last_seq_num, mh_last_ack_num );
printTrace("MTCP to re-send Mobile Host pk to ip_encap");
```

```
  // check to see if the link is broken
  if ( isBroken() )
    {
         // first time in persist mode
         if ( transMode == 0 )
     {
           // this is the flag indicating the transition into persist mode
           transMode = 1;

           // which cycle do we start the persist mode, cycle refer to brokenBeginCycle and
brokenEndCycle
           transBeginCycle=cycle;

           // for debug
       fprintf( stderr, "send ACK for persist mode at %lf*********\n", op_sim_time() );
           printTrace("send ACK for persist mode");

            // generate persist mode packet
                Packet* persist_ACK = generate_persist_ACK();

        // send persist mode packet
                op_pk_send( myEncap( persist_ACK), TO_ENCAP_STRM );
          }
          else printTrace("pk not send for persist mode" );
        }
  else
        {
          // not in persist mode
          transMode=0;

          // reduce the ACK number by 1
          modify_MH_pkt( ip_pkptr, mh_last_ack_num-1 );

          // send the modified  ACK packet to FH
      op_pk_send(ip_pkptr, TO_ENCAP_STRM );
    }
}



// This is a test function of our program, not used now
static Packet * myTest( Packet * ip_pkptr)
{ IpT_Dgram_Fields* ip_dgram_fd_ptr;
       IpT_Address                  arrived_intf_addr;
       int                                 minor_port_received;
       int                          major_port_received;
       IpT_Address                  org_addr;
       IpT_Address                  dest_addr;
       IpT_Address                  next_hop_addr;
       char                         dest_addr_str [IPC_ADDR_STR_LEN];
       int                                 type_of_service;
    int                             protocol_type ;
       TcpT_Seg_Fields*      tcp_seg_fd_ptr;
       TcpT_Seg_Fields*      tcp_seg_fd_ptr2;

    Packet * copy_pkptr;
    Packet * copy_pkptr2;
       Packet * tcp_pkptr;
       Packet * dataPtr;

       char  pkt_format[50];

       /*      Get the fields structure from the packet.                   */
       op_pk_nfd_access (ip_pkptr, "fields", &ip_dgram_fd_ptr);

       /* Determine the source address of the data.               */
       org_addr = ip_dgram_fd_ptr->src_addr;

       /* Determine the destination address of the data.          */
       dest_addr = ip_dgram_fd_ptr->dest_addr;
```

```
        /* Determine the type of service. This attribute needs to be passed to tcp        */
        /* layer (for tcp connectios),^^or to tpal (for udp connections).                 */
        type_of_service = ip_dgram_fd_ptr->tos;

        /* Determine the protocol number.                                          */
        protocol_type = ip_dgram_fd_ptr->protocol;

        printf("MTCP: source addr is %x, dest addr is %x\n", org_addr, dest_addr );
        printf("MTCP: type of service is [%d], protocol is [%d]\n", type_of_service,
protocol_type);


        copy_pkptr = op_pk_copy(ip_pkptr );

        analyse_incoming_pkt( ip_pkptr);


        if (op_pk_nfd_get (copy_pkptr, "data", &tcp_pkptr) == OPC_COMPCODE_FAILURE)
                printf ("MTCP: Unable to get data from IP datagram.\n");

        copy_pkptr2 = op_pk_copy( tcp_pkptr );

 /*   if (op_pk_nfd_is_set (tcp_pkptr, "data") == OPC_TRUE)
          if ( op_pk_nfd_get( tcp_pkptr, "data", &dataPtr ) == OPC_COMPCODE_FAILURE )
                printf( "MTCP: ** Unable to get tcp real data" );
          else
                  printf("MTCP: ** data got");
        else printf("MTCP: ** data not found ");
        */

        if (op_pk_nfd_get (tcp_pkptr, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
                printf ("MTCP: Unable to get data from TCP Segment.\n");


//      tcp_seg_fd_ptr2=tcp_seg_fdstruct_copy( tcp_seg_fd_ptr );

//      tcp_seg_fd_ptr->src_port=111;

//      op_pk_nfd_set ( copy_pkptr2, "fields", tcp_seg_fd_ptr2, tcp_seg_fdstruct_copy,
tcp_seg_fdstruct_destroy, sizeof (TcpT_Seg_Fields));

//      if (op_pk_nfd_get (copy_pkptr2, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
//              printf ("MTCP: Unable to get data from TCP Segment copy 2.\n");

        if (op_pk_nfd_access (copy_pkptr2, "fields", &tcp_seg_fd_ptr) == OPC_COMPCODE_FAILURE)
                printf ("MTCP: Unable to get data from TCP Segment copy 2.\n");

        // to test if we can directly change tcp pkt fields
    tcp_seg_fd_ptr->src_port = 9999;

        if (op_pk_nfd_get (copy_pkptr2, "fields", &tcp_seg_fd_ptr2) == OPC_COMPCODE_FAILURE)
                printf ("MTCP: Unable to get data from TCP Segment copy 22.\n");

        tcp_seg_fdstruct_print(tcp_seg_fd_ptr2);

        //printf("MTCP: from tcp pkt src port is %d, dest port is %d, seq=%d\n",
        //        tcp_seg_fd_ptr->src_port, tcp_seg_fd_ptr->dest_port, tcp_seg_fd_ptr->seq_num
);


        return OPC_NIL;

}


/*************************************************************************/
/*                                                                       */
/*   Main dispatcher of handling packets                                 */
/*   packets will be dispatched to proper handling functions             */
/*                                                                       */
/*************************************************************************/
static void myTest2( Packet * ip_pkptr)
{
```

```
   // get info from pkt
   analyse_incoming_pkt( ip_pkptr );

   // handle fix host packet
   if ( isFromFixHost() )
          handle_Fix_Host_Pkt ( ip_pkptr );

   // handle mobile host pacekt
   if ( isFromMobileHost() )
          handle_Mobile_Host_Pkt( ip_pkptr );


}


/* 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 mtcp_kernel (void);
       Compcode mtcp_kernel_init (void **);
       void mtcp_kernel_diag (void);
       void mtcp_kernel_terminate (void);
       void mtcp_kernel_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif



/* Process model interrupt handling procedure */


void
mtcp_kernel (void)
       {
       int _block_origin = 0;
       FIN (mtcp_kernel ());
       if (1)
              {


              FSM_ENTER (mtcp_kernel)

              FSM_BLOCK_SWITCH
                     {
                     /*---------------------------------------------------------*/
                     /** state (INIT) enter executives **/
                     FSM_STATE_ENTER_FORCED_NOLABEL (0, "INIT", "mtcp_kernel () [INIT enter
execs]")
                            {
                            }


                     /** state (INIT) exit executives **/
                     FSM_STATE_EXIT_FORCED (0, "INIT", "mtcp_kernel () [INIT exit execs]")
                            {
                            }


                     /** state (INIT) transition processing **/
```

```
                        FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "INIT",
"Wait")
                                /*---------------------------------------------------------*/



                        /** state (Wait) enter executives **/
                        FSM_STATE_ENTER_UNFORCED (1, state1_enter_exec, "Wait", "mtcp_kernel ()
[Wait enter execs]")
                                {
                                //printTrace("entering MTCP Wait..");
                                }


                        /** blocking after enter executives of unforced state. **/
                        FSM_EXIT (3,mtcp_kernel)


                        /** state (Wait) exit executives **/
                        FSM_STATE_EXIT_UNFORCED (1, "Wait", "mtcp_kernel () [Wait exit execs]")
                                {
                                //printTrace( "exit MTCP wait");
                                }


                        /** state (Wait) transition processing **/
                        FSM_INIT_COND (PK_ARRIVE)
                        FSM_TEST_COND (DEFAULT)
                        FSM_TEST_COND (TIME_OUT)
                        FSM_TEST_LOGIC ("Wait")

                        FSM_TRANSIT_SWITCH
                                {
                                FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;, "PK_ARRIVE", "",
"Wait", "handle")
                                FSM_CASE_TRANSIT (1, 1, state1_enter_exec, ;, "DEFAULT", "",
"Wait", "Wait")
                                FSM_CASE_TRANSIT (2, 3, state3_enter_exec, ;, "TIME_OUT", "",
"Wait", "time_out")
                                }
                                /*---------------------------------------------------------*/



                        /** state (handle) enter executives **/
                        FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "handle", "mtcp_kernel ()
[handle enter execs]")
                                {
                                printTrace( "MTCP get pk from ip_encap ");

                                // obtain incomming packet
                                pkptr=op_pk_get(FROM_ENCAP_STRM); // get pk from ip_encap
                                if ( pkptr==OPC_NIL )
                                 puts("empty pkt from ip_encap " );

                                // process the packet
                                myTest2( pkptr );

                                printTrace("MTCP send pk to ip_encap");
                                }


                        /** state (handle) exit executives **/
                        FSM_STATE_EXIT_FORCED (2, "handle", "mtcp_kernel () [handle exit execs]")
                                {
                                }


                        /** state (handle) transition processing **/
                        FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "handle",
"Wait")
                                /*---------------------------------------------------------*/
```

```
                        /** state (time_out) enter executives **/
                        FSM_STATE_ENTER_FORCED (3, state3_enter_exec, "time_out", "mtcp_kernel ()
[time_out enter execs]")
                                {


                                /* send packet to force FH into persist mode                    */
                                /* This state occur when there is time out for packets we send */
                                if ( op_intrpt_code() == MH_DISCONNECT_TIME_OUT )
                                 {
                                    // MH is disconnected this moment
                                        printTrace("send ACK for persist mode");

                                        /* Generate persist mode AcK pkt, which is rcv_wind=0  */
                                        Packet* persist_ACK = generate_persist_ACK();

                                        // force FH into persist mode
                                        op_pk_send( myEncap( persist_ACK), TO_ENCAP_STRM );
                                  }




                                }


                        /** state (time_out) exit executives **/
                        FSM_STATE_EXIT_FORCED (3, "time_out", "mtcp_kernel () [time_out exit
execs]")
                                {
                                }


                        /** state (time_out) transition processing **/
                        FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "time_out",
"Wait")
                                /*---------------------------------------------------------*/



                        }


                FSM_EXIT (0,mtcp_kernel)
                }
        }

#if defined (__cplusplus)
        extern "C" {
#endif
        extern VosT_Fun_Status Vos_Catmem_Register (const char * , int , VosT_Void_Null_Proc,
VosT_Address *);
        extern VosT_Address Vos_Catmem_Alloc (VosT_Address, size_t);
        extern VosT_Fun_Status Vos_Catmem_Dealloc (VosT_Address);
#if defined (__cplusplus)
        }
#endif


Compcode
mtcp_kernel_init (void ** gen_state_pptr)
        {
        int _block_origin = 0;
        static VosT_Address   obtype = OPC_NIL;

        FIN (mtcp_kernel_init (gen_state_pptr))

        if (obtype == OPC_NIL)
                {
                /* Initialize memory management */
                if (Vos_Catmem_Register ("proc state vars (mtcp_kernel)",
```

```
                        sizeof (mtcp_kernel_state), Vos_Vnop, &obtype) == VOSC_FAILURE)
                        {
                        FRET (OPC_COMPCODE_FAILURE)
                        }
                }

        *gen_state_pptr = Vos_Catmem_Alloc (obtype, 1);
        if (*gen_state_pptr == OPC_NIL)
                {
                FRET (OPC_COMPCODE_FAILURE)
                }
        else
                {
                /* Initialize FSM handling */
                ((mtcp_kernel_state *)(*gen_state_pptr))->current_block = 0;

                FRET (OPC_COMPCODE_SUCCESS)
                }
        }



void
mtcp_kernel_diag (void)
        {
        /* No Diagnostic Block */
        }




void
mtcp_kernel_terminate (void)
        {
        int _block_origin = __LINE__;

        FIN (mtcp_kernel_terminate (void))

        Vos_Catmem_Dealloc (pr_state_ptr);

        FOUT;
        }




void
mtcp_kernel_svar (void * gen_ptr, const char * var_name, char ** var_p_ptr)
        {

        FIN (mtcp_kernel_svar (gen_ptr, var_name, var_p_ptr))

        *var_p_ptr = (char *)OPC_NIL;

        FOUT;
        }



IP_ENCAP at MH:

/* Process model C form file: YS_disc_ip_encap_v4.pr.c */
/* Portions of this file copyright 1992-2001 by OPNET Technologies, Inc. */



/* This variable carries the header into the object file */
static const char YS_disc_ip_encap_v4_pr_c [] = "MIL_3_Tfile_Hdr_ 80C 30A op_runsim 7 3CB72C18
3CB72C18 1 payette mjzhan 0 0 none none 0 0 none 0 0 0 0 0 0
";
#include <string.h>
```

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

#if defined (__cplusplus)
extern "C" {
#endif
FSM_EXT_DECS
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif


/* Header Block */

#include <ip_addr_v4.h>
#include <oms_pr.h>
#include <oms_tan.h>
#include <ip_dgram_sup.h>
#include <ip_higher_layer_proto_reg_sup.h>
#include <ip_notif_log_support.h>
#include <ip_rte_v4.h>
#include <rsvp.h>
#include <string.h>

#include "/cs/grad1/mjzhan/op_models/msg.h"

#define FROM_TRANSPORT        op_intrpt_strm () != instrm_from_network
#define FROM_NETWORK          op_intrpt_strm () == instrm_from_network

/* Typedefs */
typedef struct
        {
        int                                     protocol_type;
        int                                     outstream;
        int                                     instream;
        }IpT_Encap_Interface;

/* Functions */
static int                                      ip_encap_proto_get (int strm_num, int num_iface,
IpT_Encap_Interface** interface_table_handle);
static void                                     ip_encap_pk_destroy (Packet* pkptr);
static void                                     ip_encap_error (const char* msg);

static int brokenLink =0;

/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef  BIN
#undef  BOUT
#define BIN             FIN_LOCAL_FIELD(last_line_passed) = __LINE__ - _block_origin;
#define BOUT    BIN
#define BINIT   FIN_LOCAL_FIELD(last_line_passed) = 0; _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 */
        IpT_Encap_Interface**                   interface_table;
        int                                     interface_table_size;
        int                                     instrm_from_network;
        int                                     outstrm_to_network;
        Boolean                                 gateway;
        } YS_disc_ip_encap_v4_state;
```

```
#define pr_state_ptr                        ((YS_disc_ip_encap_v4_state*) SimI_Mod_State_Ptr)
#define interface_table                     pr_state_ptr->interface_table
#define interface_table_size                pr_state_ptr->interface_table_size
#define instrm_from_network                 pr_state_ptr->instrm_from_network
#define outstrm_to_network                  pr_state_ptr->outstrm_to_network
#define gateway                             pr_state_ptr->gateway

/* This macro definition 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
#define FIN_PREAMBLE   YS_disc_ip_encap_v4_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };
int brokenEnable;
int cycleTime;
int brokenStartTime;
int brokenEndTime;
long cycle;

int isBroken()
{
  if ( !brokenEnable ) return 0;

  double t=op_sim_time();
  long tt =  t;
  cycle=tt%cycleTime;

  if ( (cycle>brokenStartTime) && (cycle<brokenEndTime ))
          return 1;

  return 0;
}


static int
ip_encap_proto_get (int strm_num, int num_iface, IpT_Encap_Interface** interface_table_handle)
        {
        int                     i;

        /** This function returns the protocol type corresponding to an input stream index. **/
        FIN (ip_encap_proto_get (strm_num, num_iface, interface_table_handle))

        for (i=0; i < num_iface; i++)
                {
                if (interface_table_handle[i]->instream == strm_num)
                        {
                        FRET (interface_table_handle[i]->protocol_type);
                        }
                }

        /* If no match is found, return an error code. */
        FRET (IpC_Protocol_Unspec);
        }

static void
ip_encap_pk_destroy (Packet* pkptr)
        {
        Ici*                                    intf_iciptr;
        IpT_Rte_Ind_Ici_Fields*             intf_ici_struct_ptr;

        /** Destroys the IP datagram received from lower    **/
        /** layer and the associated ip_rte_ind_v4 ICI              **/
        FIN (ip_encap_pk_destory (pkptr));

        /* Get the ICI associated with the packet    */
        intf_iciptr = op_pk_ici_get (pkptr);
```

```
	/* Destroy the ICI and its fields    */
	if (intf_iciptr != OPC_NIL)
		{
		op_ici_attr_get (intf_iciptr, "rte_info_fields", &intf_ici_struct_ptr);
		ip_rte_ind_ici_fdstruct_destroy (intf_ici_struct_ptr);
		op_ici_destroy (intf_iciptr);
		}

	/* Destroy the packet  */
	op_pk_destroy (pkptr);

	FOUT;
	}

static void
ip_encap_error (const char* msg)
	{
	/** Print an error message and exit the simulation. **/
	FIN (ip_encap_error (msg));

	op_sim_end ("Error from IP encapsulation process model (ip_encap_v4):",
				msg, OPC_NIL, OPC_NIL);

	FOUT;
	}


// this is for debug
int isBrokenPeek()
{
  double t=op_sim_time();
  long tt =  t;
  long cycle=tt%cycleTime;

  if ( (cycle>brokenStartTime) && (cycle<brokenEndTime ))
      return 1;

  return 0;
}


// this is for debug
int isBroken1()
{
  if ( !brokenEnable ) return 0;

  double t=op_sim_time();
  long tt =  t;
  long cycle=(tt)%cycleTime;

  if ( (cycle>brokenStartTime) && (cycle<brokenEndTime ))
      return 1;

  if ( ( cycle<120 ) && (tt>122 ) ) return 1;

  return 0;
}


/* 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 YS_disc_ip_encap_v4 (void);
        Compcode YS_disc_ip_encap_v4_init (void **);
        void YS_disc_ip_encap_v4_diag (void);
        void YS_disc_ip_encap_v4_terminate (void);
        void YS_disc_ip_encap_v4_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif



/* Process model interrupt handling procedure */


void
YS_disc_ip_encap_v4 (void)
        {
        int _block_origin = 0;
        FIN (YS_disc_ip_encap_v4 ());
        if (1)
                {
                Packet*                         ip_pkptr;
                Packet*                         pkptr;

                Ici*                            ul_iciptr;
                Ici*                            ll_iciptr;
                Ici*                            ip_iciptr;
                Ici*                            transp_iciptr;

                IpT_Address                     arrived_intf_addr;
                int                                     minor_port_received;
                int                             major_port_received;
                IpT_Address                     org_addr;
                IpT_Address                     dest_addr;
                IpT_Address                     next_hop_addr;
                char                            dest_addr_str [IPC_ADDR_STR_LEN];
                int                                     type_of_service;

                double                          iface_mtu;
                double                          iface_load;
                double                          iface_speed;
                double                          iface_reliability;

                int                                     i;
                int                                     data_len;
                int                                     input_strm, output_strm;
                int                                     num_char;
                int                                     conn_class;
                int                                     mcast_major_port;
                int                                     mcast_minor_port;
                int                             ttl;
                int                                     intf_index;
                char                            protocol_name[20];
                char                            proc_model_name[20];
                char                            error_string [512];

                IpT_Dgram_Fields*       ip_dgram_fd_ptr;
                int                                     protocol_type;

                List*                                   proc_record_handle_list_ptr;
                int                                     record_handle_list_size;
                OmsT_Pr_Handle          process_record_handle;
                Objid                                   own_objid;
                Objid                                   own_node_objid;
                Objid                                   neighbor_mod_objid;
                Prohandle                       own_prohandle;
                char                            gateway_str [32];

                /* IP datagrams maintain an ICI to record information       */
                /* required for proper routing (e.g., slot on which the     */
                /* datagram arrived, output interface on which it           */
                /* should be forwarded, etc.) -- all that information       */
```

```
		/* is stored in a single structure field of the                    */
		/* ip_rte_ind_v4 ICI.
	*/
		IpT_Rte_Ind_Ici_Fields*		intf_ici_fdstruct_ptr;

		/* RSVP packets must be forwarded on interface specified    */
		/* by RSVP. The routing information for RSVP packets is     */
		/* carried in ip_encap_reg ICI and also in ip_rte_reg ICI.  */
		RsvpT_Rte_Ici_Struct *		pkt_route_info_ptr;



		FSM_ENTER (YS_disc_ip_encap_v4)

		FSM_BLOCK_SWITCH
			{
			/*---------------------------------------------------------*/
			/** state (ENCAP) enter executives **/
			FSM_STATE_ENTER_FORCED (0, state0_enter_exec, "ENCAP",
"YS_disc_ip_encap_v4 () [ENCAP enter execs]")
				{
				/* Obtain the packet arriving from a higher protocol layer. */
				//printTrace( "enter ip-encap ENCAP  ...");

				input_strm = op_intrpt_strm();
				pkptr = op_pk_get (input_strm);
				if (pkptr == OPC_NIL)
					ip_encap_error ("Unable to get packet from transport
layer.");

				/* Obtain the length of the encapsulated packet (in bytes)  */
				/* (Note: up to 7 bits may be unmodeled for packets which   */
				/* do not contain an integral number of bytes.
		*/
				data_len = op_pk_total_size_get (pkptr) / 8;

				/* Create an IP packet and encapsulate the new arrival
	*/
				/* within its data field. The data field has a modeled size */
				/* of zero in order to allow breaking of the packet into    */
				/* pre-determined arbitrarily small sizes. The bulk size    */
				/* attribute of the IP packet will instead be used to model */
				/* the size of the encapsulated data.
		*/
				ip_pkptr = op_pk_create_fmt ("ip_dgram_v4");
				if (ip_pkptr == OPC_NIL)
					ip_encap_error ("Unable to create IP datagram.");

				if (op_pk_nfd_set (ip_pkptr, "data", pkptr) ==
OPC_COMPCODE_FAILURE)
					ip_encap_error ("Unable to set data in IP datagram.");

				/* Determine the protocol number corresponding to the input */
				/* stream index.
					*/
				protocol_type = ip_encap_proto_get (input_strm,
interface_table_size, interface_table);

				/* In case of an invalid protocol number, generate error.   */
				if (protocol_type == IpC_Protocol_Unspec)
					{
					sprintf (error_string, "Simulation aborted; error in
object (%d)", op_id_self ());
					op_sim_end (error_string, "ip_encap: protocol is invalid
or unregistered", " ", " ");
					}

				/* Set the bulk size of the IP packet to model the space    */
				/* occupied by the encapsulated data.
		*/
				op_pk_bulk_size_set (ip_pkptr, data_len * 8);
```

```
                        /* Obtain the destination address, source address and
       */
                        /* connection class from the accompanying ICI.
               */
                        ul_iciptr = op_intrpt_ici ();
                        if (ul_iciptr == OPC_NIL)
                                {
                                ip_encap_error ("Unable to get ICI from transport
layer.");
                                }

                        if (op_ici_attr_get (ul_iciptr, "dest_addr", &dest_addr) ==
OPC_COMPCODE_FAILURE)
                                {
                                ip_encap_error ("Unable to get destination address from
transport ICI.");
                                }

                        if (op_ici_attr_get (ul_iciptr, "src_addr", &org_addr) ==
OPC_COMPCODE_FAILURE)
                                {
                                ip_encap_error ("Unable to get source address from
transport ICI.");
                                }

                        if (op_ici_attr_get (ul_iciptr, "connection_class", &conn_class)
== OPC_COMPCODE_FAILURE)
                                {
                                ip_encap_error ("Unable to get connection class from
transport ICI.");
                                }

                        /*      Obtain type of service from the accompanying ICI. This
value will be later stored in the field                         */
                        /*      structure of the outgoing packet.

                    */
                        /*      If this is an ICI coming from OSPF, or IGRP, type of
service is 0 by default (set in ip_encap_v4.ic.m). */
                        if (op_ici_attr_get (ul_iciptr, "Type of Service",
&type_of_service) == OPC_COMPCODE_FAILURE)
                                {
                                ip_encap_error ("Unable to get type of service from
transport ICI.");
                                }

                        /* Retrieve the TTL value to be associated with the datagram.
               */
                        if (op_ici_attr_get (ul_iciptr, "TTL", &ttl) ==
OPC_COMPCODE_FAILURE)
                                {
                                ip_encap_error ("Unable to get TTL value from transport
ICI.");
                                }

                        /* If the destination address is multicast, then we need to
retrieve       */
                        /* major and minor ports, which the higher layer specifies.
               */
                        if (ip_address_is_multicast (dest_addr) &&  (protocol_type !=
IpC_Protocol_Rsvp))
                                {
                                if (op_ici_attr_get (ul_iciptr, "multicast_major_port",
&mcast_major_port) == OPC_COMPCODE_FAILURE)
                                        {
                                        mcast_major_port = 0;
                                        ipnl_protwarn_mcast_no_major_port_specified
(pkptr, dest_addr);

                                        /* ip_address_print (dest_addr_str, dest_addr); */
                                        /* sprintf (error_string, "Unable to retrieve
multicast major port for multicast address (%s)", */
```

```
                                        /*      dest_addr_str);
                                    */
                                        /* ip_encap_error (error_string);
                */
                                        }

                                    if (op_ici_attr_get (ul_iciptr, "multicast_minor_port",
&mcast_minor_port) == OPC_COMPCODE_FAILURE)
                                        {
                                        ip_address_print (dest_addr_str, dest_addr);
                                        sprintf (error_string, "Unable to retrieve
multicast minor port for multicast address (%s)",
                                                dest_addr_str);
                                        ip_encap_error (error_string);
                                        }

                                    /* Prepare an ICI that is to be sent to
ip_dispatch,indicating the major*/
                                    /* and minor ports on which to send the multicast packet.
                                */
                                    ip_iciptr = op_ici_create ("ip_rte_req_v4");
                                    op_ici_attr_set (ip_iciptr, "multicast_major_port",
mcast_major_port);
                                    op_ici_attr_set (ip_iciptr, "multicast_minor_port",
mcast_minor_port);

                                    /* Install this ICI    */
                                    op_ici_install (ip_iciptr);
                                    }
                                else if (protocol_type != IpC_Protocol_Rsvp)
                                    {
                                    /* No request should be made to ip_dispatch, so explicitly
de-install any outstanding ICIs. */
                                    op_ici_install (OPC_NIL);
                                    }

                                /* If this is an RSVP packet, also get Next hop Address and
Interface index       */
                                if (protocol_type == IpC_Protocol_Rsvp)
                                    {
                                    if (op_ici_attr_get (ul_iciptr, "RSVP Packet Route Info",
&pkt_route_info_ptr) == OPC_COMPCODE_FAILURE)
                                        {
                                        ip_encap_error ("Unable to get routing information
from transport ICI.");
                                        }

                                    /* Prepare an ICI that is to be sent to ip_dispatch,
indicating interface on       */
                                    /* which to send the RSVP packet so as IP does not do
route query                   */
                                    ip_iciptr = op_ici_create ("ip_rte_req_v4");

                                    op_ici_attr_set (ip_iciptr, "RSVP Packet Route Info",
pkt_route_info_ptr);

                                    op_ici_install (ip_iciptr);

                                    /* Destroy the ICI only for RSVP packets.    */
                                    op_ici_destroy (ul_iciptr);
                                    }

                                /*      Create fields data structure that contains orig_len,
        */
                                /*      ident, frag_len, ttl, src_addr, dest_addr, frag,
        */
                                /*      connection class, src and dest internal addresses.
        */
                                ip_dgram_fd_ptr = ip_dgram_fdstruct_create ();

                                /*      Assign values to members of the field structure.    */
                                ip_dgram_fd_ptr->src_addr                   = ip_address_copy
(org_addr);
```

```
                              ip_dgram_fd_ptr->dest_addr                  = ip_address_copy
(dest_addr);
                              ip_dgram_fd_ptr->connection_class    = conn_class;
                              ip_dgram_fd_ptr->orig_len                   = data_len;
                              ip_dgram_fd_ptr->frag_len                   = data_len;
                              ip_dgram_fd_ptr->ttl                        = ttl;
                              ip_dgram_fd_ptr->protocol                   = protocol_type;
                              ip_dgram_fd_ptr->tos                        = type_of_service;
                              ip_dgram_fd_ptr->compression_method = IpC_No_Compression;
                              ip_dgram_fd_ptr->original_size      = 160 + op_pk_total_size_get
(ip_pkptr);

                              /* Indicate that the packet is not yet fragmented.   */
                              ip_dgram_fd_ptr->frag                       = 0;

                              /*     Set the fields structure inside the ip datagram.     */
                              op_pk_nfd_set (ip_pkptr, "fields", ip_dgram_fd_ptr,
ip_dgram_fdstruct_copy, ip_dgram_fdstruct_destroy, sizeof (IpT_Dgram_Fields));

                              /* Destroy the dest_addr and free the memory allocated.     */
                              ip_address_destroy (dest_addr);

                              /* Forward the packet to the IP layer.       */



       /*********************************************************************/
                              /*
*/
                              /* if it is time to break the link, then intercept the packets
and   */
                              /* destroy it . Otherwise forward it to next layer.
*/
                              /*
*/

       /*********************************************************************/

                              /*if ( isBroken() )
                               {
                                 printTrace("***** link is broken now.");
                                 }
                              else
                              {*/

                                op_pk_send (ip_pkptr, outstrm_to_network);
                               // }
                              }


                        /** state (ENCAP) exit executives **/
                        FSM_STATE_EXIT_FORCED (0, "ENCAP", "YS_disc_ip_encap_v4 () [ENCAP exit
execs]")
                              {
                              }


                        /** state (ENCAP) transition processing **/
                        FSM_TRANSIT_FORCE (3, state3_enter_exec, ;, "default", "", "ENCAP",
"WAIT")
                              /*---------------------------------------------------------*/



                        /** state (INIT) enter executives **/
                        FSM_STATE_ENTER_UNFORCED_NOLABEL (1, "INIT", "YS_disc_ip_encap_v4 ()
[INIT enter execs]")
                              {


                              /** Register using OMS Process Registry.
               **/
                              printTrace( "YS-ip_encap INIT");
```

```
	/*****************************************************************************/
                            /*
*/
                            /* To get all the attributes of the module
*/
                            /* This will setup the parameters of the simulations
*/
                            /* brokenEnable  -- whether broken link is allowed during
simulation         */
                            /* cycleTime     -- broken time cycle
*/
                            /* brokenStartTime/brokenEndTime  -- time to begin and end the
broken stauts */
                            /*
*/
	/*****************************************************************************/

                            if ( op_ima_obj_attr_get( op_id_self(), "brokenEnable", &
brokenEnable )
                                    != OPC_COMPCODE_SUCCESS ) printf("Error get this attr
brokenEnalbe\n");

                            if ( op_ima_obj_attr_get( op_id_self(), "cycleTime", & cycleTime
)
                                    != OPC_COMPCODE_SUCCESS ) printf("Error get this attr
cycleTime\n");

                            if ( op_ima_obj_attr_get( op_id_self(), "brokenEndTime", &
brokenEndTime )
                                    != OPC_COMPCODE_SUCCESS ) printf("Error get this attr
brokenEndTime\n");

                            if ( op_ima_obj_attr_get( op_id_self(), "brokenStartTime", &
brokenStartTime )
                                    != OPC_COMPCODE_SUCCESS ) printf("Error get this attr
brokenStartTime\n");

                            printf("brokenenable=%d, startTime=%d, endTime=%d, cycle=%d\n",
                                     brokenEnable, brokenStartTime, brokenEndTime,
cycleTime);


                            /* Obtain the neccessary objids.
                        */
                            own_objid = op_id_self ();
                            own_node_objid = op_topo_parent (own_objid);

                            /* Obtain the ip_encap process's prohandle.
                */
                            own_prohandle = op_pro_self ();

                            /* Obtain the name of the process. It is the
                */
                            /* "process model" attribute of the module.
                */
                            op_ima_obj_attr_get (own_objid, "process model",
proc_model_name);

                            /* Register the process in the model-wide registry.
                */
                            process_record_handle = (OmsT_Pr_Handle) oms_pr_process_register
(own_node_objid, own_objid,
                                                                    own_prohandle,
proc_model_name);

                            /* Register the protocol attribute in the registry.
                */
                            oms_pr_attr_set (process_record_handle, "protocol",
OMSC_PR_STRING, "ip_encap", OPC_NIL);
                            }
```

```
                        /** blocking after enter executives of unforced state. **/
                        FSM_EXIT (3,YS_disc_ip_encap_v4)


                        /** state (INIT) exit executives **/
                        FSM_STATE_EXIT_UNFORCED (1, "INIT", "YS_disc_ip_encap_v4 () [INIT exit
execs]")
                                {
                                }


                        /** state (INIT) transition processing **/
                        FSM_TRANSIT_FORCE (4, state4_enter_exec, ;, "default", "", "INIT",
"STRM_DEMUX")
                                /*---------------------------------------------------------*/



                        /** state (DECAP) enter executives **/
                        FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "DECAP",
"YS_disc_ip_encap_v4 () [DECAP enter execs]")
                                {


                                /* Obtain a packet arriving from the network layer          */
                                //printTrace( "enter ip-encap DECAP  ...");

                                ip_pkptr = op_pk_get (instrm_from_network);
                                if (ip_pkptr == OPC_NIL)
                                        ip_encap_error ("Unable to get packet from network
layer.");

                                /*      Get the fields structure from the packet.
               */
                                op_pk_nfd_access (ip_pkptr, "fields", &ip_dgram_fd_ptr);

                                /* Determine the source address of the data.                */
                                org_addr = ip_dgram_fd_ptr->src_addr;

                                /* Determine the destination address of the data.           */
                                dest_addr = ip_dgram_fd_ptr->dest_addr;

                                /* Determine the type of service. This attribute needs to be
passed to tcp  */
                                /* layer (for tcp connectios),^^or to tpal (for udp connections).
                        */
                                type_of_service = ip_dgram_fd_ptr->tos;

                                /* Determine the protocol number.
                     */
                                protocol_type = ip_dgram_fd_ptr->protocol;

                                /* Determine the output stream index corresponding to       */
                                /* the protocol.
                                */
                                output_strm = -1;
                                for (i = 0; i < interface_table_size; i++)
                                        {
                                        if (interface_table [i]->protocol_type == protocol_type)
                                                {
                                                output_strm = interface_table [i]->outstream;
                                                break;
                                                }
                                        }

                                if (output_strm == -1)
                                        {
                                        /* If the protocol is unknown, print a trace error   */
                                        /* message.
                                        */
                                        if (op_prg_odb_ltrace_active ("ip_errs"))
```

```
                                        {
                                        op_pk_format (ip_pkptr, protocol_name);
                                        sprintf (error_string, "Packet of unknown protocol
[%s] received by IP encap. Destroying packet",
                                                protocol_name);
                                        op_sim_message (error_string, OPC_NIL);
                                        }

                                    /* Free the memory allocated to org_addr, dest_addr */
                                    /* and arrived_intf_addr.
                        */
                                ip_address_destroy (org_addr);
                                    ip_address_destroy (dest_addr);
                                    ip_address_destroy (arrived_intf_addr);

                                    ip_encap_pk_destroy (ip_pkptr);
                                    }
                                else
                                    {
                                    /* If this node is not a gateway, destory any routing
protocol packet       */
                                    if ((gateway == OPC_FALSE) && ((protocol_type ==
IpC_Protocol_Ospf) || (protocol_type == IpC_Protocol_Igrp)
                                                || (protocol_type == IpC_Protocol_Eigrp)))
                                        {
                                        /** We have received a routing protocol packet
but, routing          **/
                                        /** modules are not enabled on this node. Destroy
this packet    **/

                                        /* Print a trace message      */
                                        if (op_prg_odb_ltrace_active ("ip_errs"))
                                                {
                                                sprintf (error_string, "Received a packet
for the routing protocol %s but, routing modules",
                                                        ip_dgram_option_name_find
((IpT_Option_Type)protocol_type));
                                                op_sim_message (error_string, "on this node
are not enabled. Destroying the packet");
                                                }

                                        /* Destroy the packet  */
                                        ip_encap_pk_destroy (ip_pkptr);
                                        }
                                    else
                                        {
                                        /* Decapsulate the contained transport level data.
                */
                                        if (op_pk_nfd_get (ip_pkptr, "data", &pkptr) ==
OPC_COMPCODE_FAILURE)
                                        ip_encap_error ("Unable to get data from IP
datagram.");

                                        /* Obtain the ICI from IP indicating which
interface      */
                                        /* this packet arrived on.
                                    */
                                        ll_iciptr = op_pk_ici_get (ip_pkptr);
                                        op_ici_attr_get (ll_iciptr, "rte_info_fields",
&intf_ici_fdstruct_ptr);

                                        /* Retrieve the address of the interface.
                        */
                                        arrived_intf_addr = intf_ici_fdstruct_ptr-
>interface_received;


                                        /* Retrieve the major port information.
                                */
                                        major_port_received = intf_ici_fdstruct_ptr-
>major_port_received;
```

```
                                        /* Retrieve the minor port information.
                              */
                                        minor_port_received = intf_ici_fdstruct_ptr-
>minor_port_received;

                                        /* NOTE: Currently IGRP and EIGRP requires certain
               */
                                        /* interface information to be associated with the
               */
                                        /* packets that are forwarded to it from the IP
layer.  */
                                        /* If the protocol for which the packet is
destined is    */
                                        /* IGRP or EIGRP, then extract that information.
               */
                                        /* Otherwise, for simulation efficiency reasons,
we do   */
                                        /* not need it.
                                             */
                                        iface_mtu               = intf_ici_fdstruct_ptr-
>mtu;
                                        iface_load              = intf_ici_fdstruct_ptr-
>iface_load;
                                        iface_speed      = intf_ici_fdstruct_ptr-
>iface_speed;
                                        iface_reliability = intf_ici_fdstruct_ptr-
>iface_reliability;

                                        /* Destroy the lower layer ICI after use.    */
                                        ip_rte_ind_ici_fdstruct_destroy
(intf_ici_fdstruct_ptr);
                                        op_ici_destroy (ll_iciptr);

                                    /* Discard the IP packet.
                              */
                                        op_pk_destroy (ip_pkptr);

                                        /* Create an Ici for communication with the higher
layer.         */
                                        transp_iciptr = op_ici_create ("ip_encap_ind_v4");
                                        if (transp_iciptr == OPC_NIL)
                                                ip_encap_error ("Unable to create ICI for
communication with transport layer.");

                                    /* Forward the data to the transport layer or a routing
process */
                                        /* running over IP. Associate IP interface
information (source    */
                                        /* address and iface address on which IP datagram
was received)  */
                                        if (op_ici_attr_set (transp_iciptr, "src_addr",
org_addr) == OPC_COMPCODE_FAILURE)
                                                ip_encap_error ("Unable to set source
address in transport layer ICI.");

                                        if (op_ici_attr_set (transp_iciptr,
"interface_received", arrived_intf_addr) == OPC_COMPCODE_FAILURE)
                                                ip_encap_error ("Unable to set interface
received address in transport layer ICI.");

                                        if (op_ici_attr_set (transp_iciptr,
"major_port_received", major_port_received) == OPC_COMPCODE_FAILURE)
                                                ip_encap_error ("Unable to set major port
received in transport layer ICI.");

                                        if (op_ici_attr_set (transp_iciptr,
"minor_port_received", minor_port_received) == OPC_COMPCODE_FAILURE)
                                                ip_encap_error ("Unable to set minor port
received in transport layer ICI.");

                                        if (op_ici_attr_set (transp_iciptr, "Type of
Service", type_of_service) == OPC_COMPCODE_FAILURE)
```

```
                                                ip_encap_error ("Unable to set type of
service in transport layer ICI.");

                                        if (op_ici_attr_set (transp_iciptr, "dest_addr",
dest_addr) == OPC_COMPCODE_FAILURE)
                                                ip_encap_error ("Unable to set destination
address in transport layer ICI.");

                                        /* Again, if the packet is for the IGRP or EIGRP
dynamic        */
                                        /* routing protocols, then associate some more
interface              */
                                        /* information. Currently, only IGRP and EIGRP use
this to        */
                                        /* compute route metrics.
                                                */
                                        if ((protocol_type == IpC_Protocol_Igrp) ||
(protocol_type == IpC_Protocol_Eigrp))
                                                {
                                                op_ici_attr_set (transp_iciptr,
"iface_mtu",         iface_mtu);
                                                op_ici_attr_set (transp_iciptr,
"iface_load",        iface_load);
                                                op_ici_attr_set (transp_iciptr,
"iface_speed",       iface_speed);
                                                op_ici_attr_set (transp_iciptr,
"iface_reliability", iface_reliability);
                                                }

                                        /* Install the ICI and send the packet to the
               */
                                        /* higher layer.
                                */
                                        op_ici_install (transp_iciptr);


        /*********************************************************************/
                                /*
*/
                                /* if it is time to break the link, then intercept the
packets and   */
                                /* destroy it . Otherwise forward it to next layer.
*/
                                /*
*/
        /*********************************************************************/
                                if ( isBroken() )
                                {       printTrace("***** link is broken now.");
                                    op_pk_destroy( pkptr );
                                }
                                else
                                        op_pk_send (pkptr, output_strm);
                                        }

                                }

                        }


                    /** state (DECAP) exit executives **/
                    FSM_STATE_EXIT_FORCED (2, "DECAP", "YS_disc_ip_encap_v4 () [DECAP exit
execs]")
                            {
                            }


                    /** state (DECAP) transition processing **/
                    FSM_TRANSIT_FORCE (3, state3_enter_exec, ;, "default", "", "DECAP",
"WAIT")
                            /*---------------------------------------------------------*/
```

```
                        /** state (WAIT) enter executives **/
                        FSM_STATE_ENTER_UNFORCED (3, state3_enter_exec, "WAIT",
"YS_disc_ip_encap_v4 () [WAIT enter execs]")
                                {
                                }


                        /** blocking after enter executives of unforced state. **/
                        FSM_EXIT (7,YS_disc_ip_encap_v4)


                        /** state (WAIT) exit executives **/
                        FSM_STATE_EXIT_UNFORCED (3, "WAIT", "YS_disc_ip_encap_v4 () [WAIT exit
execs]")
                                {
                                }


                        /** state (WAIT) transition processing **/
                        FSM_INIT_COND (FROM_TRANSPORT)
                        FSM_TEST_COND (FROM_NETWORK)
                        FSM_TEST_LOGIC ("WAIT")

                        FSM_TRANSIT_SWITCH
                                {
                                FSM_CASE_TRANSIT (0, 0, state0_enter_exec, ;, "FROM_TRANSPORT",
"", "WAIT", "ENCAP")
                                FSM_CASE_TRANSIT (1, 2, state2_enter_exec, ;, "FROM_NETWORK", "",
"WAIT", "DECAP")
                                }
                                /*---------------------------------------------------------*/



                        /** state (STRM_DEMUX) enter executives **/
                        FSM_STATE_ENTER_FORCED (4, state4_enter_exec, "STRM_DEMUX",
"YS_disc_ip_encap_v4 () [STRM_DEMUX enter execs]")
                                {
                                /** This state is used to find out which modules are        **/
                                /** directly connected to ip_encap. The stream indices are  **/
                                /** stored together with the protocol attributes for        **/
                                /** demultiplexing different higher-layer protocols.        **/

                                /* Find the object IDs for use in process discovery.       */
                                own_objid = op_id_self ();
                                own_node_objid = op_topo_parent (own_objid);

                                /* Search for neighbors using the process registry.  All   */
                                /* neighbors must be registered.
                        */
                                proc_record_handle_list_ptr =  op_prg_list_create();
                                oms_pr_process_discover (own_objid, proc_record_handle_list_ptr,
"node objid", OMSC_PR_OBJID, own_node_objid, OPC_NIL);

                                record_handle_list_size = op_prg_list_size
(proc_record_handle_list_ptr);

                                if (record_handle_list_size == 0)
                                        {
                                        /* An error should be created if no processes are
        */
                                        /* connected to ip_encap.
                                */
                                        op_sim_end ("Error: no modules connected to ip_encap", "",
"", "");
                                        }
                                else
                                        {
                                        /** For all the directly connected networks, find the
        **/
```

```
                                        /** protocol, as well the input and output stream
        **/
                                        /** indices.
                                        **/

                                        /* Create the interface information structure.
                */
                                        interface_table = (IpT_Encap_Interface **)
op_prg_mem_alloc (record_handle_list_size *
                                                sizeof (IpT_Encap_Interface *));

                                        /* If the ip_encap module is not directly connected to
        */
                                        /* the ip module, the stream indices for the
                */
                                        /* intermediate must be 0.  If ip is a neighbor, the
        */
                                        /* correct stream indices will be used.
                        */
                                        instrm_from_network = 0;
                                        outstrm_to_network = 0;

                                        /* Maintain a count of the number of records in the
        */
                                        /* interface table.
                                        */
                                        interface_table_size = 0;

                                        /* Loop through the discovered processes, adding the */
                                        /* interface records to the list.
                        */
                                        for (i = 0; i < record_handle_list_size; ++i)
                                                {
                                                process_record_handle = (OmsT_Pr_Handle)
op_prg_list_access (proc_record_handle_list_ptr, i);

                                                /* Obtain the module object id of the neighboring
        */
                                                /* module, and the value of the protocol
attribute. */
                                                oms_pr_attr_get (process_record_handle, "module objid",
OMSC_PR_OBJID, &neighbor_mod_objid);

                                                /* There is a possibility that the record returned
        */
                                                /* by process registry may correspond to processes
        */
                                                /* registered without the "protocol" attribute (an
        */
                                                /* example is the case when child process(es) of
        */
                                                /* the OSPF protocol register -- these records do
        */
                                                /* _not_ register any "protocol" attribute.)
        */
                                                if (oms_pr_attr_get (process_record_handle, "protocol",
OMSC_PR_STRING, protocol_name) == OPC_COMPCODE_SUCCESS)
                                                        {
                                                        /* Determine the input and output stream
indices.       */
                                                        oms_tan_neighbor_streams_find (own_objid,
neighbor_mod_objid, &input_strm, &output_strm);

                                                        if (strcmp (protocol_name, "ip") == 0)
                                                                {
                                                                /* This is the ip module in this
node,  It does */
                                                                /* not need to be added to the
interface table,       */
                                                                /* but it's stream indices should
be stored.     */
                                                                instrm_from_network = input_strm;
                                                                outstrm_to_network = output_strm;
```

```
                                                        }
                                                else
                                                        {
                                                        /* Store the upper layer module
information in */
                                                        /* an interface table.
                                                */
                                                        /* Allocate memory for the
interface element.    */
                                                        interface_table
[interface_table_size] = (IpT_Encap_Interface *) op_prg_mem_alloc (sizeof
(IpT_Encap_Interface));

                                                        /* Store the protocol number and
the                   */
                                                        /* corresponding stream indices
into the             */
                                                        /* interface table.
                                                */
                                                        interface_table
[interface_table_size]->protocol_type = ip_higher_layer_proto_id_find (protocol_name);
                                                        interface_table
[interface_table_size]->outstream = output_strm;
                                                        interface_table
[interface_table_size]->instream = input_strm;

                                                        /* Maintain a count of the number
of records in  */
                                                        /* the interface table.
                                                */
                                                        interface_table_size++;
                                                        }
                                                }
                                        }
                                }

                        /* Deallocate the registry list pointer.
                */
                        op_prg_mem_free (proc_record_handle_list_ptr);

                        /** Determine whether this node is a gateway or not. Note that if
        **/
                        /** this node is a gateway, the IP process model would register
the     **/
                        /** attribute "gateway node" in its process registry
                **/

                        /* Initialize the gateway state variable to OPC_FALSE       */
                        gateway = OPC_FALSE;

                        /* Obtain the process registry for the IP process model in this
node    */
                        proc_record_handle_list_ptr = op_prg_list_create ();
                        oms_pr_process_discover (OPC_OBJID_INVALID,
proc_record_handle_list_ptr,
                                "node objid",  OMSC_PR_OBJID,
        own_node_objid,
                                "protocol",           OMSC_PR_STRING,
        "ip",
                                OPC_NIL);

                        /* There should only be one IP routing module. */
                        if (op_prg_list_size (proc_record_handle_list_ptr) != 1)
                                {
                                ip_encap_error ("Found none or more than one IP module in
this node.");
                                }

                        /* Get the process record handle for the IP process model   */
                        process_record_handle = (OmsT_Pr_Handle) op_prg_list_access
(proc_record_handle_list_ptr, OPC_LISTPOS_HEAD);
```

```
                                /* Deallocate the registry list pointer.
                */
                                op_prg_mem_free (proc_record_handle_list_ptr);

                                /* Check if IP process model has registered "gateway node"  */
                                /* attribute in its process registry
                */
                                if (oms_pr_attr_get (process_record_handle, "gateway node",
OMSC_PR_STRING, gateway_str, OPC_NIL)
                                        == OPC_COMPCODE_SUCCESS)
                                        {
                                        gateway = OPC_TRUE;
                                        }

                                }


                        /** state (STRM_DEMUX) exit executives **/
                        FSM_STATE_EXIT_FORCED (4, "STRM_DEMUX", "YS_disc_ip_encap_v4 ()
[STRM_DEMUX exit execs]")
                                {
                                }


                        /** state (STRM_DEMUX) transition processing **/
                        FSM_INIT_COND (FROM_NETWORK)
                        FSM_TEST_COND (FROM_TRANSPORT)
                        FSM_TEST_LOGIC ("STRM_DEMUX")

                        FSM_TRANSIT_SWITCH
                                {
                                FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;, "FROM_NETWORK", "",
"STRM_DEMUX", "DECAP")
                                FSM_CASE_TRANSIT (1, 0, state0_enter_exec, ;, "FROM_TRANSPORT",
"", "STRM_DEMUX", "ENCAP")
                                }
                                /*---------------------------------------------------------*/



                        }


                FSM_EXIT (1,YS_disc_ip_encap_v4)
                }
        }

#if defined (__cplusplus)
        extern "C" {
#endif
        extern VosT_Fun_Status Vos_Catmem_Register (const char * , int , VosT_Void_Null_Proc,
VosT_Address *);
        extern VosT_Address Vos_Catmem_Alloc (VosT_Address, size_t);
        extern VosT_Fun_Status Vos_Catmem_Dealloc (VosT_Address);
#if defined (__cplusplus)
        }
#endif


Compcode
YS_disc_ip_encap_v4_init (void ** gen_state_pptr)
        {
        int _block_origin = 0;
        static VosT_Address   obtype = OPC_NIL;

        FIN (YS_disc_ip_encap_v4_init (gen_state_pptr))

        if (obtype == OPC_NIL)
                {
                /* Initialize memory management */
                if (Vos_Catmem_Register ("proc state vars (YS_disc_ip_encap_v4)",
                        sizeof (YS_disc_ip_encap_v4_state), Vos_Vnop, &obtype) == VOSC_FAILURE)
                        {
```

```
                FRET (OPC_COMPCODE_FAILURE)
                }
        }

    *gen_state_pptr = Vos_Catmem_Alloc (obtype, 1);
    if (*gen_state_pptr == OPC_NIL)
        {
        FRET (OPC_COMPCODE_FAILURE)
        }
    else
        {
        /* Initialize FSM handling */
        ((YS_disc_ip_encap_v4_state *)(*gen_state_pptr))->current_block = 2;

        FRET (OPC_COMPCODE_SUCCESS)
        }
    }



void
YS_disc_ip_encap_v4_diag (void)
    {
    int _block_origin = __LINE__;

    FIN (YS_disc_ip_encap_v4_diag ())

    if (1)
        {
        Packet*                     ip_pkptr;
        Packet*                     pkptr;

        Ici*                        ul_iciptr;
        Ici*                        ll_iciptr;
        Ici*                        ip_iciptr;
        Ici*                        transp_iciptr;

        IpT_Address                 arrived_intf_addr;
        int                             minor_port_received;
        int                         major_port_received;
        IpT_Address                 org_addr;
        IpT_Address                 dest_addr;
        IpT_Address                 next_hop_addr;
        char                        dest_addr_str [IPC_ADDR_STR_LEN];
        int                             type_of_service;

        double                      iface_mtu;
        double                      iface_load;
        double                      iface_speed;
        double                      iface_reliability;

        int                             i;
        int                             data_len;
        int                             input_strm, output_strm;
        int                             num_char;
        int                             conn_class;
        int                             mcast_major_port;
        int                             mcast_minor_port;
        int                         ttl;
        int                             intf_index;
        char                        protocol_name[20];
        char                        proc_model_name[20];
        char                        error_string [512];

        IpT_Dgram_Fields*       ip_dgram_fd_ptr;
        int                             protocol_type;

        List*                           proc_record_handle_list_ptr;
        int                             record_handle_list_size;
        OmsT_Pr_Handle          process_record_handle;
        Objid                           own_objid;
        Objid                       own_node_objid;
        Objid                       neighbor_mod_objid;
```

```
		Prohandle			own_prohandle;
		char				gateway_str [32];

		/* IP datagrams maintain an ICI to record information	*/
		/* required for proper routing (e.g., slot on which the	*/
		/* datagram arrived, output interface on which it	*/
		/* should be forwarded, etc.) -- all that information	*/
		/* is stored in a single structure field of the			*/
		/* ip_rte_ind_v4 ICI.
	*/
		IpT_Rte_Ind_Ici_Fields*		intf_ici_fdstruct_ptr;

		/* RSVP packets must be forwarded on interface specified	*/
		/* by RSVP. The routing information for RSVP packets is		*/
		/* carried in ip_encap_reg ICI and also in ip_rte_reg ICI.	*/
		RsvpT_Rte_Ici_Struct *		pkt_route_info_ptr;


		/* Diagnostic Block */


		BINIT


		/* End of Diagnostic Block */

		}

	FOUT;
	}



void
YS_disc_ip_encap_v4_terminate (void)
	{
	int _block_origin = __LINE__;

	FIN (YS_disc_ip_encap_v4_terminate (void))

	if (1)
		{
		Packet*			ip_pkptr;
		Packet*			pkptr;

		Ici*			ul_iciptr;
		Ici*			ll_iciptr;
		Ici*			ip_iciptr;
		Ici*			transp_iciptr;

		IpT_Address		arrived_intf_addr;
		int				minor_port_received;
		int			major_port_received;
		IpT_Address		org_addr;
		IpT_Address		dest_addr;
		IpT_Address		next_hop_addr;
		char			dest_addr_str [IPC_ADDR_STR_LEN];
		int				type_of_service;

		double			iface_mtu;
		double			iface_load;
		double			iface_speed;
		double			iface_reliability;

		int				i;
		int				data_len;
		int				input_strm, output_strm;
		int				num_char;
		int				conn_class;
		int				mcast_major_port;
		int				mcast_minor_port;
		int			ttl;
```

```
		int                                     intf_index;
		char                            protocol_name[20];
		char                            proc_model_name[20];
		char                            error_string [512];

		IpT_Dgram_Fields*       ip_dgram_fd_ptr;
		int                                     protocol_type;

		List*                           proc_record_handle_list_ptr;
		int                                     record_handle_list_size;
		OmsT_Pr_Handle          process_record_handle;
		Objid                           own_objid;
		Objid                           own_node_objid;
		Objid                           neighbor_mod_objid;
		Prohandle                       own_prohandle;
		char                            gateway_str [32];

		/* IP datagrams maintain an ICI to record information       */
		/* required for proper routing (e.g., slot on which the     */
		/* datagram arrived, output interface on which it           */
		/* should be forwarded, etc.) -- all that information       */
		/* is stored in a single structure field of the                     */
		/* ip_rte_ind_v4 ICI.
	*/
		IpT_Rte_Ind_Ici_Fields*                 intf_ici_fdstruct_ptr;

		/* RSVP packets must be forwarded on interface specified    */
		/* by RSVP. The routing information for RSVP packets is             */
		/* carried in ip_encap_reg ICI and also in ip_rte_reg ICI.  */
		RsvpT_Rte_Ici_Struct *                  pkt_route_info_ptr;


		/* No Termination Block */

		}
	Vos_Catmem_Dealloc (pr_state_ptr);

	FOUT;
	}


/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in YS_disc_ip_encap_v4_svar function. */
#undef interface_table
#undef interface_table_size
#undef instrm_from_network
#undef outstrm_to_network
#undef gateway



void
YS_disc_ip_encap_v4_svar (void * gen_ptr, const char * var_name, char ** var_p_ptr)
	{
	YS_disc_ip_encap_v4_state               *prs_ptr;

	FIN (YS_disc_ip_encap_v4_svar (gen_ptr, var_name, var_p_ptr))

	if (var_name == OPC_NIL)
		{
		*var_p_ptr = (char *)OPC_NIL;
		FOUT;
		}
	prs_ptr = (YS_disc_ip_encap_v4_state *)gen_ptr;

	if (strcmp ("interface_table" , var_name) == 0)
		{
		*var_p_ptr = (char *) (&prs_ptr->interface_table);
		FOUT;
		}
	if (strcmp ("interface_table_size" , var_name) == 0)
		{
```

```
        *var_p_ptr = (char *) (&prs_ptr->interface_table_size);
        FOUT;
        }
if (strcmp ("instrm_from_network" , var_name) == 0)
        {
        *var_p_ptr = (char *) (&prs_ptr->instrm_from_network);
        FOUT;
        }
if (strcmp ("outstrm_to_network" , var_name) == 0)
        {
        *var_p_ptr = (char *) (&prs_ptr->outstrm_to_network);
        FOUT;
        }
if (strcmp ("gateway" , var_name) == 0)
        {
        *var_p_ptr = (char *) (&prs_ptr->gateway);
        FOUT;
        }
*var_p_ptr = (char *)OPC_NIL;

FOUT;
}
```