Performance of TCP Protocol Running over WLAN 802.11 with the Snoop Protocol

ENSC 833 Project Report

Jack Chi-Kit Chow (jackchow@home.com)

Chi-ho Ng (cng@sierrawireless.com)

Winter 2001

ABSTACT

"Performance of TCP Protocol Running over WLAN 802.11 with Performance Enchancing Proxy (PEP)"

by Jack Chi-Kit Chow and Chi-Ho Ng

With the heavy usage of the TCP/IP protocol on the Internet and the growing popularity of using wireless devices to access the Internet, it is expected that TCP protocol will be frequently used over the wireless link connecting the wireless devices in the near future.

The connection characteristics (e.g. bit error rate) of the wireless link are significantly different from that of a wired line network because data are frequently lost due to the volatile environment that a wireless link operates. TCP is originally targeted towards a wired system, which assumes any loss of data is caused by congestion. This assumption leads to poor performance of TCP over wireless link. Therefore, a number of mechanisms are proposed by various research groups to improve TCP performance over wireless link. Implementing a Performance Enhancing Proxy (PEP), such as the Snoop Protocol, is one of the ways that has been proposed to improve the TCP performance over wireless link.

The goal of this report is to study how the Snoop Protocol improves the performance of TCP over wireless link. The Snoop Protocol is implemented and simulated in OpNet WLAN devices. Improvement achieved by the Snoop Protocol is measured by comparing the performance of a FTP session with and without the Snoop Protocol implemented. It is found that the Snoop Protocol improves the TCP performance significantly (68 times under packet error rate of 30%). In addition, it is shown that the

improvement is achieved by preventing the TCP layer from reducing the congestion window size.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AP	Access Point
ARP	Address Resolution Protocol
BSS	Infrastructure Basic Service Set
IBSS	Independent Basic Service Set
MAC	Medium Access Control
PEG	Packet Error Generator
PEP	Performance Enhancing Protocol
PER	Packet Error Rate
TCP	Transmission Control Protocol
WLAN	Wireless Local Area Network

1 Introduction

TCP is a reliable protocol designed to perform well in networks with low bit-error rate, such as wired networks. TCP assumes that all errors are due to network congestion, not due to loss. When congestion is encountered, TCP adjusts its window size and re-transmits the lost packets. This mechanism works well in wired networks. However, in wireless networks where packet loss is due to high bit-error rate over the air-link, the TCP window adjustment and re-transmission mechanism results in poor end-to-end performance.

A number of approaches have been proposed to increase the TCP efficiency in an unreliable wireless network. One of these approaches is called the Snoop Protocol. The protocol modifies network-layer software mainly at a base station and preserves end-to-end TCP semantics. The main idea of the protocol is to cache packets at the base station and perform local retransmissions across the wireless link.

In the ENSC 833 project, we implement the snoop 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 snoop protocol. This report examines our implementation of the snoop protocol and the performance improvement.

2 Technical Overview

This section gives a technical overview of the technologies that are related to this project.

2.1 TCP

TCP is a reliable protocol that performs efficiently in wire-line networks. TCP uses a Go-back N protocol and a timer based retransmission mechanism. The timer is calculated based on the estimate of the round-trip delay. Packets whose acknowledgements are not received within the timer's period are retransmitted. In the presence of frequent retransmission, TCP assumes that there is a congestion loss and invokes its congestion control algorithm. The algorithm reduces the transmission window (also called congestion windows) size. As the window size is reduced, transmission rate is also reduced. This window adjustment is to prevent the source from overwhelming the destinations with an excess amount of packets.

The TCP starts by increasing the window size exponentially to quickly determine the available transmission rate. When the source fails in receiving an acknowledgement, it 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 a packet is lost, the window size is halved and the algorithm repeats.

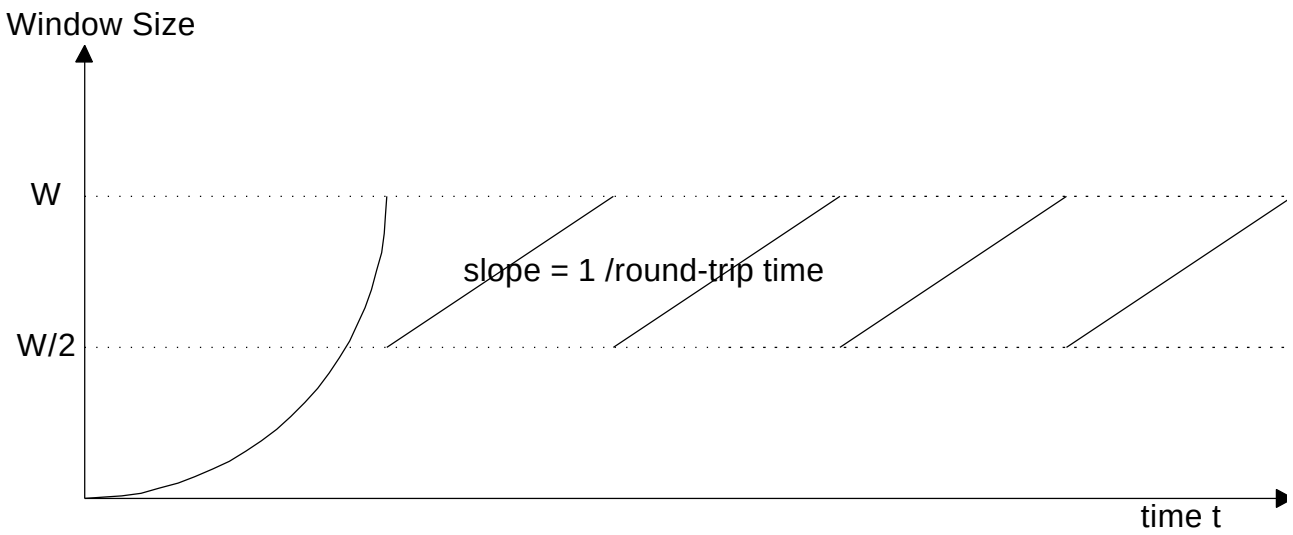


Figure 1 TCP transmission window size

In the presence of high error rates in a wireless link, TCP reacts the same way as it would in a wired link: it drops the window size before packet retransmission. This adjustment results in an unnecessary reduction in the bandwidth utilization, causing significant performance degradation in the form of poor throughput and long delays. The bandwidth of the wired-link section is especially under-utilized since the high error rate is occurring only in the wireless section but the window reduction affects the entire transmission link.

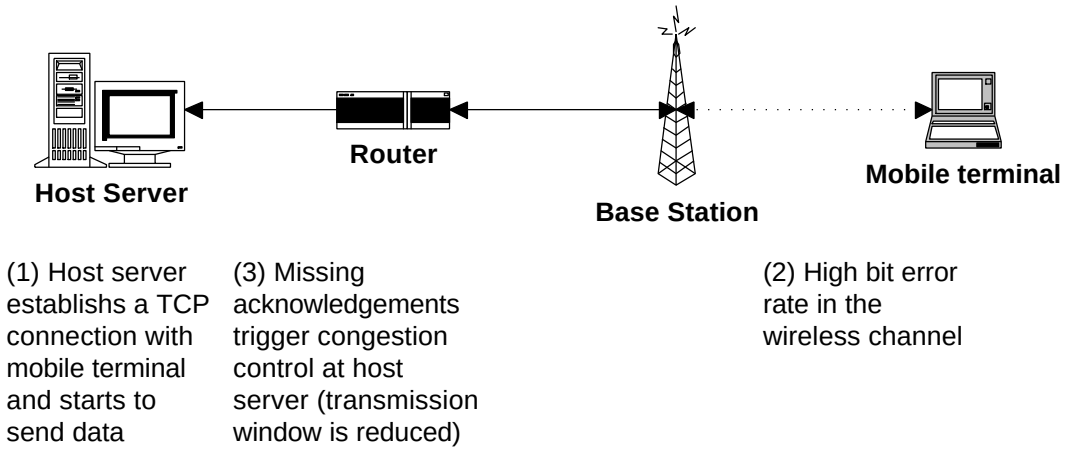


Figure 2 High Bit Error Rate in Wireless Channel trigger reduced TCP window size

2.2 Performance Enhancing Proxies (PEP)

Since the majority of network applications are running on top of TCP, it is important to improve its performance in wireless networks. Performance Enhancing Proxies (PEP) is names for a set of different tactics used in different environments to reduce performance degradation due to different link characteristics. In the wireless networks, different PEP mechanisms have been proposed and used to improve the performance of the TCP protocol. The following section is a description of the Data-link Level Retransmission Method.

2.2.1 Data-link level Retransmission Method

In this method, the wireless link implements a retransmission protocol at the data-link level. The major goal of this method is to improve the performance of communication at the wireless link without triggering retransmission and window reducing polices at the transport layer, i.e. TCP. This method attempts to improve the performance of the communication system by locally retransmiting lost packets, without informing the upper

transport layer. One mechanism that uses this method is called Snoop Protocol, and will be described next.

2.3 Snoop Protocol

Snoop Protocol runs on a Snoop Agent, which is implemented in a base station or a wireless device. It is intended to improve the performance of the TCP protocol running over a wireless link.

The Snoop agent monitors packets that passes through the base station and caches the packets in a table. After caching, the agent forwards the packets to the destination and monitors the corresponding acknowledgements.

In TCP, each acknowledgement is associated with an acknowledgement sequence number. This number tells the sender the last successfully received bytes received by the receiver. If the sender receives the same acknowledgement sequence number more than once, this suggests that data sent since the last received bytes reported in the acknowledgement sequence number is lost. Acknowledgement that contains acknowledgement sequence number smaller than the last Ack is referred to as duplicate Ack.

Consider a system that employs the Snoop Agent and a TCP sender sends packets 1, 2, 3, 4, 5, which are forwarded by a base station. Assume packet 2 is loss due to error on the wireless link. Receipt of packets 3, 4, 5 triggers sending of duplicate Acks. When the sender receives three duplicate Acks, it retransmits the packets, with a reduction of congestion window.

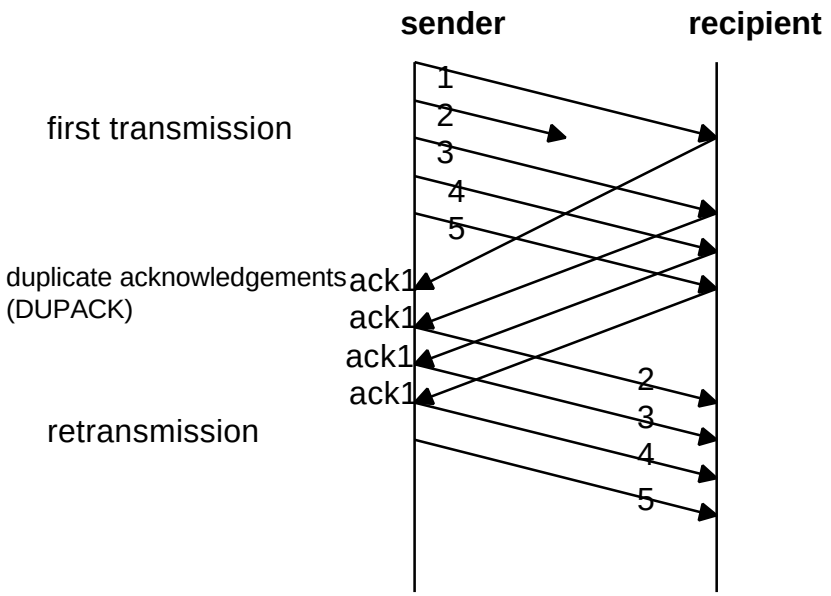


Figure 3 TCP sender retransmission

The job of the Snoop Agent is to cache data packets for a TCP connection. When the data packets are determined to be lost, identified by duplicate Acks, it retransmits the data packets. The Snoop Agent re-transmits the packets locally without forwarding the Acks to the sender. As a result, the TCP layer is not aware of the packet loss and congestion control algorithm is not triggered.

In addition, the Snoop Agent starts a retransmission timer for each TCP connection. When the retransmission timer expires, the Snoop Agent will retransmit the packets that have not yet acknowledged. This timer is called a persist timer because, unlike TCP retransmission timer, it is fixed to a particular value.

Figure 4 shows a Snoop Agent implemented in a base station.

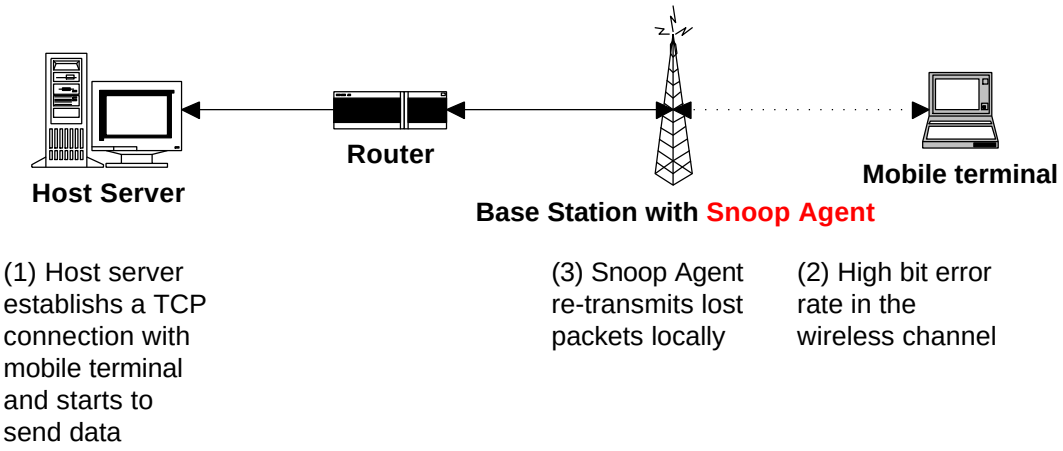


Figure 4 Snoop Agent at a Base Station

Snoop Protocol intercepts TCP packets, analyzes them, and retransmits TCP packets if necessary. As a result, no extra packet formats are introduced into the protocol, and all packets sent and received still conform to the TCP protocol. As a result, no changes to the other layer in the TCP/IP protocol should be necessary. It is extremely important since the TCP/IP protocol is designed for wire-link network, which is the most likely kind of network that servers are located. (Refer to Figure 4).

The goal of this project is implement the Snoop Agent in OpNet WLAN device. The Snoop Agent implemented should:

1. Improve the TCP performance as suggested
2. Does not require tuning of default parameters

2.4 WLAN 802.11

The Opnet's standard model WLAN LAN 802.11 will be used in our implementation of the Snoop Protocol. This section provides the background of the 802.11 standard that are related.

Originally formed in 1990, the IEEE 802.11 working group took on the task of developing standards for all kinds of wireless communications. Their efforts gradually became more focused on individual areas such as wireless LANs. As a result, in 1993, the first portion of a wireless standard, known as WLAN 802.11 was approved. Two protocol layers are the focus of the standard: the Medium Access Control (MAC) Layer and the Physical Layer. Each of the layers is explained in the following sections.

2.4.1 MAC

The MAC protocol uses Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). When using CSMA/CA, wireless nodes wait for silence on the network before it tries to transmit. When silence is detected, a node transmits its data, and waits to receive an acknowledgement from the recipient. If it does not receive an acknowledgement, it assumes that a collision has occurred. The sender will then wait a random period of time and then retransmit again.

One benefit of the 802.11 MAC-layer protocol is that it is tied to the IEEE 802.2 Logical Link Control (LLC) layer as the MACs of other 802.x wired LAN standard does. This makes the 802.11 WLAN easy to integrate with other LANs.

2.4.2 Physical Layer

There are 3 different sets of specifications for the physical layer:

- Infrared light LANs

- Frequnce-hopping spread-spectrum (FHSS) LANs

- Direct sequence spread-spectrum (DSSS) LANs

The latter two protocols are for radio frequency. They can support up to 2 Mbps while the Infrared Light spec can support up to 1Mbps. However, improvements to the protocols are made regularly and the transmission rates are increased. Details on these physical layers can be found in most books on the WLAN 802.11 standard or from the 802.11 workgroup's web site: http://www.manta.ieee.org/groups/802/11

2.4.3 Network Configuration

802.11 architecture supports two network configurations: Infrastructure Basic Service Set (BSS) and Independent BSS (IBSS). A BSS is a set of stations that communicate with one another. The BSS is called an IBSS when all the stations can communicate directly with each other. IBSS is commonly called an adhoc network.

On the other hand, when a BSS includes an access point (AP), the BSS is called and infrastructure BSS or simply BSS. All the workstations communicate via the AP. The AP also provides connection to the wired LAN. One type of AP is a base station.

3 Opnet WLAN model

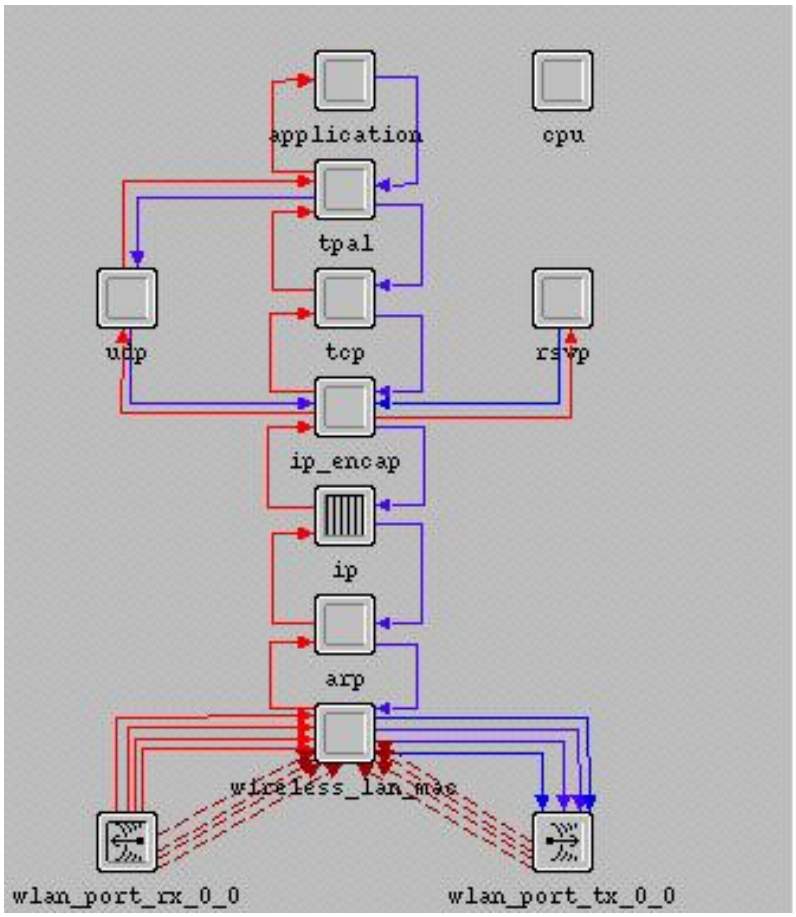
The Opnet WLAN stimulation model support both Independent BSS (IBSS) and Infrustrature BSS. The following models are contained in the WLAN object palette:

- Wireless Workstation (fixed and mobile)
- Wireless Server (fixed and mobile)
- Wireless terminal station (WLAN MAC without IP)
- Wireless Router (with one wireless interface and one or more wired interface like Ethernet, FDDI, etc)

In particular, the wireless workstation modeling of the MAC and the physical layer is comprised of the wirless_lan_mac process, transmitter, receiver, and the channel streams. The address resolution protocol (ARP) is an interface between the MAC and the higher layers. The ARP translates IP addresses into network interface addresses. For Ethernet, the address is a 48-bit address that is hard-coded with every Ethernet network card.

Please note that the Opnet WLAN model does not stimulate the actual physical layer IEEE 802.11 specification.

Since this project is focused on the improvements of TCP performance over the wireless



link, only independent network configuration is used. In other words, only the wireless workstation and the wireless server models will be used.

3.1 Model Modifications

The Opnet WLAN model does not allow the user to inject packet loss or bit error rate into the WLAN network. To facilitate testing of the snoop protocol, an additional layer, called the Packet Error Generator (PEG), is added in between the ARP and the IP layer. The PEG allows the user to specific the percentage of packet error, which is generated by an uniform distribution function.

On top of the PEG layer, the Snoop Agent is added. The following is a diagram of the modified WLAN layers.

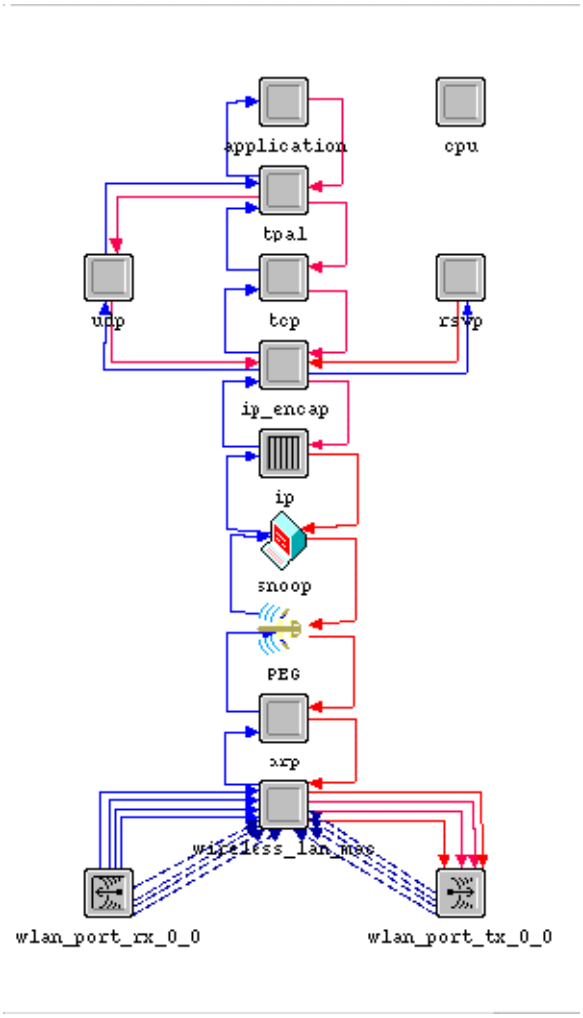


Figure 5. Modified WLAN Device Model

Of the various places that the additional process models can be added, the location is chosen.because of the way OpNet models are implemented. Placing the extra process

models in other locations would require changes to both the upper and lower layers of However, the location chosen only requires changes to the ARP model.

The ARP model is modified to send and receive packets to/from the Snoop Agent.

4 Packet Error Generator

In order to evaluate the performance of the Snoop protocol under conditions where packets are corrupted due to link errors, a mechanism is required to inject packet loss within the network. Unfortunately, the standard OpNet model does not provide a direct method to inject errors in wireless link.

As a result, a process model is developed and places within the WLAN device node model. This process model will simulate packet loss by dropping packets that are received. The amount of packet to be dropped (in percentage of the total packets received) can be exported as model attributes.

For the rest of the document, the process model to generate packet loss is referred as the Packet Error Generator (PEG).

4.1 PEG State Diagram

Figure 6 depicts the state diagram of the PEG process model.

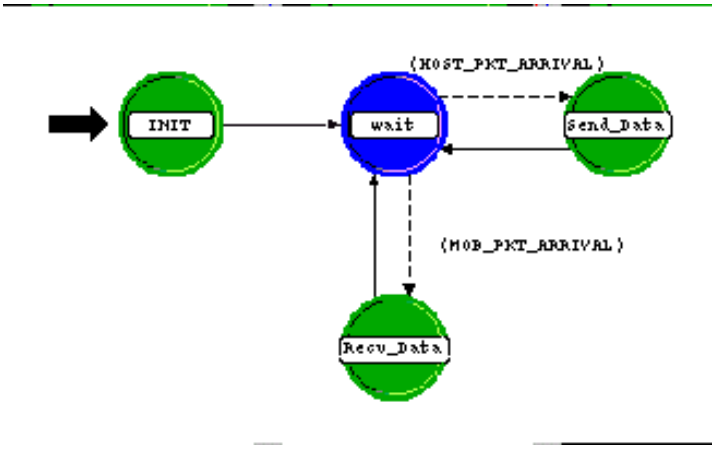


Figure 6. State Models of the PEG Process Model

The INIT state initializes the various global variables when the process is first instantiated. The process then sits in the Wait state until either a packet is received from the upper layer or the lower layer. Depending on the direction that the packet is traveling, the process enters either the Send_Data or Recv_Data state.

The instructions in the Send_Data and the Recv_Data make the decision whether the current packet should be dropped or not.

4.2 Algorithm/ Implementation

The following shows the pseudo-code that resides in both the Send_Data and Recv_Data that decides if a packet should be dropped.

```
random_val = op_dist_uniform (100.0)
get TCP info of the packet
if (packet is a SYN or FIN packet)
{
  pass packets to the next layer
}
else if (random_val > packet_drop_rate)
{
  pass packets to the next layer
}
else
{
   destroy packet
}
```

The PEG model employs a uniform distribution variable to determine if a certain packet should be dropped. Under this model, all packets have the same probability of getting dropped. This model fits the real scenarios where packet loss occur randomly depending on the conditions of the environment.

Figure 7 shows a typical scenario of how packets are dropped under this approach.

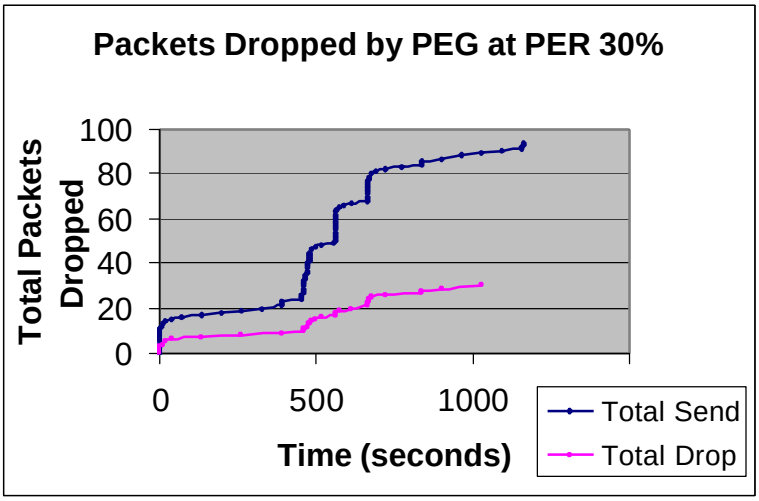


Figure 7. Packet Drops at Data_Drop_Rate of 30%

4.3 Model Attributes

Table 1 shows the model attributes that are exported and can be adjusted through the Opnet user interface.

Model Attribute	Attribute Type	Explanation
Send Data Drop Rate	Double	Percentage of packets to be dropped for packets that pass from the upper layer (IP) to the lower layer (MAC)
Receive Data Drop Rate	Double	Percentage of packets to be dropped for packets that pass from the lower layer (MAC) to the upper layer (IP)

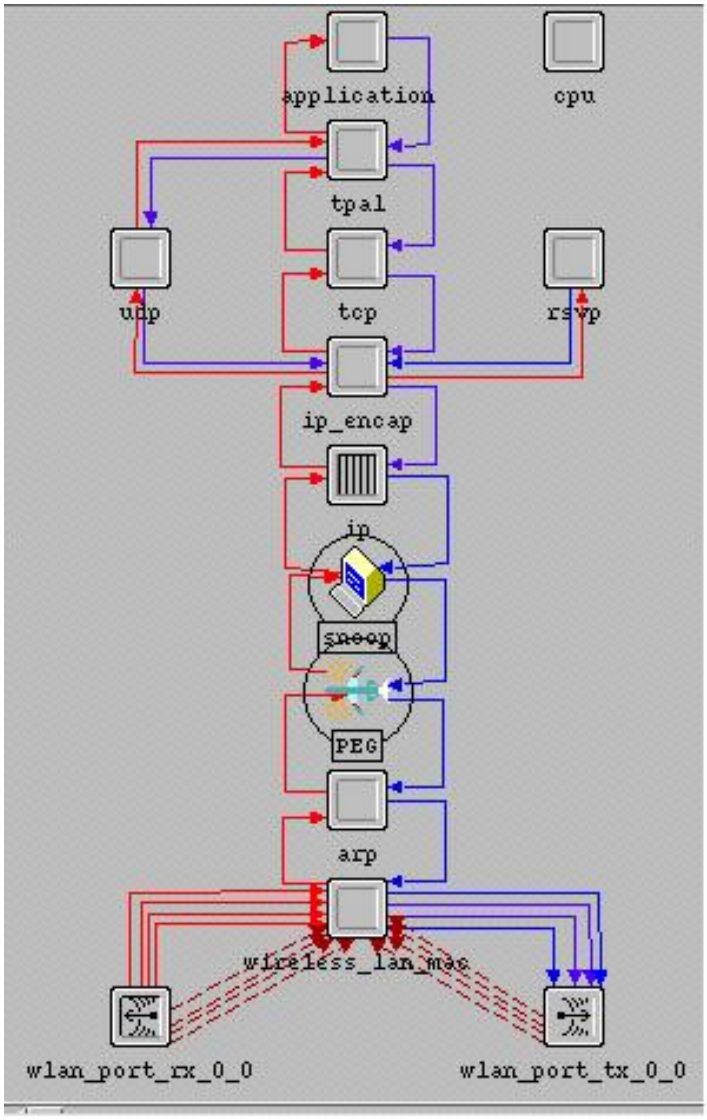


Table 1 Model Attribute of PEG Process Model

5 Implementation of the Snoop Protocol

We will examine the implementation of the Snoop Agent as a process model in this section.

The following diagram shows the 5 states and the state transitions in the snoop process.

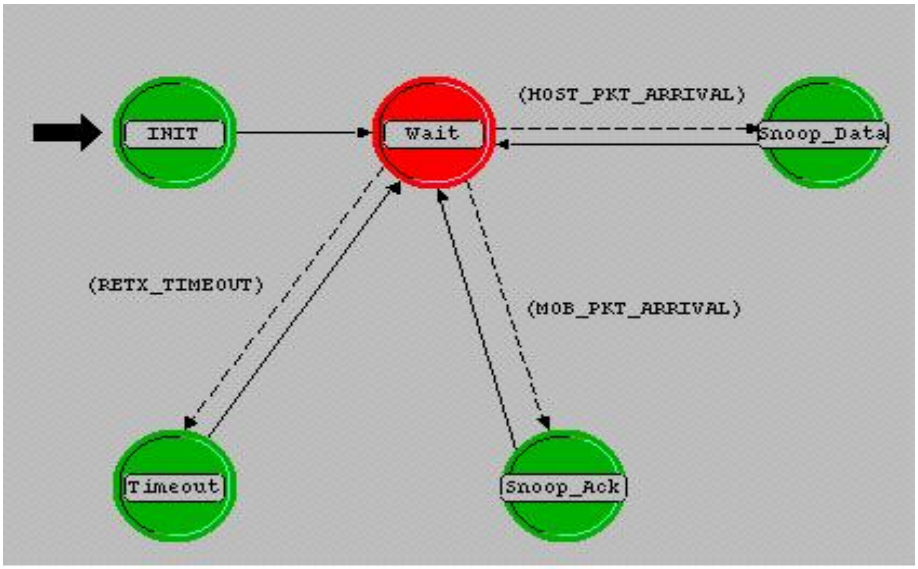


Figure 8 The Snoop State Transition Diagram

5.1 Snoop Agent Structure

5.1.1 Snoop Cache

The Snoop Cache is used to cache the packets that are received from higher layer. Packets are cached in a linear array. The C definition of the Snoop Cache is defined as:

```
typedef struct
 {
  Packet cached_pkp[SNOOP_CACHE_SIZE]; /* packet cache */
```

```
  int ici_addr[SNOOP_CACHE_SIZE]; /* ICI content cache */
  int max_cached;   /* Max number of pkt that has been cached at
the same time */
  int num_cached;    /* Number of packets that has been cached */
  int num_removed;   /* Number of packets that has been removed */
  int curr_cached;   /* Current number of packets cached */
} struct_snCache;
```

The cache table contains the copy of packets and the ICI content as well as various counters such as the total number of packets that has been cached.

The following table summarizes the cache function routines:

Function Name	Description	Input parameters	Return Value
snCacheInit	Initialize the cache records	null	null
snCachedPkt	Copies a packet into the cache	Packet *pkp, int ici_addr	SN_FUNC_SUCCESS or SN_FUNC_FAIL
SnCacheRetrieve	Retrieves a packet from the cache	unsigned int src_ip, unsigned int dest_ip, int src_port, int dest_port, int seq_num, Packet **pkpp, int * ici_addr	SN_FUNC_SUCCESS or SN_FUNC_FAIL
snCacheDestroy	Deletes a packet from the cache	unsigned int src_ip, unsigned int dest_ip, int src_port, int dest_port, int seq_num	SN_FUNC_SUCCESS or SN_FUNC_FAIL
snStripTcpAck	If the Ack field in the TCP header is set to 1, set to 0	Packet *pkp	SN_FUNC_SUCCESS or SN_FUNC_FAIL
NewInCache	Determine if the packet already exists in the cache	Packet *pkp	1 if the packet cannot be found in cache, 0 otherwise
snGetTcpInfo	Abstract the TCP	Packet *pkp,	SN_FUNC_SUCCESS

	header information	TcpInfo *fd_p	or SN_FUNC_FAIL

Table 2 Cache Function Calls

5.1.2 Snoop Connection Table

The Snoop Agent maintains a table to keep track of the TCP connections. This table is required because the two ends could have multiple connections established, and each of these connections are required to be maintained separately (because sequence numbers are not unique among the connections).

The following is the record format in the Snoop connection table:

```
typedef struct
 {
   unsigned int    src_ip;   /* Source IP */
   unsigned int    dest_ip;  /* Destination IP */
   int             src_port; /* Source Port */
   int             dest_port; /* Destination Port */
   unsigned int    last_seq_num; /* Last sent seq number */
   unsigned int    last_ack_num; /* Last received seq number */
   int             repeat_ack; /* specify if we have received repeat
ack */
   int             fin_flag;  /* Flag to specify if we have received
fin packet */
   unsigned        fin_seq_num; /* Seq num of the fin packet */
   Evhandle   timeout_evt;  /* Timeout event handle */
 } struct_snTable;
```

Each record identifies a TCP connection by its source and destination IP and port pairs. It also keeps track of the last sequence number and last acknowledgement number recorded for the connection. The timeout_evt is used for the retransmission timer which will be explained in the Timeout State later on.

The following Table contains the function routines related to the Snoop connection table:

Function Name	Description	Input parameters	Return Value
TableInit	Initialize the table	null	null
AddTable	Add a new tcp connection into the table	TcpInfo info, int *TabIdx	1 if successful, 0 otherwise
FindTable	Find a table entry	unsigned int src_ip, unsigned int dest_ip, int src_port, int dest_port, int *TabIdx	1 if successful, 0 otherwise
snTableDestroy	Delete a table record	int tab_idx	1 if successful, 0 otherwise
snExtendTO	Extends the retranmission timer	int tab_idx	1 if successful, 0 otherwise

Table 3 TCP Connection Table Function Calls

5.2 The Init State

The Snoop process begins in the Init state when it is initialized. In the Init state, the cache table (explain in details in section 5.1) and the TCP connection table are initialized

5.3 The Wait State

After the Init State has finished its tasks, the process is forced to transit to the Wait State. There is no task performed in the Wait State except to wait for the following events to arrive:

- packet arrival from the MAC layer.
- packet arrival from the upper TCP/IP layer
- the retransmission timer expires

All events arrive as interrupts. The packet arrivals is represented as a stream interrupt. The retransmission timer expire event is represented as self-scheduled interrupt.

5.4 The Snoop_Data State

When a packet from the higher TCP/IP layer arrives, the process will transit from the Wait State to the Snoop_Data State. The Snoop Agent keeps track of the last sequence number seen from the higher layer. Depending on the sequence number of the arrived packets, the packet is processed in different way.

A new packet in the normal TCP sequence. This is the common case when a new packet with a higher sequence number arrives. The packet is added to the Snoop Cache and is forwarded to the lower layer.

An out-of-sequence packet that has been cached earlier. This happens when lost packets cause timeouts at the sender. If the sequence number is greater than the last acknowledgement seen by Snoop Agent, it is assumed that the packet is lost. The packet is therefore forwarded to the destination. On the other hand, if the sequence number is less than the last acknowledgement, the packet is discarded.

After the packet is processed, the Snoop process transits back to the Wait state to wait for the next coming events.

The following figure shows the flow chart of how the algorithm is implemented:

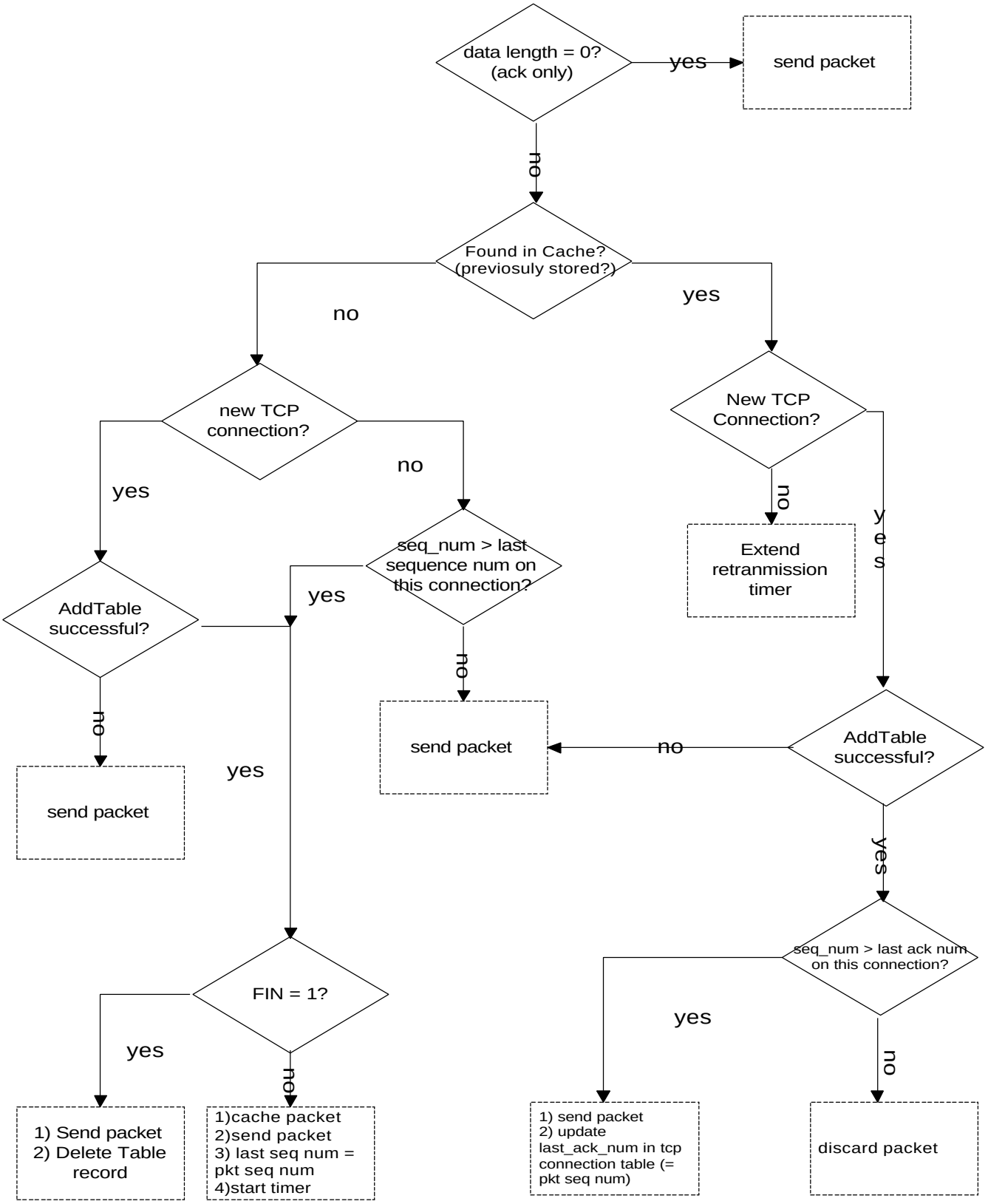

data length = 0? (ack only)
yes
send packet
no
Found in Cache? (previosuly stored?)
no
yes
new TCP connection?
New TCP Connection?
yes
no
yes
no
AddTable successful?
seq_num > last sequence num on this connection?
Extend retranmission timer
yes
yes
no
no
send packet
send packet
AddTable successful?
no
yes
yes
FIN = 1?
seq_num > last ack num on this connection?
yes
yes
no
no
1) Send packet 2) Delete Table record
1)cache packet 2)send packet 3) last seq num = pkt seq num 4)start timer
1) send packet 2) update last_ack_num in tcp connection table (= pkt seq num)
discard packet

Figure 9 Snoop_Data() Implementation flow Chart

Besides implementing the Snoop algorithm mentioned earlier, the implementation also takes care of cleaning the connection table entry when the last packet of the connection (i.e the FIN packet) is detected. In addition, the check at the top of the flow chart is to ensure that the Snoop Agent does not mistakenly cached packets does not contain any data related to the opposite direction (e.g. Acknowledgement packets for acknowledgment data packet in the opposite direction).

5.5 The Snoop_Ack State

When a packet from the MAC layer arrives at the Snoop Agent, the process will transit from the Wait State to the Snoop_Ack State. Depending on the acknowledgement sequence number, the packet is processed in the following way:

A new acknowledgement. This is an acknowledgement packet with an acknowledgement sequence number larger than the last received one. This initiates the cleaning of the corresponding cache entries. The acknowledgement is passed to the higher TCP/IP layer.

A false acknowledgement. This is an acknowledgement packet with an acknowledgement sequence number less than the last received one. This rarely happens and the acknowledgement is discarded.

A duplicate acknowledgement. This is an acknowledgment packet that is identical to the last received one. This causes the Snoop Agent to assume that packets that have been sent

with higher sequence number are lost. The snoop will retransmit all the packets from the first lost one.

The following flow chart shows the steps executed by the Snoop_Ack State:

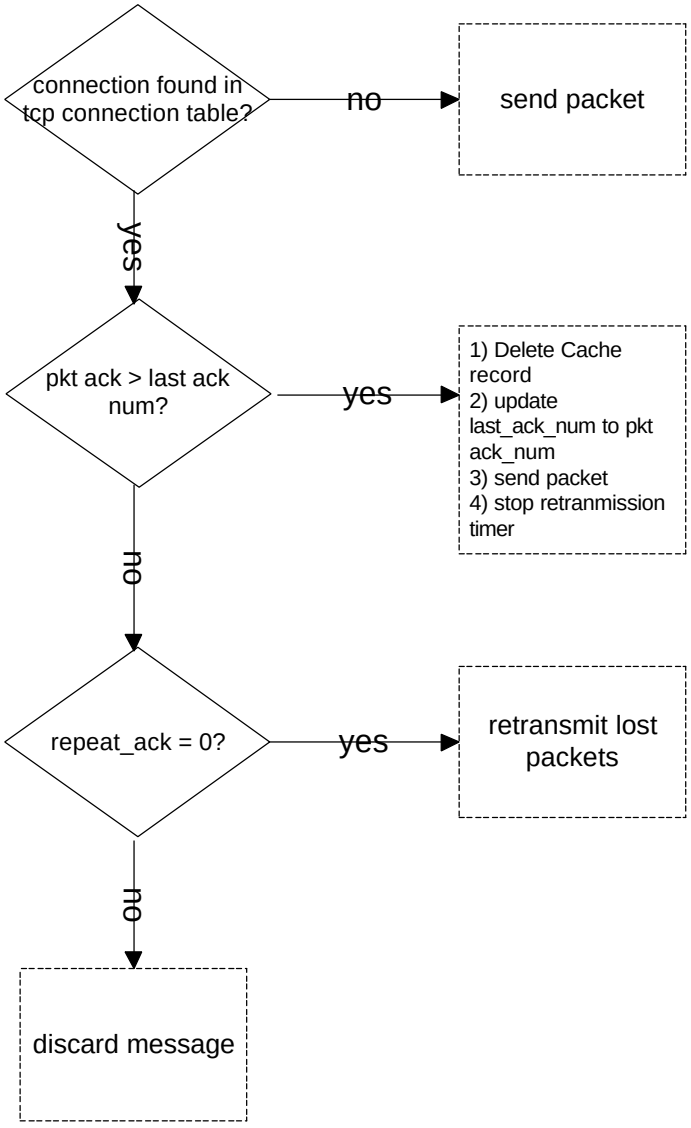


Figure 10 Snoop_Data() Implementation flow Chart

5.6 Timeout State

When the retransmission timer of a connection expires, the process transit from the Wait state to the timeout state. The processing during the timeout state is very similar to that of the processing of receiving a duplicate acknowledgement in the Snoop_Ack state. Essentially, it re-transmits all packets that have not been acknowledged.

The retransmission timer of a connection is extended when a new data packet is received from the upper layer, a new acknowledgement packet is received from the lower layer or a retransmission timer has just expired.

5.7 Model Attributes

Model Attribute	Attribute Type	Explanation
Snoop Enabled	Integer	Set to 1 to enable the Snoop Protocol. Set to 0 to disable it
Retransmission Timeout Period	Double	The persist retransmission timeout period of the Snoop Protocol.

6 Model Statistics

Current Number of packets Cached	Integer	Number of packets cached in the Snoop cache. This number is updated when a packet is added or removed from the cache.

Table 4 shows the model statistics that are collected in the PEG model

Local Statistcs	Statistics Type	Explanation
Total Packet Sent	Integer	Total number of packet sent from upper layer (IP) to lower layer (MAC). This statistics is updated every time a packet is sent.
Sent Packets Dropped	Integer	Total number of packet dropped in the sent direction This statistics is updated every time a packet in that direction is dropped.
Total Packets Received	Integer	Total number of packet pass from the lower layer (MAC) to upper layer (IP). This statistics is updated every time a packet is received.
Sent Packets Received	Integer	Total number of packet dropped in the receive direction This statistics is updated every time a packet in that direction is dropped.
Current Number of packets Cached	Integer	Number of packets cached in the Snoop cache. This number is updated when a packet is added or removed from the cache.

Table 4 Local Statistics of PEG Process Model

7 Performance Comparison

This section describes how are the models developed in section 3 are used to measure the performance.

7.1 Scenario Setup

Three scenarios are setup to verify the operations of the models and to measure the performance of a WLAN device with Snoop protocol compared to a device with unmodified TCP.

7.1.1 Scenario Devices

All scenarios consist of two main types of WLAN nodes: mobile workstations and servers. The Snoop Protocol Model is built into both devices and are enabled/disabled depending on the different scenarios. A device with Snoop Protocol enabled is referred to enhanced device. The devices without the Snoop Protocol enabled are referred to as the original device.

The other nodes that are used in the scenarios are Application Configuration node and the Profile Configuration nodes. These two nodes allow the scenarios to generate traffic using the OpNet Application Model paradigm.

7.1.2 TCP Model Parameters

WLAN nodes in all scenarios use the following TCP parameters for the TCP models.

TCP Parameters	Values
Maximum Segment Size (bytes)	2264
Receiver Buffer Size (bytes)	8760

Receiver Buffer Usage Threshold	0.0
Delayed Ack Mechansim	Segment / Clock Based
Maximum Ack Delay	0.2
Fast Transmit	Enabled
Fast Recovery	Enabled
Selective Ack (SACK)	Disabled
Nagle SWS Algorithm	Disabled
Karn's Algorithm	Enabled
Retransmission Threshold	Attempt Based
Initial RTO	1.0
Minimum RTO	0.5
Maximum RTO	64
RTT Gain	0.125
Deviation Gain	0.25
RTT Deviation Coefficient	4.0
Timer Granularity	0.5
Persist Timeout	1.0

Table 5 TCP Model Parameters

Most of the values are default values from OpNet, and they do not affect the results of the experiments performed in this report. The fact that these values are left as defaults follows the rationale of choosing the Snoop Protocol as PEP: TCP parameters should be left unchanged (section 2.3) on wireless devices such that it can be tuned to handle congestion cases.

The parameters that have direct effects on the results are the Maximum Segment Size and the Receiver Buffer Size. Having smaller values for both parameters will causes more TCP messages to be sent. Since the Snoop Protocol will show the greatest effect when more packets are lost, it is expected that setting these parameters small will increase the performance results measured in this report.

7.1.3 Scenario Traffic

All scenarios use the OpNet Application Model to generate the traffic. In order to study in details about the effects of the Snoop Protocol, a simple traffic with predictable pattern is strongly desired. The chosen traffic is a FTP file transfer of 100000 bytes files.

The following diagram shows a typical message sequence of FTP file transfer.

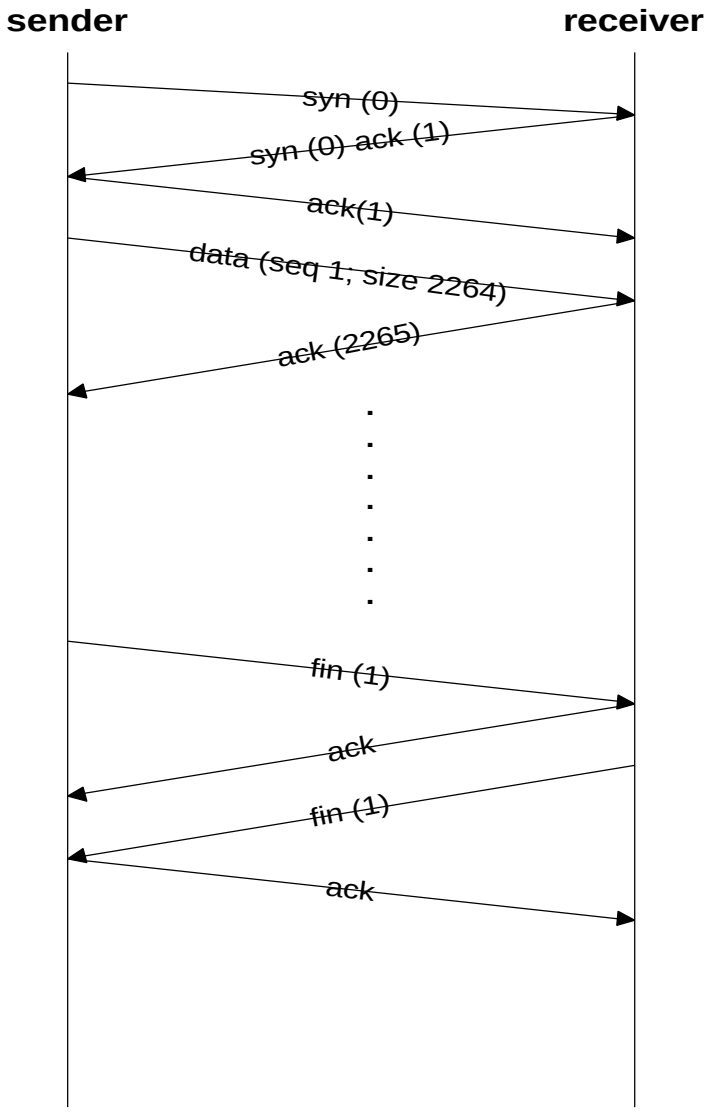


Figure 11 FTP Transfer Message Sequence

During the data transfer portion, each message contains 2264 bytes of data. This value corresponds to the TCP window size for WLAN.

7.1.4 Wireless LAN Model Parameters

All scenarios use the default values for the Wireless LAN parameters. The goal of the scenarios is to study how does an enhanced device performed relative to an original device. Therefore, it is important that all devices use the same Wireless LAN parameters. These various WLAN parameters are set to the values listed in Table 6.

Wireless LAN Parameters	Values
Rts Threshold	None
Fragmentation Threshold	None
Data Rate	1 Mbps
Physical Characteristics	Frequency Hopping
Buffer Size (bits)	256000
Maximum Receive Lifetime (seconds)	0.5

Table 6 Wireless LAN Parameters

The use of the PEG process model inside the device protocol stack minimizes the effects of the Wireless LAN parameters in the results. Therefore, most of the Wireless LAN parameters use the default values that are supplied by OpNet.

7.2 Scenario 1: Single Mobile Upload

The first scenario is designed to verify the basic operation of the Snoop Protocol and to study in details how the Snoop Protocol improves the performance of the network.

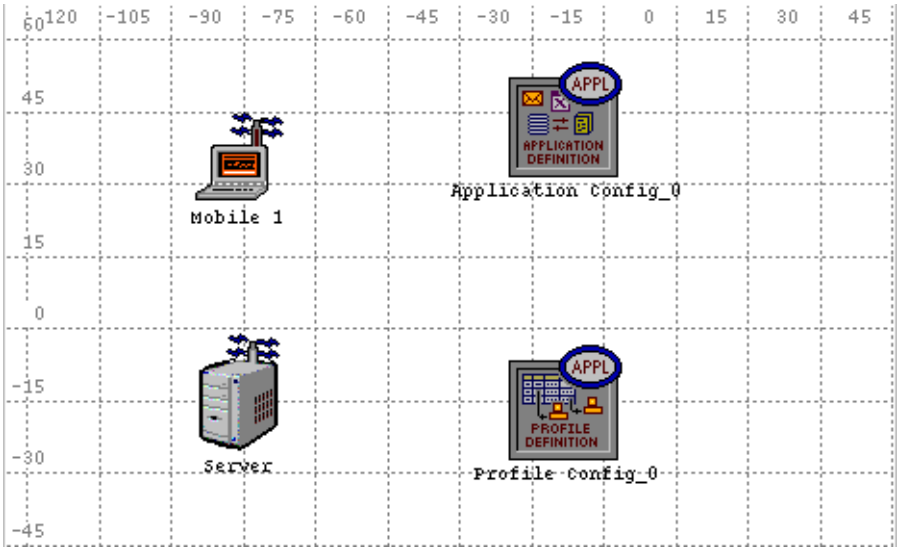


Figure 12 Network Model for Scenario 1

In this scenario, the mobile performs a 100000 bytes FTP upload to the server. The simulation settings are listed below:

Parameters Setup	Values
Snoop Model Attribute in Mobile 1	
Snoop Enabled	1 (Enabled)
Snoop Retransmission Timeout	1 sec
PEG Model Attribute in Mobile 1	
Send_Data_Drop_Rate	Varied
Recv_Data_Drop_Rate	Varied
Snoop Model Attribute in Server	
Snoop Enabled	0 (Disabled)
PEG Model Attribute in Server	
Send_Data_Drop_Rate	0
Recv_Data_Drop_Rate	0

Table 7 Simulation Parameters for Scenario 1

In this scenarios, the mobile acts as the sent file host in Figure 11 and the server acts as the receive file host.

Note that the send_data_drop_rate and the recv_data_drop_rate are set to the identical value in this scenario. This is done to simulate the scenario where both TCP data message and TCP acknowledgement messages can be lost. The values of the send_data_drop_rate and the recv_data_drop rate are referred to as the Packet Error Rate (PER) in later sections.

The value of Snoop Protocol retransmission timeout is sent to the initial RTO values of the TCP models.

7.2.1 Results / Observations

Figure 13 shows the upload response time of the 100,000-byte file transfer. The response time is measured from the first SYN packet sent from Mobile 1 to the last acknowledgement is received for the last FIN packet.

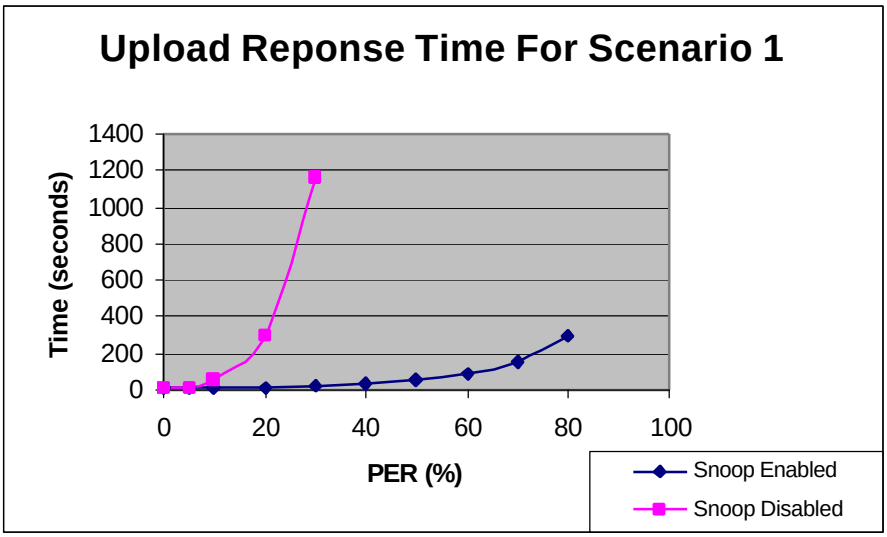


Figure 13. Upload Response Time of 100,000-byte File

It is clearly shown that the performance improvement of snoop protocol increases as Packet Error Rate increases. At the packet lost rate of 30%, the Snoop Protocol improves the performance by 68 times.

This enormous improvement is due to the combination of two effects: Caching Packets and Local Retransmission Timeout.

7.2.1.1 Caching Packets

The Snoop Protocol caches packets, and retransmit them on behalf of the TCP once it detects packets are lost (due to duplicate acknowledgement sequence being received, as described in section 5.5). Figure 14 shows the number of packets cached during the File upload.

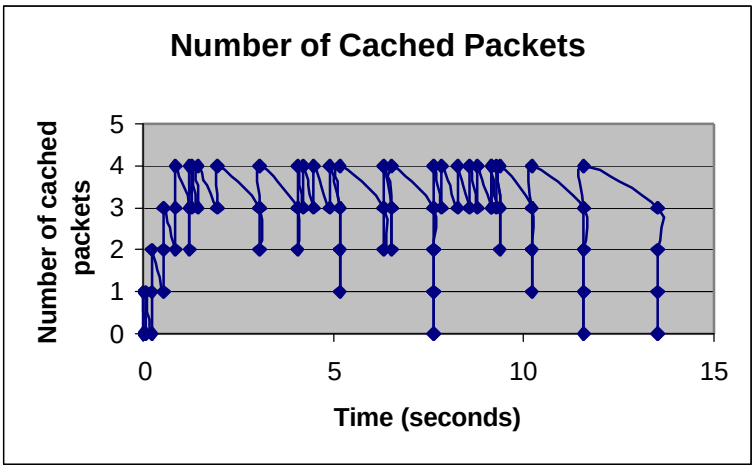


Figure 14. Number of Cache Packets during File Upload

The number of cached packets increases as the mobile transmits packets to the server without receiving acknowledgement packets. It can be seen that maximum number of packets cached is four. The reason is that the mobile can transmit up to four data message before filling up the receive buffer[0]. The number of packets in the cache decreases as the Snoop Protocol receives the acknowledgement of those packets. Another point that is worth noting is that the number of cached packets is zero when the connection finishes. This verifies that all packets are cleared when the connection is finished.

Figure 15 shows the size of the Congestion Windows during file upload at PER of 20%.

[0] The receive window buffer size is 8760 and the maximum segment size is 2264.

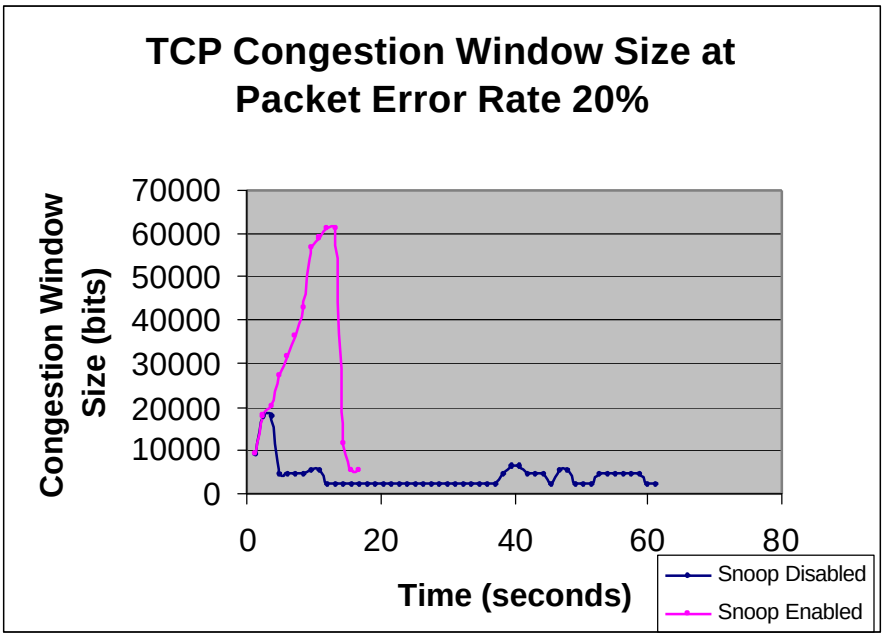


Figure 15 Congestion Windows of the TCP connection at PER 20%

TCP shrinks its congestion windows when it detects that packets are lost. The reason for this action is because TCP assumes packets lost are due to congestion in the network. While this is often true in wired line system, it is not true in wireless system. In wireless systems, packets lost are mostly due to the bit errors resulting from the effects of environmental impact (background noise, Doppler effect, Multipath delay).

The effect of caching packet is that it prevents TCP congestion windows from shrinking. It is the most critical factors in improving the performance of data transfer. Shrinking the congestion windows has a “cyclic” effect. Shrinking the congestion windows causes the TCP to sent less data in each packet, which means sending more data packets across the network. Since sending more data packets increases the chances of packets getting

corrupted, this in turn will shrink the congestion windows more and the cycle repeats itself again.

Figure 16 shows the sequence number (recorded at the TCP layer) that is sent across the network during the file transfer. The sequence number ends at 100,001 because the sequence number starts from 0, and is incremented for every byte of data sent across. In addition, one extra byte for the SYN packet and one for the FIN packet.

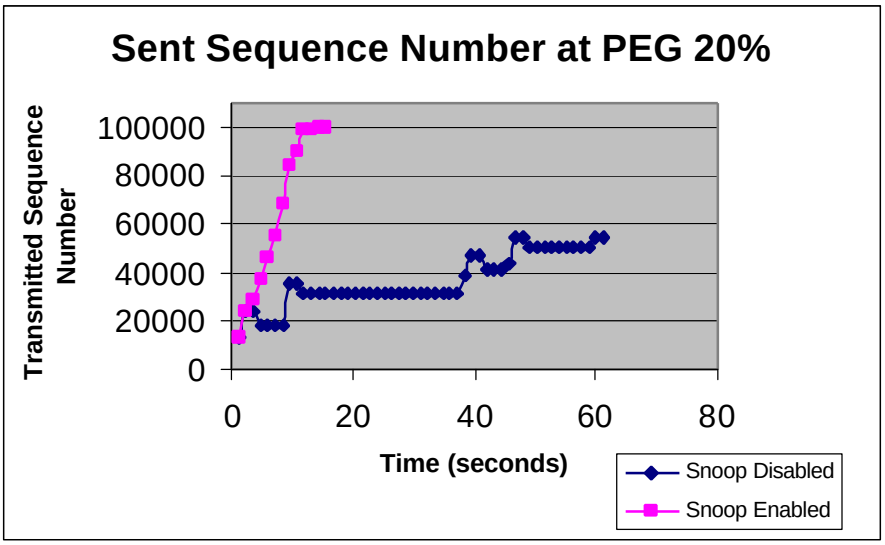


Figure 16 Sent Sequence Number

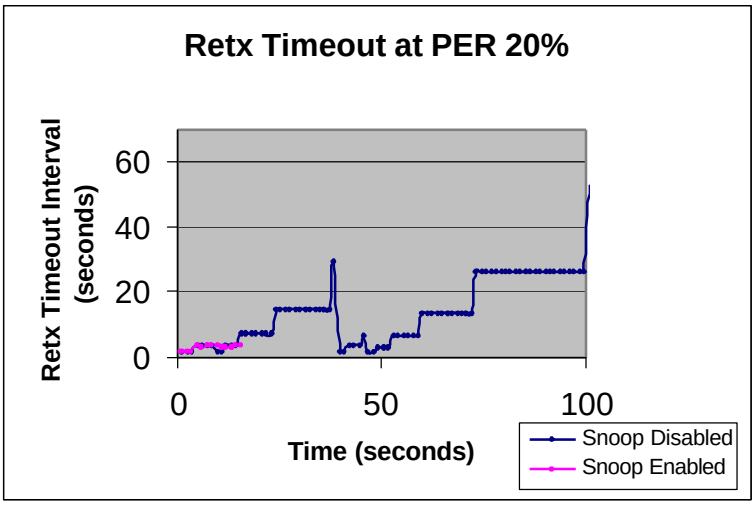
It is clear that without the Snoop Protocol, the TCP layer has to retransmit the same sequence number (hence same data bytes) several times.

7.2.1.2 Local Retransmission Timeout

The retransmission timeout mechanism in TCP is designed to handle network congestion, therefore, the TCP timeout values are modified according to the level of network congestion, which is inferred by the amount of packet loss. Similar to the congestion

windows size, this assumption is often not true in case wireless system. The Snoop Protocol that is implemented use persist retransmission timeout period of 1 seconds. The Snoop Protocol increases the performance by using a more appropriate timeout period[0] for the wireless link, and prevents the TCP model from changing its own retransmission timeout period due to packet loss.

Figure 17 shows the changes of the retransmission timeout period recorded from the TCP layer.



[0] The timeout period can be tuned to the wireless link characteristics by employ some form of algorithms, such as Karn's algorithm, for updating the period over time. However, what is important is that the Snoop Protocol allows a retransmission timeout value to be adjusted only for the wireless link.

Figure 17 Retransmission Timeout of TCP Models for File Upload at PER 20%

Due to the result of caching packets, the Snoop Protocol prevents the TCP model from incorrectly adjusting the retransmission timeout due to packet lost.

7.3 Scenario 2: Multiple Mobiles Download

This scenario is designed to verify that the Snoop Protocol can handle multiple connections simultaneously. In this scenario, the Snoop Protocol in the server is enabled. The OpNet Application Model is configured such that the each mobile downloads a 100,000 bytes file from the server. Figure 18 shows the network model for this scenario.

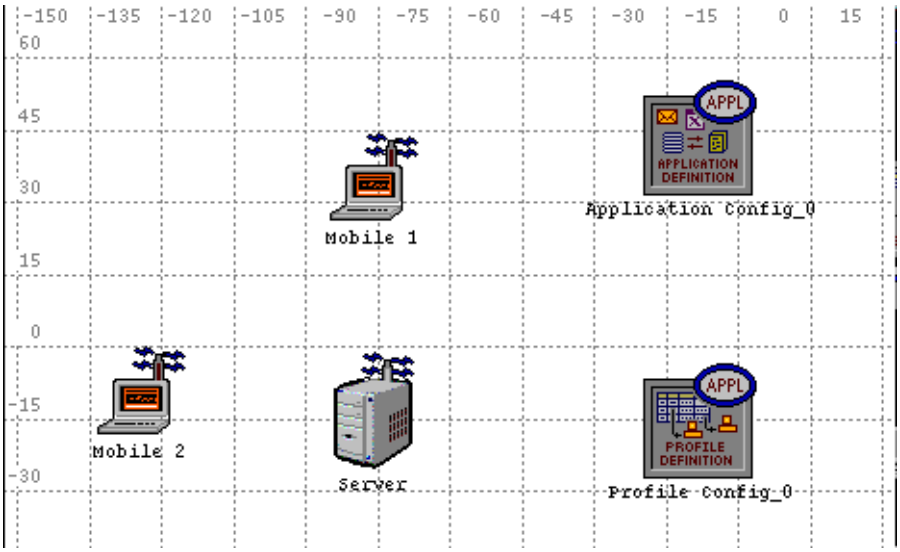


Figure 18. Network Model for Scenario 2

In this scenario, each mobile performs a 100,000 bytes FTP download from the server. The simulation settings are listed below:

Parameters Setup	Values
Mobile 1	
Snoop Enabled	0 (Disabled)
PEG Model Attribute in Mobile 1	
Send_Data_Drop_Rate	0
Recv_Data_Drop_Rate	0
Mobile 2	
Snoop Enabled	0 (Disabled)
PEG Model Attribute in Mobile 2	
Send_Data_Drop_Rate	0
Recv_Data_Drop_Rate	0
Server	
Snoop Enabled	1 (Enabled)
Snoop Retransmission Timeout (seconds)	1
PEG Model Attribute in Server	
Send_Data_Drop_Rate	Varied
Recv_Data_Drop_Rate	Varied

Table 8 Simulation Parameters for Scenario 2

7.3.1 Results / Observations

Figure 19 shows the average download time of the two mobiles

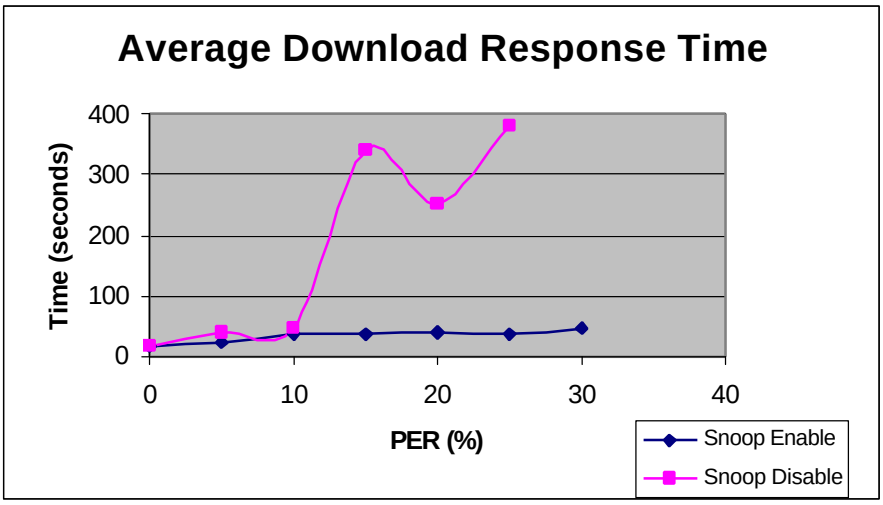


Figure 19 Average Download Response Time of 100,000 bytes File

Similar to the previous scenario, the Snoop Protocol improves the performance of the 100,000 bytes file. The fact that the mobile can download simultaneously verifies that the Snoop Protocol can handle multiple connections simultaneously. The drop in response time of the original mobile between PER of 15% to 20% is probably caused by the random nature of the way the packets are dropped in the PEG process model.

It can be noticed that the average download time of the two mobiles are longer than the upload time shown in scenario 1 (For PER of 0%, scenario 1 needs 9.4 seconds, where scenario 2 needs 17 seconds). It is because the two mobiles are competing resources on the radio link. The effect of this is shown again in scenario 3.

7.4 Scenario 3: Multiple Mobiles Upload

This scenario is designed to show the effect of having an enhanced mobile in the same network with an original mobile

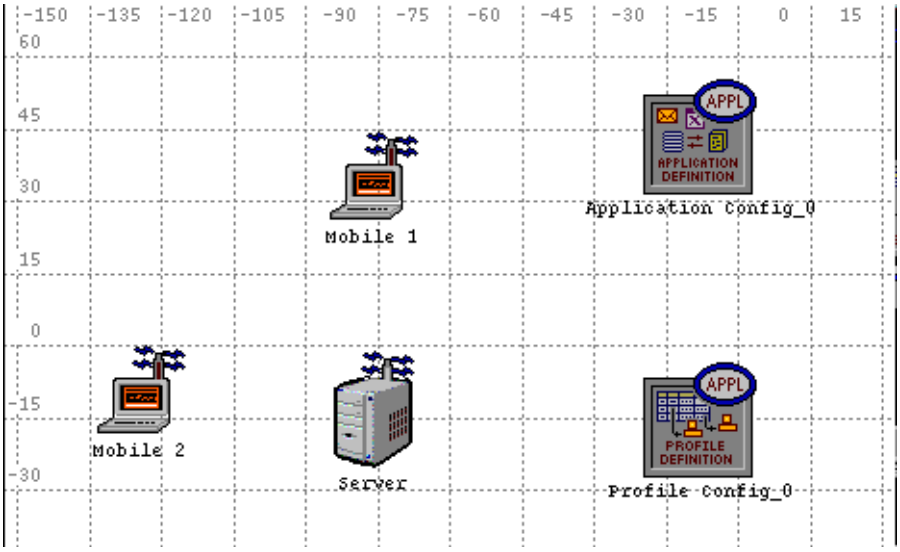


Figure 20. Network Model for Scenario 3

Mobile 1 is configured with Snoop Protocol disabled, and mobile 2 is configured with snoop enabled. The Application Model is configured to have the both mobile to upload a 100,000 bytes file to the server.

The simulation settings are listed below:

Parameters Setup	Values
Mobile 1	
Snoop Enabled	0 (Disabled)
PEG Model Attribute in Mobile 1	
Send_Data_Drop_Rate	0
Recv_Data_Drop_Rate	0
Mobile 2	
Snoop Enabled	1 (Enabled)
Snoop Retransmission Timeout (seconds)	1.0

PEG Model Attribute in Mobile 2	
Send_Data_Drop_Rate	0
Recv_Data_Drop_Rate	0
Server	
Snoop Enabled	0 (Disabled)
PEG Model Attribute in Server	
Send_Data_Drop_Rate	Varied
Recv_Data_Drop_Rate	Varied

Table 9 Simulation Parameters for Scenario 3

7.4.1 Results / Observations

The upload response time of the two mobiles are shown in Figure 21.

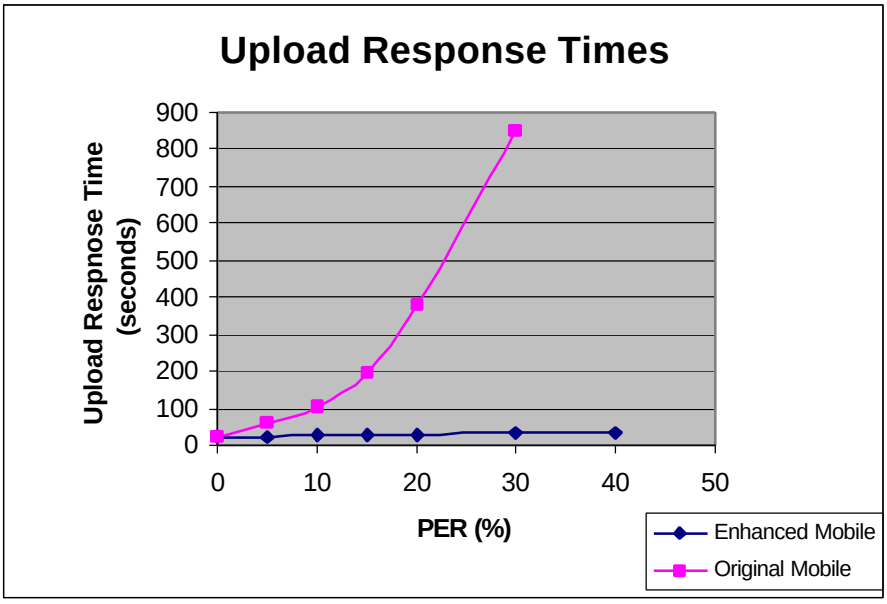


Figure 21 Upload Response Time (seconds) of the Mobiles in Scenario 3.

As expected, the mobile with Snoop Protocol outperforms the mobile without Snoop Protocol. It is interesting to note that the time required for the uploading 100,000 bytes file in this scenario for Mobile 2 (Snoop Protocol disabled) is longer than that required in scenario 2. This suggests that the Mobile 1 (Snoop Enabled) takes a larger piece of the bandwidth available (it retransmits more often) and hence reduces the bandwidth that can be allocated to Mobile 2.

8 Conclusion

The results in this report indicate that the Snoop Protocol is implemented correctly (can support multiple connection that the same time) and can significantly increase the performance of the TCP transfer (68 times at packet error rate of 30%) without changing other layers in the protocol stack. In addition, the results show that the Snoop Protocol improves the performance by preventing the transport layer, i.e. TCP layer, from reducing the congestion window size, hence significantly increase the performance.

In addition, the result also show that devices with the Snoop Protocol (enhanced device) can co-exist with that those without the Snoop Protocol (original device). However, when transfer data simultaneously, the performance of the original device will get affected because the enhanced device is more active in completing for resources (retransmit more often) than the original device.

9 Difficulty and Challenges in Doing the Project

Most of the work in this project is involved in designing and developing the Snoop Agent model and the PEG model. Because the Snoop Agent and the PEG models are not based on any existing model, both models have to be designed and implemented from scratch. As a result, significant amount of time is spent in developing and debugging the code for the two models

The second difficulty lies in integrating the two models into the existing WLAN device stack. Since the two new models are added between the ARP and the IP node models, the ARP and the IP node models have to be separated. However, the ARP and IP models are tightly coupled to each other (the two models use op_tan functions to determine their the connection between them), the ARP model has to be modified in order to separate the two. This involves studying the code of the ARP model and the IP model, and determines how they can be separated.

10 Future Work

Future work would be to implement a mechanism to calculate the retransmission timer value based on the round-trip delay estimated from the time of sending a data packet to the time the acknowledgement is received. Currently, only the retransmission timer value is set to a fixed value (1.0 second in the experiment performed).

11 References

[1] IEEE 802.11 Workgroup

http://grouper.ieee.org/groups/802/11/index.html

[2] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, Performance Enchancing Proxy (PEP) Request for Comments (RFC)

http://community.roxen.com/developers/idocs/drafts/draft-ietf-pilc-pep-04.html

[3] Hari Balakrishnan, Srinivasan Seshan, Elan Amir and Randy H. Katz, Improving TCP/IP Performance over Wireless Networks

http://www.cs.berkeley.edu/~ss/papers/mobicom95/html/mobicom-final.html

[4] W.Richard Stevens, TCP/IP Illustrated Volume 1, Addison Wesley, Professional Computing Series, 1984

[5] Andrew S. Tanenbaum, Computer Networks Third Edition, Prentice-Hall Press, 1996

[6] Wireless LAN Model Description, Opnet Manual.

12 Appendix

12.1 Packet Error Generator Source Code

```
/* Process model C form file: peg.pr.c */
/* Portions of this file copyright 1992-2000 by OPNET, Inc. */



/* This variable carries the header into the object file */
static const char peg_pr_c [] = "MIL_3_Tfile_Hdr_ 70B 30A op_runsim 7 3ABE8A73 3ABE8A73 1
kelvin chihon 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 <nato.h>
#include <oms_pr.h>
#include <oms_tan.h>
#include <ip_addr_v4.h>
#include <ip_rte_v4.h>
#include <ip_dgram_sup.h>
#include <tcp_seg_sup.h>

/* Function Return Constants */
#define SN_FUNC_SUCCESS 0
#define SN_FUNC_FAILED  1
#define SN_FUNC_CACHE_FULL 10

/* Define a transition condition corresponding */
/* to the IP datagram arrival                  */
#define HOST_PKT_ARRIVAL (intrpt_type == OPC_INTRPT_STRM &&  intrpt_strm == 2)
#define MOB_PKT_ARRIVAL (intrpt_type == OPC_INTRPT_STRM && intrpt_strm == 1)

/* Function return code */
#define PEG_FUNC_SUCCESS 0
#define PEG_FUNC_FAILED  1

/* TCP Information Structure */
typedef struct
{
    unsigned int  src_ip;
unsigned int  dest_ip;
```

```
    int                             src_port;
        int                             dest_port;
        unsigned int    seq_num;
        unsigned int    ack_num;
        unsigned int    rcv_win;
        int                             urgent_pointer;
        int                             data_len;
        int                             urg;
        int                             ack;
        int                             push;
        int                             rst;
        int                             syn;
        int                             fin;
} TcpInfo;

int pegGetTcpInfo (Packet *pkp,  TcpInfo *tcp_infop);

/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undefBIN
#undefBOUT
#defineBINFIN_LOCAL_FIELD(last_line_passed) = __LINE__ - _block_origin;
#defineBOUTBIN
#defineBINITFIN_LOCAL_FIELD(last_line_passed) = 0; _block_origin = __LINE__;
#else
#defineBINIT
#endif /* #if !defined (VOSD_NO_FIN) */



/* State variable definitions */
typedef struct
{
/* Internal state tracking for FSM */
FSM_SYS_STATE
/* State Variables */
Ici*                    send_iciptr;
int                     send_total_pkcnt;
int                     recv_total_pkcnt;
int                     send_drop_pkcnt;
int                     recv_drop_pkcnt;
Stathandle              send_total_pkcnt_stathandle;
Stathandle              send_drop_pkcnt_stathandle;
Stathandle              recv_total_pkcnt_stathandle;
Stathandle              recv_drop_pkcnt_stathandle;
Ici*                    recv_iciptr;
int                     send_drop_rate;
int                     recv_drop_rate;
} peg_state;

#define pr_state_ptr            ((peg_state*) SimI_Mod_State_Ptr)
#define send_iciptr             pr_state_ptr->send_iciptr
#define send_total_pkcnt        pr_state_ptr->send_total_pkcnt
#define recv_total_pkcnt        pr_state_ptr->recv_total_pkcnt
#define send_drop_pkcnt         pr_state_ptr->send_drop_pkcnt
#define recv_drop_pkcnt         pr_state_ptr->recv_drop_pkcnt
#define send_total_pkcnt_stathandlepr_state_ptr->send_total_pkcnt_stathandle
#define send_drop_pkcnt_stathandlepr_state_ptr->send_drop_pkcnt_stathandle
```

```
#define recv_total_pkcnt_stathandlepr_state_ptr->recv_total_pkcnt_stathandle
#define recv_drop_pkcnt_stathandlepr_state_ptr->recv_drop_pkcnt_stathandle
#define recv_iciptr             pr_state_ptr->recv_iciptr
#define send_drop_rate          pr_state_ptr->send_drop_rate
#define recv_drop_rate          pr_state_ptr->recv_drop_rate

/* 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_PREAMBLEpeg_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };
int pegGetTcpInfo (Packet *pkp,  TcpInfo *tcp_infop)
{
Packet *encap_tcp_pktp;
TcpT_Seg_Fields *tcp_fdp;
IpT_Dgram_Fields *ip_fdp;

 op_pk_nfd_access (pkp, "fields", &ip_fdp);

tcp_infop->src_ip = ip_fdp->src_addr;
tcp_infop->dest_ip = ip_fdp->dest_addr;


/* Check if the packet is TCP packet */

/* Get TCP Packet contained inside the IP packet */
if (op_pk_nfd_get (pkp, "data", &encap_tcp_pktp) == OPC_COMPCODE_FAILURE)
{
     printf  ("Unable to extract TCP packet from ip");
return PEG_FUNC_FAILED;

}
else
{
    /* Extract Tcp Info */
op_pk_nfd_access (encap_tcp_pktp, "fields", &tcp_fdp);

tcp_infop->src_port = tcp_fdp->src_port;
tcp_infop->dest_port = tcp_fdp->dest_port;
tcp_infop->seq_num = tcp_fdp->seq_num;
tcp_infop->ack_num = tcp_fdp->ack_num;
tcp_infop->rcv_win = tcp_fdp->rcv_win;
tcp_infop->urgent_pointer = tcp_fdp->urgent_pointer;
tcp_infop->data_len = tcp_fdp->data_len;
tcp_infop->urg = tcp_fdp->urg;
tcp_infop->ack = tcp_fdp->ack;
tcp_infop->push = tcp_fdp->push;
tcp_infop->rst = tcp_fdp->rst;
tcp_infop->syn = tcp_fdp->syn;
tcp_infop->fin = tcp_fdp->fin;

   op_pk_nfd_set (pkp, "data", encap_tcp_pktp);

return PEG_FUNC_SUCCESS;
```

```
}
}

/* End of Function Block */

#if defined (__cplusplus)
extern "C" {
#endif
void peg (void);
Compcode peg_init (void **);
void peg_diag (void);
void peg_terminate (void);
void peg_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif



/* Process model interrupt handling procedure */


void
peg (void)
{
int _block_origin = 0;
FIN (peg ());
if (1)
{
Packet*pkptr;
int intrpt_type = OPC_INT_UNDEF;
int intrpt_strm = OPC_INT_UNDEF;
double rand_val;
TcpInfo tcpinfo;


FSM_ENTER (peg)

FSM_BLOCK_SWITCH
{
/*---------------------------------------------------------*/
/** state (wait) enter executives **/
FSM_STATE_ENTER_UNFORCED (0, state0_enter_exec, "wait", "peg () [wait enter execs]")
{
}


/** blocking after enter executives of unforced state. **/
FSM_EXIT (1,peg)


/** state (wait) exit executives **/
FSM_STATE_EXIT_UNFORCED (0, state0_exit_exec, "wait", "peg () [wait exit execs]")
{
/* Obtain interrupt parameters.*/
intrpt_type = op_intrpt_type ();
if (intrpt_type == OPC_INTRPT_STRM)
{
intrpt_strm = op_intrpt_strm ();
```

```
}

}


/** state (wait) transition processing **/
FSM_INIT_COND (HOST_PKT_ARRIVAL)
FSM_TEST_COND (MOB_PKT_ARRIVAL)
FSM_TEST_LOGIC ("wait")

FSM_TRANSIT_SWITCH
{
FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;)
FSM_CASE_TRANSIT (1, 3, state3_enter_exec, ;)
}
/*---------------------------------------------------------*/



/** state (INIT) enter executives **/
FSM_STATE_ENTER_FORCED (1, state1_enter_exec, "INIT", "peg () [INIT enter execs]")
{
send_total_pkcnt_stathandle = op_stat_reg ("send_total_pkcnt", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
recv_total_pkcnt_stathandle = op_stat_reg ("recv_total_pkcnt", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
send_drop_pkcnt_stathandle = op_stat_reg ("send_drop_pkcnt", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
recv_drop_pkcnt_stathandle = op_stat_reg ("recv_drop_pkcnt", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
send_total_pkcnt = 0;
recv_total_pkcnt = 0;
send_drop_pkcnt = 0;
recv_drop_pkcnt = 0;
op_ima_obj_attr_get (op_id_self(), "peg_send_drop_rate", &send_drop_rate);
op_ima_obj_attr_get (op_id_self(), "peg_recv_drop_rate", &recv_drop_rate);

op_stat_write(send_total_pkcnt_stathandle, send_total_pkcnt);
op_stat_write(recv_total_pkcnt_stathandle, recv_total_pkcnt);
op_stat_write(send_drop_pkcnt_stathandle, recv_drop_pkcnt);
op_stat_write(recv_drop_pkcnt_stathandle, recv_drop_pkcnt);

printf ("Send Drop Rate: %d\n", send_drop_rate);
printf ("Recv Drop Rate: %d\n", recv_drop_rate);
}


/** state (INIT) exit executives **/
FSM_STATE_EXIT_FORCED (1, state1_exit_exec, "INIT", "peg () [INIT exit execs]")
{
}


/** state (INIT) transition processing **/
FSM_TRANSIT_FORCE (0, state0_enter_exec, ;)
/*---------------------------------------------------------*/



/** state (host_pkt) enter executives **/
```

```
FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "host_pkt", "peg () [host_pkt enter
execs]")
{
if (intrpt_type == OPC_INTRPT_STRM)
{
/* Get packet */
pkptr = op_pk_get (op_intrpt_strm ());

/* Get ICI */
send_iciptr = op_intrpt_ici();

if (send_iciptr != OPC_NIL)
{
      op_ici_install (send_iciptr);
}

   send_total_pkcnt++;

   op_stat_write(send_total_pkcnt_stathandle, send_total_pkcnt);

   rand_val = op_dist_uniform (100);


   pegGetTcpInfo (pkptr, &tcpinfo);

   if ( (tcpinfo.syn == 1 )|| ( tcpinfo.fin == 1) )
   {
   /* Don't drop connection control packets */
   op_pk_send (pkptr, 1);
}

   /* Pick a number, see if we should send or not */
   else if ( send_drop_rate < (int) rand_val)
   {
       /* Send packet */
       op_pk_send (pkptr, 1);
   }
   else
   {
   /* Don't send packet, upate counter */
   /* and destroy packet */
   send_drop_pkcnt++;

printf ("(send_total_pkcnt == %d) && \n", send_total_pkcnt);
op_stat_write(send_drop_pkcnt_stathandle,
   send_drop_pkcnt);

op_pk_destroy (pkptr);
   }


    op_ici_install (OPC_NIL);

}


}


/** state (host_pkt) exit executives **/
```

```
FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "host_pkt", "peg () [host_pkt exit execs]")
{
}


/** state (host_pkt) transition processing **/
FSM_TRANSIT_FORCE (0, state0_enter_exec, ;)
/*---------------------------------------------------------*/



/** state (mob_pkt) enter executives **/
FSM_STATE_ENTER_FORCED (3, state3_enter_exec, "mob_pkt", "peg () [mob_pkt enter execs]")
{
if (intrpt_type == OPC_INTRPT_STRM)
{
/* Get packet */
pkptr = op_pk_get (op_intrpt_strm ());

/* Get ICI */
recv_iciptr = op_intrpt_ici();

if (recv_iciptr != OPC_NIL)
{
      op_ici_install (recv_iciptr);
}

recv_total_pkcnt++;

op_stat_write(recv_total_pkcnt_stathandle, recv_total_pkcnt);

pegGetTcpInfo (pkptr, &tcpinfo);

   if ( (tcpinfo.syn == 1 )|| ( tcpinfo.fin == 1) )
   {
   /* Don't drop connection control packets */
   op_pk_send (pkptr, 2);
}

  /* Pick a number, see if we should send or not */
   else if ( recv_drop_rate < (int) op_dist_uniform (100) )
   {

       /* Send packet */
       op_pk_send (pkptr, 2);
   }
   else
   {
   /* Don't send packet, upate counter */
   /* and destroy packet */
   recv_drop_pkcnt++;

printf ("Loss ack packet \n");

op_stat_write(recv_drop_pkcnt_stathandle,
   recv_drop_pkcnt);

op_pk_destroy (pkptr);
   }
```

```
   op_ici_install (OPC_NIL);

}

}


/** state (mob_pkt) exit executives **/
FSM_STATE_EXIT_FORCED (3, state3_exit_exec, "mob_pkt", "peg () [mob_pkt exit execs]")
{
}


/** state (mob_pkt) transition processing **/
FSM_TRANSIT_FORCE (0, state0_enter_exec, ;)
/*---------------------------------------------------------*/



}


FSM_EXIT (1,peg)
}
}

#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
peg_init (void ** gen_state_pptr)
{
int _block_origin = 0;
static VosT_Addressobtype = OPC_NIL;

FIN (peg_init (gen_state_pptr))

if (obtype == OPC_NIL)
{
/* Initialize memory management */
if (Vos_Catmem_Register ("proc state vars (peg)",
sizeof (peg_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 */
((peg_state *)(*gen_state_pptr))->current_block = 2;

FRET (OPC_COMPCODE_SUCCESS)
}
}



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




void
peg_terminate (void)
{
int _block_origin = __LINE__;

FIN (peg_terminate (void))

if (1)
{
Packet*pkptr;
int intrpt_type = OPC_INT_UNDEF;
int intrpt_strm = OPC_INT_UNDEF;
double rand_val;
TcpInfo tcpinfo;

/* 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 peg_svar function. */
#undef send_iciptr
#undef send_total_pkcnt
#undef recv_total_pkcnt
#undef send_drop_pkcnt
#undef recv_drop_pkcnt
#undef send_total_pkcnt_stathandle
#undef send_drop_pkcnt_stathandle
#undef recv_total_pkcnt_stathandle
#undef recv_drop_pkcnt_stathandle
#undef recv_iciptr
#undef send_drop_rate
#undef recv_drop_rate
```

```
void
peg_svar (void * gen_ptr, const char * var_name, char ** var_p_ptr)
{
peg_state*prs_ptr;

FIN (peg_svar (gen_ptr, var_name, var_p_ptr))

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

if (strcmp ("send_iciptr" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->send_iciptr);
FOUT;
}
if (strcmp ("send_total_pkcnt" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->send_total_pkcnt);
FOUT;
}
if (strcmp ("recv_total_pkcnt" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->recv_total_pkcnt);
FOUT;
}
if (strcmp ("send_drop_pkcnt" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->send_drop_pkcnt);
FOUT;
}
if (strcmp ("recv_drop_pkcnt" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->recv_drop_pkcnt);
FOUT;
}
if (strcmp ("send_total_pkcnt_stathandle" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->send_total_pkcnt_stathandle);
FOUT;
}
if (strcmp ("send_drop_pkcnt_stathandle" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->send_drop_pkcnt_stathandle);
FOUT;
}
if (strcmp ("recv_total_pkcnt_stathandle" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->recv_total_pkcnt_stathandle);
FOUT;
}
if (strcmp ("recv_drop_pkcnt_stathandle" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->recv_drop_pkcnt_stathandle);
```

```
FOUT;
}
if (strcmp ("recv_iciptr" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->recv_iciptr);
FOUT;
}
if (strcmp ("send_drop_rate" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->send_drop_rate);
FOUT;
}
if (strcmp ("recv_drop_rate" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->recv_drop_rate);
FOUT;
}
*var_p_ptr = (char *)OPC_NIL;

FOUT;
}
```

12.2 Snoop Protocol Module Source Code

```
/* Process model C form file: snoop_recv.pr.c */
/* Portions of this file copyright 1992-2000 by OPNET, Inc. */
/* This variable carries the header into the object file */
static const char snoop_recv_pr_c [] = "MIL_3_Tfile_Hdr_ 70B 30A opnet 7 3AC6DBB9
3AC6DBB9 1 kelvin chihon 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 <nato.h>
#include <oms_pr.h>
#include <oms_tan.h>
#include <ip_addr_v4.h>
#include <ip_rte_v4.h>
#include <ip_dgram_sup.h>
#include <tcp_seg_sup.h>
#define SNOOP_ON 1
/* Function Return Constants */
#define SN_FUNC_SUCCESS 0
#define SN_FUNC_FAILED  1
#define SN_FUNC_CACHE_FULL 10
/* Define a transition condition corresponding */
/* to the IP datagram arrival                  */
#define HOST_PKT_ARRIVAL (intrpt_type == OPC_INTRPT_STRM &&  intrpt_strm == 0)
#define MOB_PKT_ARRIVAL (intrpt_type == OPC_INTRPT_STRM && intrpt_strm == 1)
#define RETX_TIMEOUT    (intrpt_type == OPC_INTRPT_SELF)
/* Cache Size */
```

```
#define SNOOP_CACHE_SIZE   100
/* Snoop Table */
/* table size */
#define SNOOP_TABLE_SIZE   100
/* indicate whether a record in the table is used or free */
#define FREE 0
#define USED 1
/* TCP Information Structure */
typedef struct
{
    unsigned int  src_ip;
unsigned int  dest_ip;
    int                              src_port;
        int                              dest_port;
        unsigned int    seq_num;
        unsigned int    ack_num;
        unsigned int    rcv_win;
        int                              urgent_pointer;
        int                              data_len;
        int                              urg;
        int                              ack;
        int                              push;
        int                              rst;
        int                              syn;
        int                              fin;
} TcpInfo;
/* Global Structure for Cache */
typedef struct
{
       Packet *cached_pkp[SNOOP_CACHE_SIZE];
   int ici_addr[SNOOP_CACHE_SIZE];
   int max_cached;   /* Maximum number of packet that has been cached at the same time */
   int num_cached;   /* Total number of packets that has been cached */
   int num_removed;   /* Total number of packets that has been removed */
   int curr_cached;  /* Current number of packets cached */
   } struct_snCache;

/* Snoop Table record */
typedef struct
{
 unsigned int    src_ip;
unsigned int    dest_ip;
    int             src_port;
        int             dest_port;
        unsigned int    last_seq_num;
        unsigned int    last_ack_num;
int             repeat_ack;
int             fin_flag;
unsigned        fin_seq_num;
Evhandle   timeout_evt;  /* Timeout event handle */
} struct_snTable;
/* Local Function Declaration */
int snGetTcpInfo (Packet *pkp, TcpInfo *fd_p);
void snCacheInit (void);
int snCachedPkt (Packet *pkp, int ici_addr);
int snCacheRetrieve (unsigned int src_ip,
 unsigned int dest_ip,
 int src_port,
 int dest_port,
                 int seq_num,
```

```
 Packet **pkpp,
 int * ici_addr);
int snCacheDestroy (unsigned int src_ip,
                unsigned int dest_ip,
int src_port,
int dest_port,
int seq_num);
int snStripTcpAck (Packet *pkp);
int NewInCache(Packet *pkp);
int AddTable(TcpInfo info, int *TabIdx);
void TableInit();
int FindTable(unsigned int src_ip,
          unsigned int dest_ip,
  int src_port,
  int dest_port,
  int *TabIdx);
void snExtendTO (int tab_idx);
TcpInfo tcpinfo;
TcpInfo tcpinfo2;
struct_snCache snoop_cache;
struct_snTable snoop_table[SNOOP_TABLE_SIZE];
int table_free = SNOOP_TABLE_SIZE;
int tab_idx = 0;
Ici * iciptr;
int rv;

/* End of Header Block */
#if !defined (VOSD_NO_FIN)
#undefBIN
#undefBOUT
#defineBINFIN_LOCAL_FIELD(last_line_passed) = __LINE__ - _block_origin;
#defineBOUTBIN
#defineBINITFIN_LOCAL_FIELD(last_line_passed) = 0; _block_origin = __LINE__;
#else
#defineBINIT
#endif /* #if !defined (VOSD_NO_FIN) */
/* State variable definitions */
typedef struct
{
/* Internal state tracking for FSM */
FSM_SYS_STATE
/* State Variables */
Ici*                    snoop_host_iciptr;
int                     host_pkcnt;
int                     mob_pkcnt;
Stathandle              host_pkcnt_stathandle;
Stathandle              mob_pkcnt_stathandle;
Ici*                    snoop_mob_iciptr;
int                     ici_next_addr;
int                     snoop_enable;
Stathandle              curr_cached_stathandle;
double                  timeout_interval;
} snoop_recv_state;

#define pr_state_ptr            ((snoop_recv_state*) SimI_Mod_State_Ptr)
#define snoop_host_iciptr       pr_state_ptr->snoop_host_iciptr
#define host_pkcnt              pr_state_ptr->host_pkcnt
#define mob_pkcnt               pr_state_ptr->mob_pkcnt
#define host_pkcnt_stathandle   pr_state_ptr->host_pkcnt_stathandle
#define mob_pkcnt_stathandle    pr_state_ptr->mob_pkcnt_stathandle
```

```
#define snoop_mob_iciptr        pr_state_ptr->snoop_mob_iciptr
#define ici_next_addr           pr_state_ptr->ici_next_addr
#define snoop_enable            pr_state_ptr->snoop_enable
#define curr_cached_stathandle  pr_state_ptr->curr_cached_stathandle
#define timeout_interval        pr_state_ptr->timeout_interval

/* 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_PREAMBLEsnoop_recv_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };
/* Debug message functions */


/* snGetTcpInfo */
/* Extract TCP info from a packet */
int snGetTcpInfo (Packet *pkp,  TcpInfo *tcp_infop)
{
Packet *encap_tcp_pktp;
TcpT_Seg_Fields *tcp_fdp;
IpT_Dgram_Fields *ip_fdp;

 op_pk_nfd_access (pkp, "fields", &ip_fdp);

tcp_infop->src_ip = ip_fdp->src_addr;
tcp_infop->dest_ip = ip_fdp->dest_addr;


/* Check if the packet is TCP packet */

/* Get TCP Packet contained inside the IP packet */
if (op_pk_nfd_get (pkp, "data", &encap_tcp_pktp) == OPC_COMPCODE_FAILURE)
{
     printf  ("Unable to extract TCP packet from ip");
return SN_FUNC_FAILED;

}
else
{
    /* Extract Tcp Info */
op_pk_nfd_access (encap_tcp_pktp, "fields", &tcp_fdp);

tcp_infop->src_port = tcp_fdp->src_port;
tcp_infop->dest_port = tcp_fdp->dest_port;
tcp_infop->seq_num = tcp_fdp->seq_num;
tcp_infop->ack_num = tcp_fdp->ack_num;
tcp_infop->rcv_win = tcp_fdp->rcv_win;
tcp_infop->urgent_pointer = tcp_fdp->urgent_pointer;
tcp_infop->data_len = tcp_fdp->data_len;
tcp_infop->urg = tcp_fdp->urg;
tcp_infop->ack = tcp_fdp->ack;
tcp_infop->push = tcp_fdp->push;
tcp_infop->rst = tcp_fdp->rst;
tcp_infop->syn = tcp_fdp->syn;
```

```
tcp_infop->fin = tcp_fdp->fin;

   op_pk_nfd_set (pkp, "data", encap_tcp_pktp);

return SN_FUNC_SUCCESS;
}
}

/* snCachedPkt */
/* Copy a packet to an empty spot in cache */
int snCachedPkt (Packet *pkp, int ici_addr)
{
   int i=0;

   /* Find an empty space */
   for (i=0; i < SNOOP_CACHE_SIZE; i++)
   {
      if (snoop_cache.cached_pkp[i] == OPC_NIL)
  {
     /* Found an empty spot */
     break;
 }
    }

   if (i == SNOOP_CACHE_SIZE)
{
   printf("snCachePkt: Cache full !!");
   /* Cache is full */
   return SN_FUNC_CACHE_FULL;
}

/* Copy packet to cache */

snoop_cache.cached_pkp[i] = op_pk_copy (pkp);
snoop_cache.ici_addr[i] = ici_addr;

printf ("Cached packet at %d\n",i);

/* Update statistics */
snoop_cache.num_cached++;
snoop_cache.curr_cached++;

/* Update statistics handle */
op_stat_write(curr_cached_stathandle, snoop_cache.curr_cached);
return SN_FUNC_SUCCESS;
}


/* snCacheInit */
/* Initialize the cache system */
void snCacheInit (void)
{
   int i;

   for (i=0; i < SNOOP_CACHE_SIZE; i++)
   {
     snoop_cache.cached_pkp[i] = (Packet *) OPC_NIL;
 snoop_cache.ici_addr[i] = 0;
   }
```

```
   snoop_cache.max_cached = 0;
   snoop_cache.num_cached = 0;
   snoop_cache.num_removed = 0;
   snoop_cache.curr_cached = 0;
 }

/* snCacheRetrieve */
/* Retrieve the first packet that follows the supplied sequence */
/* Note: We make a copy before sending it out */
/* We will destroy the packet when snCacheDestroy is called */
int snCacheRetrieve (unsigned int src_ip,
 unsigned int dest_ip,
 int src_port,
 int dest_port,
                int seq_num,
 Packet **pkpp,
 int *ici_addr)
{
   int i;
   TcpInfo tcp_info;

   printf ("Enter Retrieve %d\n", seq_num);

   /* Find the packet that has the sequence number closest to the start number */
   for (i=0; i < SNOOP_CACHE_SIZE; i++)
   {
    if (snoop_cache.cached_pkp[i] != OPC_NIL)
 {
 /* Extract info from cached packet */
  snGetTcpInfo (snoop_cache.cached_pkp[i], &tcp_info);

if ( (tcp_info.src_ip == src_ip) &&
 (tcp_info.src_port == src_port) &&
 (tcp_info.dest_ip == dest_ip) &&
 (tcp_info.dest_port == dest_port) &&
          (tcp_info.seq_num == seq_num)
   )
{
/* Packet found */
    /* Copy packet */
    /* We need to copy the packet because we don't want to lose */
    /* the ownership of it . Ownership is gone when the packet is sent */

    *pkpp = op_pk_copy (snoop_cache.cached_pkp[i]);
*ici_addr = snoop_cache.ici_addr[i];

printf ("Retrieve packet at %d\n", i);

return SN_FUNC_SUCCESS;
}
 }
 }
   printf ("Exit Retrieve %d\n", seq_num);

   return SN_FUNC_FAILED;


}
```

```
/* snCacheDestroy */
/* Destroy packet with sequence number smaller than specified */
int snCacheDestroy (unsigned int src_ip,
                unsigned int dest_ip,
int src_port,
int dest_port,
int seq_num)
{
int i;
TcpInfo tcp_info;


/* Search for all packets with smaller sequence number */

for (i=0; i < SNOOP_CACHE_SIZE; i++)
{
    if (snoop_cache.cached_pkp[i] != OPC_NIL)
{
snGetTcpInfo (snoop_cache.cached_pkp[i], &tcp_info);

/* The last check using data legnth is */
/* for the case when datalegnth is 0 */
if ( (tcp_info.src_ip == src_ip) &&
 (tcp_info.src_port == src_port) &&
 (tcp_info.dest_ip == dest_ip) &&
 (tcp_info.dest_port == dest_port) &&
 (tcp_info.seq_num + tcp_info.data_len <= seq_num) )
{
/* Packet found */

op_pk_destroy (snoop_cache.cached_pkp[i]);

printf ("Destroy packet at %d\n", i);
snoop_cache.cached_pkp[i] = OPC_NIL;

/* Update statistics */
snoop_cache.num_removed++;
snoop_cache.curr_cached--;

op_stat_write(curr_cached_stathandle, snoop_cache.curr_cached);

}
}


  }

return SN_FUNC_SUCCESS;
 }


/* snStripTcpAck */
/* Strip off the ack portion of TCP */
int snStripTcpAck (Packet *pkp)
{
Packet *encap_tcp_pktp;
TcpT_Seg_Fields *tcp_fdp;

 /* Get TCP Packet contained inside the IP packet */
if (op_pk_nfd_get (pkp, "data", &encap_tcp_pktp) == OPC_COMPCODE_FAILURE)
```

```
{
     printf  ("Unable to extract TCP packet from ip");
return SN_FUNC_FAILED;
}
else
{
/* Extract Tcp Info */
op_pk_nfd_get (encap_tcp_pktp, "fields", &tcp_fdp);

/* Remove the ack */
tcp_fdp->ack = 0;

/* Put it back to the TCP packet */
op_pk_nfd_set (encap_tcp_pktp, "fields", tcp_fdp, tcp_seg_fdstruct_copy,
tcp_seg_fdstruct_destroy, sizeof (TcpT_Seg_Fields) );

         op_pk_nfd_set (pkp, "data", encap_tcp_pktp);

return SN_FUNC_SUCCESS;
}
}

/* This function returns 1 if the input packet cannot be found in the cache*/
/* Otherwise, returns 0 */
int NewInCache(Packet *pkp)
{
   int i;
   TcpInfo tcp_info;
   TcpInfo tcp_info2;

   snGetTcpInfo (pkp, &tcp_info);

   /* Find the packet that has the sequence number closest to the start number */
   for (i=0; i < SNOOP_CACHE_SIZE; i++)
   {
    if (snoop_cache.cached_pkp[i] != OPC_NIL)
 {
 /* Extract info from cached packet */
 snGetTcpInfo (snoop_cache.cached_pkp[i], &tcp_info2);

if ( (tcp_info.src_ip == tcp_info2.src_ip) &&
 (tcp_info.src_port == tcp_info2.src_port) &&
 (tcp_info.dest_ip == tcp_info2.dest_ip) &&
 (tcp_info.dest_port == tcp_info2.dest_port) &&
          (tcp_info.seq_num == tcp_info2.seq_num) &&
 (tcp_info.data_len == tcp_info2.data_len) &&
      (tcp_info.ack == tcp_info2.ack) &&
      (tcp_info.ack == tcp_info2.fin)


)
{

printf ("Pkt found in cache\n");
return 0;
}
 }
}

   return 1;
```

```
}

/* This function initializes all the values in each record of the Snoop Table */
void TableInit()
{
  int i;
  for (i=0; i< SNOOP_TABLE_SIZE; i++)
  {
   snoop_table[i].src_ip =0;
   snoop_table[i].dest_ip =0;
   snoop_table[i].src_port =0;
   snoop_table[i].dest_port =0;
   snoop_table[i].last_seq_num =0;
   snoop_table[i].last_ack_num =0;
   snoop_table[i].repeat_ack =0;
   snoop_table[i].fin_flag = 0;

  }
}

/* The function returns the index to the table that matched the input parameters*/
/* Returns 1 if the record can be found; else returns 0 */
int FindTable(unsigned int src_ip,
          unsigned int dest_ip,
  int src_port,
  int dest_port,
  int *TabIdx)

{
      int i;
     for (i=0; i< SNOOP_TABLE_SIZE; i++)
  {


   if ( (snoop_table[i].src_ip == src_ip) &&
        (snoop_table[i].dest_ip == dest_ip) &&
        (snoop_table[i].src_port == src_port) &&
        (snoop_table[i].dest_port == dest_port)
      )
 {


  *TabIdx = i;
      return 1;
 }
  }

 return 0;
}

/* This function adds the tcpinfo into the snoop table*/
/* Returns 1 if the add is successful, else returns 0 (table full */
/* also updates table counter */
int AddTable(TcpInfo info, int *TabIdx)

    {
      int i;
  if (table_free == 0)
  {
```

```
    printf ("snoop table full !!");
    return 0;
  }

     for (i=0; i< SNOOP_TABLE_SIZE; i++)
  {
   if ( (snoop_table[i].src_ip == 0) &&
 (snoop_table[i].dest_ip == 0) &&
 (snoop_table[i].src_port == 0) &&
 (snoop_table[i].dest_port == 0)
)
 {
  snoop_table[i].src_ip = info.src_ip;
              snoop_table[i].dest_ip = info.dest_ip;
  snoop_table[i].src_port = info.src_port;
  snoop_table[i].dest_port = info.dest_port;
  snoop_table[i].last_seq_num = info.seq_num;
  snoop_table[i].last_ack_num = 0;
  snoop_table[i].repeat_ack = 0;

  table_free--;

  *TabIdx = i;
  return 1;
 }
   }
  return 0;
}

/* This function deletes a record in the snoop table */
/* Also updates the table counter */
void snTableDestroy (int tab_idx)
{
printf ("Destroying connection %d\n", tab_idx);

snoop_table[tab_idx].src_ip =0;
snoop_table[tab_idx].dest_ip =0;
snoop_table[tab_idx].src_port =0;
snoop_table[tab_idx].dest_port =0;
snoop_table[tab_idx].last_seq_num =0;
snoop_table[tab_idx].last_ack_num =0;
snoop_table[tab_idx].repeat_ack =0;
snoop_table[tab_idx].fin_flag = 0;

/* Unschedule timeout event */
if (op_ev_valid (snoop_table[tab_idx].timeout_evt) )
{
/* Cancel the previously scheduled event */
op_ev_cancel (snoop_table[tab_idx].timeout_evt);
}

table_free++;
}


/* Extend the timeout period of a connection */
void snExtendTO (int tab_idx)
{
if (op_ev_valid (snoop_table[tab_idx].timeout_evt) )
{
```

```
/* Cancel the previously scheduled event */
op_ev_cancel (snoop_table[tab_idx].timeout_evt);
}
/* Schedule the timeout event */
snoop_table[tab_idx].timeout_evt = op_intrpt_schedule_self (op_sim_time() +
timeout_interval,
  tab_idx);
}


/* Handle timeout event */
/* Basically re-tx packets since last received ack */
void snHandleTimeout (int tab_idx)
{
Packet *cached_pkptr;

int rv = SN_FUNC_SUCCESS;

printf("Retx Connection %d\n", tab_idx);

   if (tab_idx < 0 || tab_idx > SNOOP_TABLE_SIZE)
{
printf ("Invalid table index %d\n", tab_idx);
}

tcpinfo.dest_ip = snoop_table[tab_idx].dest_ip;
tcpinfo.src_ip = snoop_table[tab_idx].src_ip;
tcpinfo.dest_port = snoop_table[tab_idx].dest_port;
tcpinfo.src_port = snoop_table[tab_idx].src_port;
tcpinfo.seq_num = snoop_table[tab_idx].last_ack_num;

/* retrieve pkts and send */
/* note: src is actually dest and dest is src */
while  ((snCacheRetrieve (tcpinfo.src_ip,
tcpinfo.dest_ip,
tcpinfo.src_port,
tcpinfo.dest_port,
tcpinfo.seq_num,
&cached_pkptr,
&ici_next_addr) == SN_FUNC_SUCCESS )
        &&
(rv == SN_FUNC_SUCCESS))

{
iciptr = op_ici_create ("ip_arp_req_v4");
op_ici_attr_set (iciptr, "next_addr", ici_next_addr);
op_ici_install (iciptr);


/* need to get size to get next seq_num*/
rv = snGetTcpInfo (cached_pkptr, &tcpinfo2);
tcpinfo.seq_num += tcpinfo2.data_len;

printf ("Retx Data: Ip: src %d, port %d, dest %d, port %d ici_addr %d \n",
tcpinfo2.src_ip, tcpinfo2.src_port, tcpinfo2.dest_ip, tcpinfo2.dest_port, ici_next_addr);

printf ("Retx Data: Tcp: data_len %d, seq %d, ack_seq %d, syn %d, fin %d\n",
tcpinfo2.data_len, tcpinfo2.seq_num, tcpinfo2.ack_num, tcpinfo2.syn, tcpinfo2.fin);

op_pk_send (cached_pkptr, 1);
```

```
op_ici_install (OPC_NIL);

   }

/* Extend timeout */

/* We can't use snExtendTO because we can't cancel current event */
snoop_table[tab_idx].timeout_evt = op_intrpt_schedule_self (op_sim_time() +
timeout_interval,
  tab_idx);
}

/* End of Function Block */

#if defined (__cplusplus)
extern "C" {
#endif
void snoop_recv (void);
Compcode snoop_recv_init (void **);
void snoop_recv_diag (void);
void snoop_recv_terminate (void);
void snoop_recv_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif



/* Process model interrupt handling procedure */


void
snoop_recv (void)
{
int _block_origin = 0;
FIN (snoop_recv ());
if (1)
{
Packet*pkptr;
Packet*     cached_pkptr;
int intrpt_type = OPC_INT_UNDEF;
int intrpt_strm = OPC_INT_UNDEF;
char format_name[256];
int tmp_val;  /* General purpose temp variable */



FSM_ENTER (snoop_recv)

FSM_BLOCK_SWITCH
{
/*---------------------------------------------------------*/
/** state (Wait) enter executives **/
FSM_STATE_ENTER_UNFORCED (0, state0_enter_exec, "Wait", "snoop_recv () [Wait enter
execs]")
{
}
```

```
/** blocking after enter executives of unforced state. **/
FSM_EXIT (1,snoop_recv)


/** state (Wait) exit executives **/
FSM_STATE_EXIT_UNFORCED (0, state0_exit_exec, "Wait", "snoop_recv () [Wait exit execs]")
{
/* Obtain interrupt parameters.*/
intrpt_type = op_intrpt_type ();
if (intrpt_type == OPC_INTRPT_STRM)
{
intrpt_strm = op_intrpt_strm ();
}

}


/** state (Wait) transition processing **/
FSM_INIT_COND (HOST_PKT_ARRIVAL)
FSM_TEST_COND (MOB_PKT_ARRIVAL)
FSM_TEST_COND (RETX_TIMEOUT)
FSM_TEST_LOGIC ("Wait")

FSM_TRANSIT_SWITCH
{
FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;)
FSM_CASE_TRANSIT (1, 3, state3_enter_exec, ;)
FSM_CASE_TRANSIT (2, 4, state4_enter_exec, ;)
}
/*---------------------------------------------------------*/



/** state (INIT) enter executives **/
FSM_STATE_ENTER_FORCED (1, state1_enter_exec, "INIT", "snoop_recv () [INIT enter execs]")
{
host_pkcnt_stathandle = op_stat_reg ("host_pkt_cnt", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
mob_pkcnt_stathandle = op_stat_reg ("mob_pkt_cnt", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
curr_cached_stathandle = op_stat_reg ("curr_cached", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);

host_pkcnt = 0;
mob_pkcnt = 0;

snCacheInit();
TableInit();

op_ima_obj_attr_get (op_id_self(), "snoop_enable", &snoop_enable);
op_ima_obj_attr_get (op_id_self(), "timeout_int", &timeout_interval);

/* Initialize all statistics */
op_stat_write(host_pkcnt_stathandle, host_pkcnt);
op_stat_write(mob_pkcnt_stathandle, mob_pkcnt);
op_stat_write(curr_cached_stathandle, snoop_cache.curr_cached);
}


/** state (INIT) exit executives **/
FSM_STATE_EXIT_FORCED (1, state1_exit_exec, "INIT", "snoop_recv () [INIT exit execs]")
```

```
{
}


/** state (INIT) transition processing **/
FSM_TRANSIT_FORCE (0, state0_enter_exec, ;)
/*---------------------------------------------------------*/



/** state (Snoop_Data) enter executives **/
FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "Snoop_Data", "snoop_recv () [Snoop_Data
enter execs]")
{
if (intrpt_type == OPC_INTRPT_STRM)
{
/* Get packet */
pkptr = op_pk_get (op_intrpt_strm ());

/* Get ICI */
snoop_host_iciptr = op_intrpt_ici();

/* if ICI not null, install ICI */
if (snoop_host_iciptr != OPC_NIL)
{
      op_ici_install (snoop_host_iciptr);

  if (op_ici_attr_get(snoop_host_iciptr, "next_addr", &ici_next_addr) ==
OPC_COMPCODE_FAILURE )
  printf("ici_get snoop_host failure\n");

}
/* update statistics */
host_pkcnt++;
op_stat_write(host_pkcnt_stathandle, host_pkcnt);

/* if snoop is turn off */
if (!snoop_enable)
{
    if (snGetTcpInfo (pkptr, &tcpinfo) != SN_FUNC_FAILED)
{
        /* Debug Msg */
    /*printf ("Data (Dis): Ip: src %d, port %d, dest %d, port %d ici_addr %d \n",
tcpinfo.src_ip, tcpinfo.src_port, tcpinfo.dest_ip, tcpinfo.dest_port, ici_next_addr);

printf ("Data (Dis): Tcp: data_len %d, seq %d, ack_seq %d\n",
tcpinfo.data_len, tcpinfo.seq_num, tcpinfo.ack_num); */
}

    /* Snoop protocol disabled. Just forward packet to destination */
op_pk_send (pkptr, 1);

op_ici_install (OPC_NIL);
}

/* if snoop is turned on */
else
{
/* Get the info in the tcp header and the next_addr in the ICI */
if (snGetTcpInfo (pkptr, &tcpinfo) == SN_FUNC_FAILED ||
```

```
(op_ici_attr_get(snoop_host_iciptr, "next_addr", &ici_next_addr)
  == OPC_COMPCODE_FAILURE ) )
{
/* if get tcip info or ICI fails */
       printf ("Snoop_Data: Unable to get data from IP datagram OR get ICI next addr
fail");
   op_pk_send (pkptr, 1);
      op_ici_install (OPC_NIL);

}
else /* gettcpinfo success */
{
/* Debug Msg */
printf ("Data: Ip: src %d, port %d, dest %d, port %d ici_addr %d \n",
tcpinfo.src_ip, tcpinfo.src_port, tcpinfo.dest_ip, tcpinfo.dest_port, ici_next_addr);

printf ("Data: Tcp: data_len %d, seq %d, ack_seq %d, syn %d, fin %d\n",
tcpinfo.data_len, tcpinfo.seq_num, tcpinfo.ack_num, tcpinfo.syn, tcpinfo.fin);



  /* Check to see if data length is equal to zero */
  /* If zero, this packet is just an acknowledgement for the opposite side */
  if (tcpinfo.data_len == 0)
  {
     /* Just pass forward */
      /* Snoop protocol disabled. Just forward packet to destination */
      op_pk_send (pkptr, 1);

  op_ici_install (OPC_NIL);


  }
  /* check if the cache has already cached this packet */
  else if (NewInCache(pkptr) )
  /* if new packet*/
  {
  /* Check if this is a new tcp connection */
   if ( !FindTable(tcpinfo.src_ip,
               tcpinfo.dest_ip,
   tcpinfo.src_port,
   tcpinfo.dest_port,
   &tab_idx) )
   /* new tcp connection */
   {
    /* if cannot add to table just forward packet*/
     if (AddTable(tcpinfo, &tab_idx) == 0)
   {

   op_pk_send (pkptr, 1);

op_ici_install (OPC_NIL);
   }
 /* add table successful */
 else
 {/* if FIN, indicates the end of the tcp connection */
       if (tcpinfo.fin == 1)
   {
/* deletes the connection record in snoop table */
          snTableDestroy (tab_idx);
```

```
   printf ("Mark connection as finished, %d\n", tcpinfo.seq_num);

}
else
{
/* no matter cache success or not , sedn pkt */
snCachedPkt (pkptr, ici_next_addr);

/* Extend timeout period */
snExtendTO (tab_idx);

 /* First packet for connection, mark down seq number as last ack */
snoop_table[tab_idx].last_ack_num = tcpinfo.seq_num;
}

   op_pk_send (pkptr, 1);
   op_ici_install (OPC_NIL);




 }
   } /* end if FindTable */


   else /* if found in table (connection has been recorded)*/
   {
    /* new pkt */
     if (tcpinfo.seq_num > snoop_table[tab_idx].last_seq_num)
 {

 /* if FIN, indicates the end of the tcp connection */
    if (tcpinfo.fin == 1)
   {
   /* Close connection here. Don't wait for the ack,since the host can deal with it */

   snTableDestroy (tab_idx);

printf ("Mark connection as finished, %d\n", tcpinfo.seq_num);

}
   else
   {
   /* no matter cache success or not , sedn pkt */
   snCachedPkt (pkptr, ici_next_addr);
snoop_table[tab_idx].last_seq_num = tcpinfo.seq_num;
   }


   op_pk_send (pkptr, 1);
   op_ici_install (OPC_NIL);


 }

     else /* sender retransmission */
 {

    /* no matter cache success or not , send pkt */
```

```
   /* snCachedPkt (pkptr, ici_next_addr); */
 /* We don't cache sender retx packet. We might not */
 /* delete them if cached */
   op_pk_send (pkptr, 1);
   op_ici_install (OPC_NIL);
 }



  /* Extend timeout period */
  snExtendTO (tab_idx);


   } /* end else found in table */



} /* end if NewInCache */

  else /* already cached */
  {

if ( !FindTable(tcpinfo.src_ip,
               tcpinfo.dest_ip,
   tcpinfo.src_port,
   tcpinfo.dest_port,
   &tab_idx) )
   {

     /* If enter here, there is something wrong */
    /* if cannot add to table just forward packet*/
     if (AddTable(tcpinfo, &tab_idx) == 0)
   {
   op_pk_send (pkptr, 1);
op_ici_install (OPC_NIL);
   }
 /* add table successful */
 else
 {
    /* did not reach target, resend */
   if (tcpinfo.seq_num > snoop_table[tab_idx].last_ack_num)
    {
               op_pk_send (pkptr, 1);
op_ici_install (OPC_NIL);


/* Extend timeout period */
snExtendTO (tab_idx);

   /* First packet for connection, mark down seq number as last ack */
   snoop_table[tab_idx].last_ack_num = tcpinfo.seq_num;


    }
 } /* else reached target already; discard pkt */
   }
else /* If cached, and connection found in snoop table */
{
/* Extend timeout period */
snExtendTO (tab_idx);
```

```
}


   } /* end else already cached */


} /* end else gettcpinfo success */

 } /* end else SNOOP_ON */

}


}


/** state (Snoop_Data) exit executives **/
FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "Snoop_Data", "snoop_recv () [Snoop_Data exit
execs]")
{
}


/** state (Snoop_Data) transition processing **/
FSM_TRANSIT_FORCE (0, state0_enter_exec, ;)
/*---------------------------------------------------------*/



/** state (Snoop_Ack) enter executives **/
FSM_STATE_ENTER_FORCED (3, state3_enter_exec, "Snoop_Ack", "snoop_recv () [Snoop_Ack
enter execs]")
{
if (intrpt_type == OPC_INTRPT_STRM)
{
/* Get packet */
pkptr = op_pk_get (op_intrpt_strm ());

/* Get ICI */
snoop_mob_iciptr = op_intrpt_ici();

/* Update Statistics */
mob_pkcnt++;
op_stat_write(mob_pkcnt_stathandle, mob_pkcnt);

/* if snoop is turned off */
if (!snoop_enable)
{
if (snoop_mob_iciptr != OPC_NIL)
{
      op_ici_install (snoop_mob_iciptr);

}
 /* just Forward packet if snoop not on*/
     op_pk_send (pkptr, 0);
      op_ici_install (OPC_NIL);

}
```

```
/* if snoop is turn on */
else

{

/* Try to get some info about the ICI */
if (snoop_mob_iciptr != OPC_NIL)
{
op_ici_print (snoop_mob_iciptr);

}
/* get tcp header info and get next address in the ICI*/
/* if fails */
if (snGetTcpInfo (pkptr, &tcpinfo) == SN_FUNC_FAILED)
{
        printf ("Unable to get data from IP datagram ");

   if (snoop_mob_iciptr != OPC_NIL)
 {
   op_ici_install (snoop_mob_iciptr);
             }

   op_pk_send (pkptr, 0);
      op_ici_install (OPC_NIL);
}

/* get tcp ici info suceeds */
else
{

   /* Debug Msg */
printf ("Ack: Ip: src %d, port %d, dest %d, port %d\n",
tcpinfo.src_ip, tcpinfo.src_port, tcpinfo.dest_ip, tcpinfo.dest_port);

printf ("Ack: Tcp: data_len %d, seq %d, ack_seq %d, syn %d, fin %d\n",
tcpinfo.data_len, tcpinfo.seq_num, tcpinfo.ack_num, tcpinfo.syn, tcpinfo.fin);

/* if not an ack or if fin, just forward packet */
 if (tcpinfo.ack != 1 || tcpinfo.fin == 1)
 {
 if (tcpinfo.ack != 1)
{
printf ("Not ack\n");
}
if (tcpinfo.fin == 1)
{
printf ("Fin = 1\n");
}

   /* This packet is not an acknowledgement */
   /* Or it is a FIN packet, then we will not process it */
         if (snoop_mob_iciptr != OPC_NIL)
{
op_ici_install (snoop_mob_iciptr);
}

op_pk_send (pkptr, 0);
op_ici_install (OPC_NIL);
 }
```

```
 /* check if this tcp connection is already recored in the snoop table */
 else if ( !FindTable(tcpinfo.dest_ip,
  tcpinfo.src_ip,
 tcpinfo.dest_port,
 tcpinfo.src_port,
 &tab_idx) )
  /* if new connection */
  {
  printf("Received packet with unknown connection\n");
  if (snoop_mob_iciptr != OPC_NIL)
  {
  op_ici_install (snoop_mob_iciptr);
                    }
   /* if not in table, not in cache */
    /* just forward packet */
op_pk_send (pkptr, 0);
op_ici_install (OPC_NIL);
  }

 /* new ack */
 else if (tcpinfo.ack_num > snoop_table[tab_idx].last_ack_num)
 {
   printf("Received new ack \n");
   snCacheDestroy (tcpinfo.dest_ip,
      tcpinfo.src_ip,
                           tcpinfo.dest_port,
   tcpinfo.src_port,
                           tcpinfo.ack_num);

   /* update last ack num in snoop table */
   snoop_table[tab_idx].last_ack_num = tcpinfo.ack_num;
   snoop_table[tab_idx].repeat_ack = 0;

   if (snoop_mob_iciptr != OPC_NIL)
  {
  op_ici_install (snoop_mob_iciptr);
                      }

   /* send pkt */
   op_pk_send (pkptr, 0);
              op_ici_install (OPC_NIL);

   /* Extend timeout */
   snExtendTO (tab_idx);


 }
 /* duplicate ack from sender; first duplicate ack */
 else if (snoop_table[tab_idx].repeat_ack == 0)
 {
    printf("Duplicate ack, not repeat\n");
/* set repeat ack so won't send again for same dupack */
    snoop_table[tab_idx].repeat_ack++;
    rv = SN_FUNC_SUCCESS;


tmp_val = 0;
   /* retrieve packet since last ack and send pkts*/
    /* note: src is actually dest and dest is src */
 while  ((snCacheRetrieve (tcpinfo.dest_ip,
```

```
  tcpinfo.src_ip,
  tcpinfo.dest_port,
  tcpinfo.src_port,
  tcpinfo.ack_num,
  &cached_pkptr,
  &ici_next_addr) == SN_FUNC_SUCCESS )
        &&
     (rv == SN_FUNC_SUCCESS))

{
 /* create, set and install ICI */
 iciptr = op_ici_create ("ip_arp_req_v4");
 op_ici_attr_set (iciptr, "next_addr", ici_next_addr);
 op_ici_install (iciptr);

 tmp_val++;
 /* need to get size to get next seq_num*/
 rv = snGetTcpInfo (cached_pkptr, &tcpinfo2);
     tcpinfo.ack_num += tcpinfo2.data_len;

 op_pk_send (cached_pkptr, 1);
                    op_ici_install (OPC_NIL);



 } /* end while */

/* Since we have decided to retransmit the packet ourselves, the host */
/* should not see the ack packet, otherwise it will half its window size */

/* However, if we no more data to retransmit, then pass this ack back to the host */
/* Let the host deals with it.. The host might have lost the last ack that we passed */
/* up */
if (tmp_val == 0)
{
 if (snoop_mob_iciptr != OPC_NIL)
 {
 op_ici_install (snoop_mob_iciptr);
 }

op_pk_send (pkptr, 0);
op_ici_install (OPC_NIL);
}


else if (tcpinfo.data_len != 0)
{
   /* There are data for the opposite path, so we strip off the ack */
snStripTcpAck (pkptr);

 if (snoop_mob_iciptr != OPC_NIL)
 {
 op_ici_install (snoop_mob_iciptr);
 }

op_pk_send (pkptr, 0);
op_ici_install (OPC_NIL);
}
else
{
```

```
   /* No data, destroy the packet */
   op_pk_destroy (pkptr);
}

 /* Extend timeout */
 snExtendTO (tab_idx);

  }
 /* duplicate repeat ack */
 else if (snoop_table[tab_idx].repeat_ack > 0)
    {
printf("Duplicate ack,repeat pack\n");
snoop_table[tab_idx].repeat_ack++;
/* should do something if receive a lot */

if (tcpinfo.data_len != 0)
{
/* There are data for the opposite path, so we strip off the ack */
//snStripTcpAck (pkptr);

if (snoop_mob_iciptr != OPC_NIL)
 {
 op_ici_install (snoop_mob_iciptr);
 }

op_pk_send (pkptr, 0);
op_ici_install (OPC_NIL);
}
else
{
/* No data, destroy the packet */
op_pk_destroy (pkptr);
}

 /* Extend timeout */
 snExtendTO (tab_idx);

}




}

} /* end else SNOOP_ON */

}

}


/** state (Snoop_Ack) exit executives **/
FSM_STATE_EXIT_FORCED (3, state3_exit_exec, "Snoop_Ack", "snoop_recv () [Snoop_Ack exit
execs]")
{
}


/** state (Snoop_Ack) transition processing **/
FSM_TRANSIT_FORCE (0, state0_enter_exec, ;)
```

```
/*---------------------------------------------------------*/



/** state (Timeout) enter executives **/
FSM_STATE_ENTER_FORCED (4, state4_enter_exec, "Timeout", "snoop_recv () [Timeout enter
execs]")
{

/* The code that comes with the interrupt */
/* is the table index */
snHandleTimeout (op_intrpt_code());



}


/** state (Timeout) exit executives **/
FSM_STATE_EXIT_FORCED (4, state4_exit_exec, "Timeout", "snoop_recv () [Timeout exit
execs]")
{
}


/** state (Timeout) transition processing **/
FSM_TRANSIT_FORCE (0, state0_enter_exec, ;)
/*---------------------------------------------------------*/



}


FSM_EXIT (1,snoop_recv)
}
}

#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
snoop_recv_init (void ** gen_state_pptr)
{
int _block_origin = 0;
static VosT_Addressobtype = OPC_NIL;

FIN (snoop_recv_init (gen_state_pptr))

if (obtype == OPC_NIL)
{
```

```
/* Initialize memory management */
if (Vos_Catmem_Register ("proc state vars (snoop_recv)",
sizeof (snoop_recv_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 */
((snoop_recv_state *)(*gen_state_pptr))->current_block = 2;

FRET (OPC_COMPCODE_SUCCESS)
}
}



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




void
snoop_recv_terminate (void)
{
int _block_origin = __LINE__;

FIN (snoop_recv_terminate (void))

if (1)
{
Packet*pkptr;
Packet*     cached_pkptr;
int intrpt_type = OPC_INT_UNDEF;
int intrpt_strm = OPC_INT_UNDEF;
char format_name[256];
int tmp_val;  /* General purpose temp variable */


/* 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 snoop_recv_svar function. */
#undef snoop_host_iciptr
#undef host_pkcnt
#undef mob_pkcnt
#undef host_pkcnt_stathandle
#undef mob_pkcnt_stathandle
#undef snoop_mob_iciptr
#undef ici_next_addr
#undef snoop_enable
#undef curr_cached_stathandle
#undef timeout_interval



void
snoop_recv_svar (void * gen_ptr, const char * var_name, char ** var_p_ptr)
{
snoop_recv_state*prs_ptr;

FIN (snoop_recv_svar (gen_ptr, var_name, var_p_ptr))

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

if (strcmp ("snoop_host_iciptr" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->snoop_host_iciptr);
FOUT;
}
if (strcmp ("host_pkcnt" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->host_pkcnt);
FOUT;
}
if (strcmp ("mob_pkcnt" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->mob_pkcnt);
FOUT;
}
if (strcmp ("host_pkcnt_stathandle" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->host_pkcnt_stathandle);
FOUT;
}
if (strcmp ("mob_pkcnt_stathandle" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->mob_pkcnt_stathandle);
FOUT;
}
if (strcmp ("snoop_mob_iciptr" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->snoop_mob_iciptr);
FOUT;
}
if (strcmp ("ici_next_addr" , var_name) == 0)
```

```
{
*var_p_ptr = (char *) (&prs_ptr->ici_next_addr);
FOUT;
}
if (strcmp ("snoop_enable" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->snoop_enable);
FOUT;
}
if (strcmp ("curr_cached_stathandle" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->curr_cached_stathandle);
FOUT;
}
if (strcmp ("timeout_interval" , var_name) == 0)
{
*var_p_ptr = (char *) (&prs_ptr->timeout_interval);
FOUT;
}
*var_p_ptr = (char *)OPC_NIL;

FOUT;
}
```