b20921329

TK
7872
.D48
N67
2011

ENHANCING BROADCAST AUTHENTICATION IN SENSOR NETWORKS

Arayeh Norouzi
Honours BSc, Computer Science, York University, Toronto, Canada, 2002

A project
presented to Ryerson University

in partial fulfillment of the
requirements for the degree of

MASTER OF ENGINEERING

in the Program of

ELECTRICAL AND COMPUTER ENGINEERING

from the

RYERSON UNIVERSITY
Toronto, Ontario, Canada, 2011

©(Arayeh Norouzi) 2011

PROPERTY OF
RYERSON UNIVERSITY LIBRARY

THIS PAGE INTENTIONALLY LEFT BLANK

I hereby declare that I am the sole author of this project.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

Arayeh Norouzi

I further authorize Ryerson University to reproduce this project by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Arayeh Norouzi

ENHANCING BROADCAST AUTHENTICATION IN SENSOR NETWORKS

by

Arayeh Norouzi

Master of Engineering, Electrical and Computer Engineering, Ryerson University, 2011

Supervisor: Dr. Truman Yang

ABSTRACT

Due to the nature of wireless sensor networks, security is a critical problem since resource constrained and usually unattended sensors are much vulnerable to malicious attackers that may impersonate the sender. Therefore authenticating received messages is a crucial matter to protect the system integrity. Generally used TESLA (Timed Efficient Stream Loss-tolerant Authentication) based authentication techniques involve consecutive delays for decryption purposes. These delays render the network vulnerable to different malicious attacks such as Denial of Service attack. As several techniques try to achieve immediate authentication to alleviate these threats, other factors such as reliability and buffer requirements may have been compromised. This project proposes an integration of Low Buffer μTESLA protocol and an immediate authentication protocol to achieve a new refined scheme in broadcast authentication in sensor networks. Performance analysis and simulation results demonstrate that the proposed method succeeds to achieve immediate authentication while preserving desired security and low memory requirements in sensor nodes.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like first to express my appreciation to Dr. Truman Yang, my project advisor, and my co-advisor, Dr. Abdolreza Abhari for their guidance through the completion of this project.

I would also like to thank my other professors whose valuable courses I have taken throughout the years at Ryerson University.

Lastly, and most importantly, I wish to thank Farzin, Mauna, Nima and my parents for their patience and their presence. I would love to thank Farzin for his greatest and unconditional support in all aspects of my life, especially during the course of my studies. He is the best thing that has happened to me in life. This project is dedicated to all my family.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER I: INTRODUCTION

Wireless sensor networks (WSNs) are a novel paradigm for large distributed systems. They consist of many small sensing devices which are spread over large geographic areas and are used to collect and process different data. Sensor networks offer economic solutions to challenging problems such as traffic monitoring, environment data gathering, medical monitoring, industrial automation and homeland security. The sensor nodes in WSN are usually battery-powered and resource-restricted and often operate with more powerful base stations that serve as network bridges and provide computational resources and collect data [6]. Securing wireless sensor networks is an essential and on-going issue due to their critical applications. Nature of wireless communication in sensor networks makes it easily vulnerable to attackers that aim to inject malicious data or change legitimate messages [4, 10]. Therefore these applications need to employ efficient and reliable authentication mechanisms to ensure received data originated from a valid source and was not altered maliciously along the way [3].

Recently proposed TESLA (Timed Efficient Stream Loss-tolerant Authentication) based techniques [2] have embarked on resolving the authentication problem by employing symmetric encryption and achieving the desired security level by mimicking asymmetric encryption through delayed key disclosure. The suggested delay renders the network vulnerable to Denial of Service attack since an adversary can flood the nodes by sending bogus messages and forcing the sensors to buffer the messages until they receive the corresponding delayed keys.

A. PROJECT OBJECTIVE

The objective of this project is to propose an integrated scheme to be able to perform broadcast authentication in sensor networks without delay while improving receiver buffer requirements. The immediate authentication will increase system resistance against malicious attacks such as Denial of Service attack. Furthermore, reducing receiver buffering offers an advantage to the system due to resource-restrained nature of sensor nodes. In this project two various versions of μTesla protocol are incorporated as a new authentication scheme. Analysis and the result of project simulation that has been implemented in a network modeler are included to demonstrate the enhanced buffer size and system delay of the proposed scheme.

B. RELATED WORK

Many security protocols have been proposed to ensure data integrity, confidentiality and authenticity throughout sensor networks. While some protocols like SNEP (Secure Network Encryption Protocol) mentioned in [3] that propose to provide two-party authentication along with data confidentiality and data freshness, others intend to provide broadcast authentication.

μTESLA-based techniques and digital signatures are two general approaches to deliver broadcast authentication. However, both these methods are vulnerable to DoS attacks. In signature-based systems an attacker can forge a large number of broadcast messages with digital signatures and force the nodes to verify these signatures and eventually deplete their resources. Similarly due to the delay existing in μTESLA-based systems, a receiver cannot authenticate a packet immediately after receiving it and the adversary can flood the system by reusing the same key to forge many packets and exhaust sensor nodes' battery power. On-going research is being done to enhance both schemes.

The approach offered in [8] intends to mitigate DoS attacks against both signature-based and μTESLA-based broadcast authentication. The scheme uses an efficiently verifiable weak authenticator along with broadcast authentication. Each sensor performs the expensive signature verification or packet forwarding only when the weak authenticator is verified. A message specific puzzle is formed for the weak authentication and a computationally powerful sender with sufficient power supply is needed for this technique. The formation and solving the puzzle requires all nodes to acquire additional parameters and computational capabilities which are rare commodities in resource-restricted low-battery sensor nodes.

Unlike general TESLA-based protocols that use loosely synchronized clocks, a protocol called TIK (TESLA with Instant Key disclosure) proposed in [16] offers to eliminate the authentication delay by using tightly synchronized clocks within the sensor nodes. Extremely precise timing allows the sender to even disclose the key in the same packet that carries the corresponding message authentication code. This enables the sender to produce receiver-specific keys. The use of tight time synchronization in this system requires a highly reliable network environment and therefore seems not feasible in unattended and usually harsh geographic conditions.

Low capacity sensor nodes require little buffering on the receiver side. Low-Buffer μTESLA protocol stated in [1] intends to swap receiver buffering with the sender buffering which is usually a more powerful base station. In this scheme, the Message Authentication Code or MAC that is used for encryption purposes and is a shorter version of the original message is sent instead of the longer message. This will allow the receiver to buffer the MAC only and wait for the message with corresponding key in a later interval to authenticate the packet. Once the message and the key are received, the corresponding MAC is popped from the queue and the message is authenticated.

Mechanism of μTESLA authentication technique involves consecutive delays for key revealing purposes. This renders the distributed sensor network vulnerable to outside attacks. A malicious attacker might impersonate the authentic sender and transmit forged packets. This makes immediate authentication a beneficial commodity to remove the risk of various attacks such as DoS attacks. The scheme cited in [2] proposes a new technique to achieve immediate authentication and remove the delay existing in original μTESLA technique. They, too, propose to replace receiver buffering with sender buffering. They include the hash of the message of a future packet in the current packet. Once the current packet is authenticated on the receiver side, the message in the future packet is also immediately authenticated.

This project intends to incorporate some of offered authentication solutions to attain a refined and highly reliable authentication mechanism in wireless sensor networks. Among all protocols, μTESLA-based technique proves to be more viable and realistic in the unfriendly surroundings of the resource-limited sensor nodes. This project embarks on creating an enhanced version of this technique to make the system resilient to DoS by providing instant authentication while considering the low-capacity requirements of the sensor nodes. To achieve this objective, this work specifically intends to integrate Low-Buffer μTESLA protocol with the immediate authentication TESLA protocol to produce a more reliable system. This incorporation preserves the desired security of the original TESLA and the sought after immediate authentication while considering low memory requirements of the resource-constrained sensors.

C. PROJECT ORGANIZATION

This project is organized as follows. Chapter II give an overview of data authentication and TESLA-based authentication mechanism in sensor networks. This chapter also discusses the shortcomings of TESLA-based authentication techniques. Chapter III addresses two different TESLA authentication protocols that propose to implement Low-Buffer μTESLA protocol and immediate authentication scheme. Furthermore, this chapter introduces the new integrated protocol that incorporates the two previously stated schemes. An overview of the network modeler OPNET is given in chapter IV. Chapter V describes the OPNET simulation of the new protocol. This chapter describes the two scenarios that have been implemented in the simulator in order to investigate the efficiency and performance of the new scheme compared with the previous protocols. Chapter VI examines the performance of the proposed protocol through analysis and simulation and illustrates result graphs obtained by two scenarios. Chapter VI provides conclusions of the project as well as recommendations for future work.

CHAPTER II: AUTHENTICATION IN SENSOR NETWORKS

Sensor network is an unsupervised distributed system that is much vulnerable to outside attacks. Receiving authentic messages from base station is vital and critical in fragile information gathering. Authenticating senders is a way to ensure system integrity and reliability.

A. NETWORK SECURITY CONCEPTS

Network systems are generally susceptible to different security issues such as interruption, interception and modification. Interruption is an attack on the availability such as cutting a communication link. Interception is an attack on confidentiality where an unauthorized party gains access to an asset such as wiretapping. Modification is an attack on the integrity of the system where an unauthorized party tampers with an asset such as changing values in a file. To alleviate some of these attacks, encryption algorithms and digital signatures are two cryptography techniques that are used to ensure data confidentiality and data integrity respectively. Encryption algorithm scrambles messages such that only the intended receiver can unscramble them. Encrypted message is generated by encryption function and the original message is extracted by decryption function, and a key is used to control these functions. The key is a sequence of random numbers that has a value from 0 to 255 bytes. The encryption algorithm determines key's length. The key ought to be truly random and long enough so that an attacker cannot exhaust all possible combinations. [37]

Encryption algorithms are used to ensure data confidentiality in a system. Symmetric and asymmetric key algorithms are two classes of encryption algorithms. In symmetric scheme, a shared key is used for both encryption and decryption, whereas a pair of keys is used in asymmetric mechanism. A public key is used in the encryption function and a private key in the decryption algorithm. Asymmetric cryptography as opposed to symmetric mechanism is more secure while it requires more computation and storage and its process takes longer. The encryption algorithm used in this project is symmetric HMAC (Hash-based Message Authentication Code) which is a hash function that is using a shared key to be produced.

However, the desired security that is lacking in symmetric scheme is achieved by disclosing the corresponding key with a delay.

Hash functions or digital signature are used in order to preserve data integrity. A hash function is a method that is applied to a piece of data to turn it to a definite-size small number serving as a digital fingerprint or the message digest. Changing one bit of data results in a complete alteration of the achieved hash value. MD5 and SHA-1 are two most-commonly used hash functions. To make the system resilient to interception and modification, every message is accompanied by its hash code. If the sender and the receiver's hash are not equal, the message was altered along the way.

A digital signature is a number attached to a message. A hash function using the public key of the receiver is used to encrypt the message to produce the signature and the receiver's private key is used to decrypt it. If the message that is obtained by decrypting the signature matches the sent message, the integrity of the data has been preserved and message is considered genuine.

B. DATA AUTHENTICATION

An authentication protocol is a process that is designed to decide if a party is, in fact, who it declares to be. Asymmetric key authentication protocols use public key encryption and they are certificate based. These protocols are ideal for secure communications with a large variable user base that are not known in advance and where it is hard to distribute a shared key. On the contrary, symmetric key authentication protocols use secret key encryption and usually rely on a trusted third party that should be available at the time. This protocol is ideal for networked environments where all service and users are known in advance. [37]

Data could be authenticated through a purely symmetric technique in a two-party communication scenario or in a point-to-point authentication. The sender and receiver share a secret key to compute a cryptographic message authentication code (MAC) over all messages [3,5]. MAC can be efficiently implemented on resource-constrained sensor network nodes. The sender uses the shared key to run the MAC algorithm on the message and sends the result code along with the original message. The receiver computes the algorithm over the received message and accepts the message if the result code is the same as the one received from the sender.

This efficient method is not secure enough when a sender wants to broadcast an authentic message to non-trusting receivers since any of the receivers know the shared MAC key and can impersonate the sender and forge messages to other receivers. Moreover, the traditional asymmetric techniques that use digital signatures for authentication have high computation, communication and storage overhead rendering them impractical for the resource-constrained sensor nodes [8]. The efficiency of the symmetric technique that is preserved by constructing authenticated broadcast from purely symmetric primitives could be integrated with the security of the asymmetric technique that could be mimicked by introducing asymmetry through delayed key disclosure and one-way function key chains. The integrated authentication method or the TESLA-based technique aims for the simultaneous efficiency and security of the sensor networks in a resource constrained environment.

C. TESLA-BASED BROADCAST AUTHENTICATION OVERVIEW

If immediate authentication is not required, TESLA-based authentication technique is the recommended protocol that provides efficient broadcast authentication. All receivers are loosely time synchronized with the sender up to some errors. The sender splits up the time into uniform intervals and assigns a different key to each interval. It forms a one-way key chain by randomly choosing the key for the last interval and repeatedly applying a one-way hash function to derive the keys for the earlier intervals. Keys are revealed by the sender after an assigned disclosure delay of some number of intervals. The MAC of a message which is computed by the key of the current time interval is appended to the message and sent to the receiver. [3, 4, 5, 6, 9] The same packet (in case of TESLA) or a special packet (in case of μ TESLA) also reveals the key for an assigned earlier interval [9].

The receiver that has buffered the message of the earlier interval pops it and verifies the MAC with the revealed key and buffers the current message to wait for the corresponding key that will be disclosed after the next disclosure delay in future. Assuming a key disclosure delay of two time intervals, figure 1 shows that the sender uses key K_{i+1} to compute the MAC of the message M_{j+3} and reveals the key K_{i-1} in the same interval. [5]

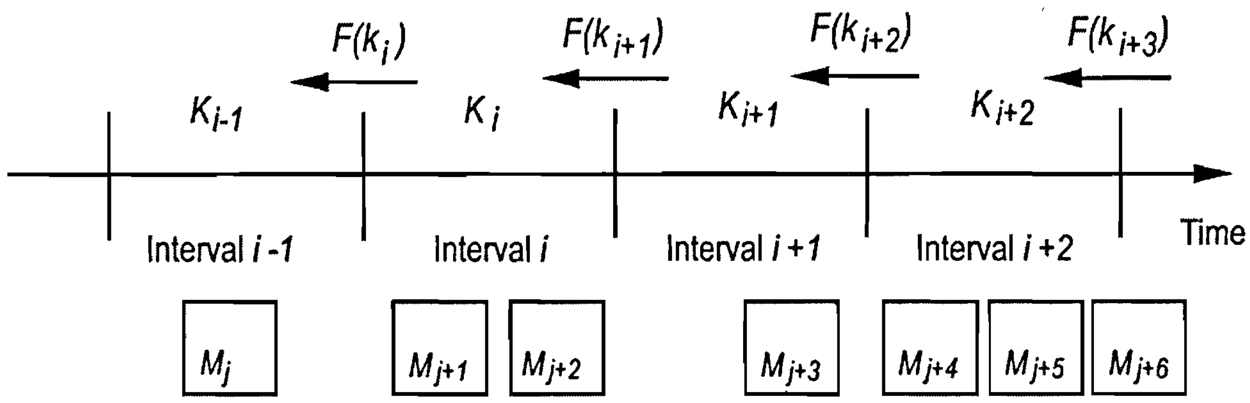


Figure 1. TESLA key chain (From Ref. [5])

Because all sensors are loosely synchronized and the receiver knows the schedule of disclosing keys, it can verify that the MAC key is still secret. When receiver is assured that the MAC key has not been yet published, it buffers the message for later verification. Using self-authentication, one way hash function and previously released keys each receiver can check if the disclosed key is correct. It can also use the key to verify the MAC of the buffered message. If channel is lossy or jammed and some keys are lost, they can be re-computed using later keys to check the authenticity of the earlier messages.

Prior to data transmission, all necessary initial parameters such as key disclosure delay and the duration of an interval must be deployed among all sensor nodes.

μTESLA is an extension to TESLA [2]. The only difference between TESLA and μTESLA is how the first key in the key chain which is called key chain commitment is distributed among the sensor nodes. TESLA uses expensive asymmetric cryptography to bootstrap new receivers, while μTESLA employs symmetric cryptography with a master key shared between the sender and each receiver to bootstrap the new receivers individually. In this arrangement, each node sends a request to obtain initial parameters. Once the request is received, sender transmits a packet containing the current time for synchronization purposes, the initial key, the key disclosure delay and duration of time intervals. [9]

D. SHORTCOMINGS OF TESLA-BASED TECHNIQUES

In Tesla-based techniques the receiver needs to buffer packets until the corresponding keys are disclosed in future packets for authentication purposes [7]. This hinders the nodes to have immediate authentication and to be able to use the received data as soon as they receive them. This may result in storage problems and buffer overflow in receivers. Moreover an adversary can flood the network with bogus messages that can cause the receiver to buffer unnecessary packets resulting in DoS attacks [8]. Denial of Service is the result of any action that prevents any part of a WSN from functioning correctly or in a timely manner. This attack is malicious, meaning it is intentional and not accidental. It disrupts service and is performed remotely or over the network. Furthermore, it is asymmetric since with little effort great harm could be done to the system.

Due to the nature of the system, nodes need to buffer and forward all the messages that they receive in one interval. This renders the whole network vulnerable to a malicious adversary that could intentionally flood the system by merely falsely claiming all sent messages belong to the current interval [5, 6]. This could easily lead to destructive DoS attacks in the system since sensor nodes are exceedingly resource-constrained and wireless transmission is costly. Undoubtedly, all these attacks are due to authentication delay of the broadcast messages.

CHAPTER III: PROTOCOLS

A. REDUCING μTESLA MEMORY REQUIREMENTS

First using protocol is Low-Buffer μTESLA protocol that is cited in [1]. This scheme intends to decrease the buffering of the receiver nodes by replacing it with little increase of sender buffering. In general, buffering at the sender side is often more acceptable since the sender in WSN is usually a base station such as a powerful laptop whereas the receiving nodes are small resource-constraint devices. Usually it is supposed that the base station is not susceptible to DoS attacks in these networks. This protocol is established on the basis that the receiver needs to buffer the message MAC instead of the usually larger message itself. The sender broadcasts only the message MAC first in an earlier interval and then sends the message itself along with the corresponding key in a later interval. The receiver needs to buffer the MAC and wait for the key in a later interval. Upon receiving the key and the original message in a later interval, receiver pops the corresponding MAC message and authenticates the current message with the previously buffered MAC and the currently received key. Since the receiver buffers the message MAC instead of the message, its buffer requirements will be lower hence more efficient.

Sender generates the key chain, buffers the message body, generates the message MAC and broadcasts it. In the later interval sender releases the encrypted buffered message, and the corresponding key. Receiver buffers the MAC, and in a later interval receives the original message and the key and verifies the message integrity by checking the MAC, the corresponding key and the message body.

B. IMMIDIATE AUTHENTICATION

Second using protocol is the immediate authentication μTESLA protocol. Buffering packets until the release of the corresponding delayed key for authentication purposes is a drawback of the original TESLA-based techniques which could also result in Denial of Service (DoS) attacks. Packets are subject to be dropped due to probable insufficient buffer in receiving

nodes. A new protocol cited in [2] discusses immediate authentication to allow the receiver to authenticate packets upon their arrival. They suggest replacing receiver buffering with sender buffering to eliminate the delay. Sender buffering which is structurally more powerful and reliable is believed to be more practical and reasonable in a pool of resource constraint receiving nodes in WSN. In this scheme the sender buffers the packets during one disclosure delay and each packet is designed to store the hash value of the data of a later packet. This design enables the data in the later packet to be immediately authenticated through the stored hash value as soon as the earlier packet is authenticated via the MAC of the message.
As illustrated in figure 2, the packet for the message M_j in interval T_j is constructed by first appending the hash value of the message M_{j+vd} to M_j and then by computing the MAC value over the appended values. Each packet consists of the message chunk of the current packet, the hash value of the data of a later packet, the MAC value over the current message concatenated by the hash value of the message of a later packet.

(1) P_j : $M_j + H(M_{j+vd}) + MAC(K_j, M_j \mid H(M_{j+vd})) + K_{i-d}$

"d" is the key disclosure delay which is the number of intervals after which a key is disclosed (unit is interval)

"v" is the number of packets sent out per time interval which is assumed constant for simplicity
The key of an earlier packet could be released in the same packet (in case of TESLA) or in a special packet (in case of μTESLA). The key used in MAC will be revealed as in the original TESLA scheme after an assigned disclosure delay.

When a node receives the packet P_{j+vd} which also discloses the key K_i, the packet P_j which was sent in previous interval could be authenticated. Since P_j contains the hash value of the data M_{j+vd} , if P_j is authentic, $H(M_{j+vd}))$ is also genuine and the message M_{j+vd} is immediately authenticated.

If a packet is lost or dropped, the later packet can still be authenticated using the MAC value. In this example if P_j is missed, P_{j+vd} will not be immediately authenticated; however, it still can be authenticated using the MAC value in future. This improves the reliability of the system which is integrated in the proposed idea in this paper.

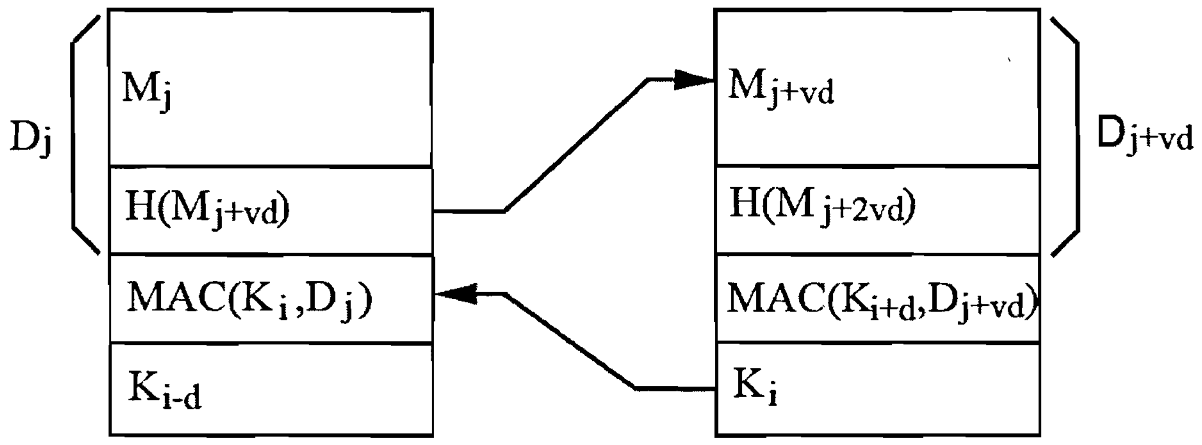


Figure 2. Immediate authentication packet example (From Ref. [2])

C. NEW PROTOCOL: RELIABLE, LOW BUFFERING IMMIDIATE AUTHENTICATION

The main idea in Low-Buffer μTESLA protocol [1] seems undoubtedly attractive since it intends to reduce the memory requirements for receiving nodes by simply substituting the buffering of the content of a message and the corresponding MAC by storing only the MAC. In this scheme, the more resourceful sender buffers the message chunk instead of the receiver and sends only the MAC of the message. The receiver simply needs to buffer the MAC until the disclosure of the original message along with the corresponding key. This provides full and efficient authentication without much buffering. Although this scheme increases the capacity of the node buffers and decreases the risk of buffer overflow in case of a DoS attack, it does not consider the factor of system reliability to a great extent and does not provide immediate authentication which is the main cause of the DoS attacks [2,8].

The basic observation of the new method is to use the MAC of the message for a more efficient receiver buffering while having the hash value of the data of the next packet stored in an earlier packet to maintain the reliability of the system and achieve immediate authentication.

Having the hash value of the data of a later packet in an earlier packet increases the probability of the authentication of a packet even if one of the packets is dropped or lost due to network flaws. At the same time, a later packet can immediately be authenticated through the MAC of its hash value as soon as an earlier packet is authenticated.

The protocol consists of two alternating general phases in terms of packet formatting. In the first phase which is during one specific interval, the current message chunk is concatenated with the hash of the message of a future packet and the MAC algorithm is computed over them using the current interval key. The packets of this form are sent to receivers as labeled "First Interval" in figure 3. In the next phase, during the next interval, the content of the message and the hash of the message of the future packet are sent in separate packets. In this interval the related key is also disclosed in the packet. The receiver applies this key to authenticate all the message packets sent in this interval.

Upon receiving the packets in the first alternating phase, receiver buffers the MAC and waits for next corresponding interval. Once the corresponding packets in the next phase are received, receiver pops the MAC and uses the interval key to authenticate the MAC which computed over the current message and the message in the upcoming packet. Once the MAC is verified, the hash of the data of the next coming packet is also immediately authenticated. As shown in figure 3, the packet in the first interval carries a hash of the data M_{j+vd}. If this packet is authentic, $H(M_{j+vd})$ is also authentic and the data M_{j+vd} is immediately authenticated. If this packet is lost, the next upcoming packet is not immediately authenticated; however, it could be verified later using the MAC value.

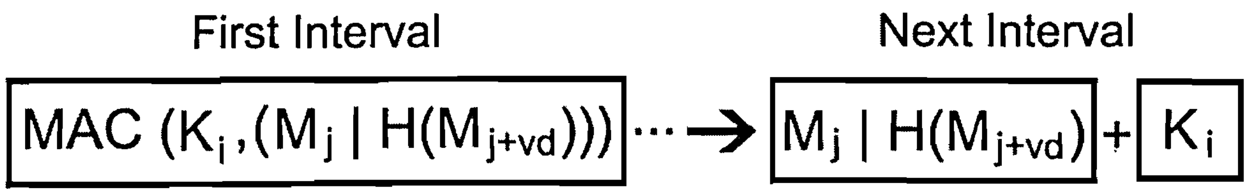


Figure 3. **Immediate authentication packet example**

Packets in the first phase contain the computed MAC over current message concatenated by hash of next message. In the next phase, packets consist of message contents and the key needed for authentication purposes. ("d" and "v" are used as explained in section III-B.)

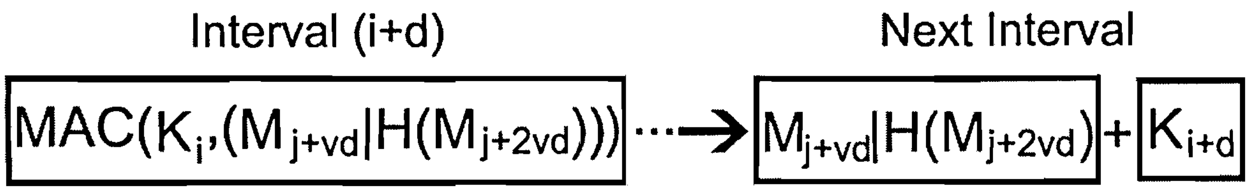


Figure 4. Immediate authentication of the next packets to come

Unlike standard μTESLA technique, only the MAC of the message is sent in the first interval instead of sending the large combination of message and the MAC. The receiver does not need to buffer the message and it is able to store only the MAC until the original message and the key arrive. Furthermore, placing the hash of the data of the next packets in previous packets increases the reliability of the system to a great extent. If the next upcoming packet is dropped or lost, it has already been authenticated using the previous combined MAC value. Furthermore, its message could also be retrieved using the hash of the data which was buffered at the time.

CHAPTER IV: OPNET MODELER OVERVIEW

The new protocol is best tried to be simulated and examined in a network simulator called OPNET. The Optimized Network Engineering Tools (OPNET) is a very powerful network simulator that is mainly applied to optimize cost, performance and availability. OPNET is a three-tiered technology with three main domains called network model, node model and the process model. Node model specifies the objects in network domain and the process model specifies objects in the node domain. [38, 39]

A. WORKFLOW

The workflow centers on the Project Editor. Network models are created in this editor and statistics are collected directly from each network object or from the whole network. Simulation originates from this editor and run parameters such as duration of the simulation are configured here.

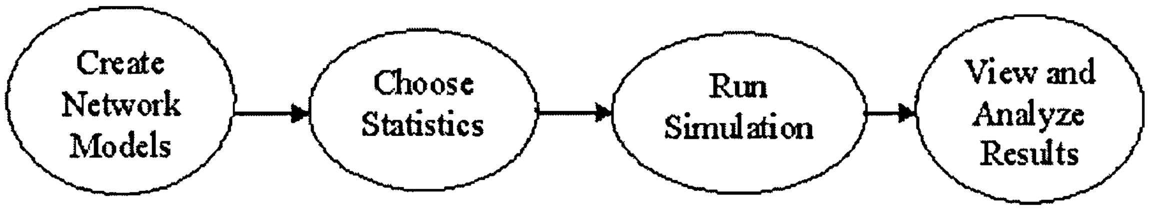


Figure 5. Modeler workflow (From Ref. [38])

B. EDITORS

1. Project Editor

Network model is created in Project Editor from where network simulation is initialized. In this editor the network topology is specified and nodes and links are configured. After choosing the results and running the simulation, the results are viewed from this editor.

2. Node Editor

Node models are created in Node Editor by specifying internal structures and capabilities. Node models are then used to create node instances within networks in the Project Editor. Nodes are defined by connecting various modules with packet streams and statistic wires. Each node may contain several modules that serve specific purpose, such as generating, queuing, processing, transmitting or receiving packets.

3. Process Editor

Process models are created in the Process Editor that control the underlying functionality of the node models created in the Node Editor. The purpose of this editor is to develop models of decision-making processes. Process models are represented by finite state machines (FSMs) consisting of several forced or unforced states with state transition among them. Operations performed in each state are described in embedded C or C++ code blocks. [39]

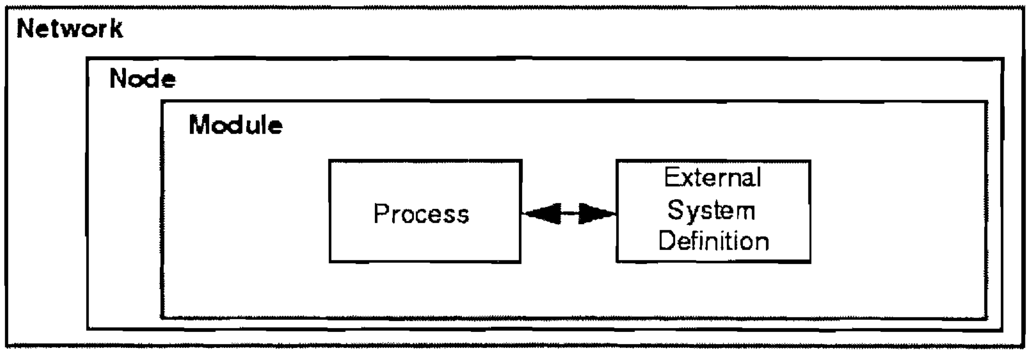
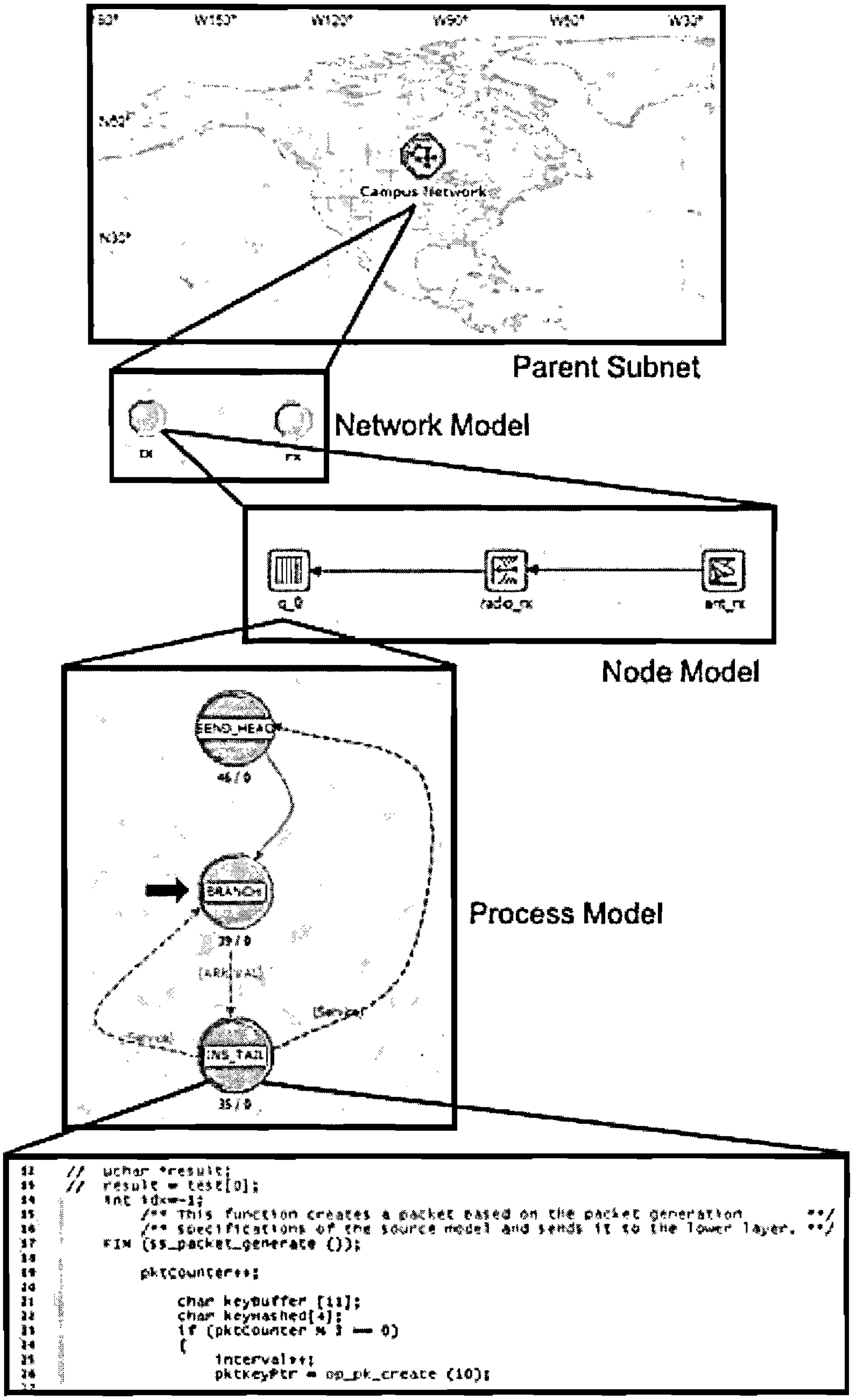


Figure 6. Hierarchical levels in models (From Ref. [39])



```
//  uchar *result;
//  result = test[0];
    int idx=-1;
        /** This function creates a packet based on the packet generation     **/
        /** specifications of the source model and sends it to the lower layer. **/
    FIN (ss_packet_generate ());

        pktCounter++;

            char keyBuffer [11];
            char keyHashed[4];
            if (pktCounter % 3 == 0)
            {
                interval++;
                pktKeyPtr = op_pk_create (10);
```

Figure 7. Three-tiered OPNET hierarchy

Although project is originated from Project editor, creation flow of the project is from Process model to Node model ending with the Network model. Once a model is created and saved, it will be added to a list where can be accessed from an upper level editor. Project is initially created by building a Process model. Once the Process model is finished and saved, it will be appended to a list accessible from the Node editor. A Node model is built by linking to the previously built Process model existing in the list. Similarly, Network model is built by linking the previously created Node model.

C. OBJECT TYEPS

1. Network Model

In simulating the current scenario a wireless network topology is applied and several types of nodes are employed accordingly in the Project, Node and Process Editors. A network terminal or device is among different kinds of nodes that may exist in the Project Editor. These nodes have the ability to communicate to other nodes and other capabilities determined by its model.

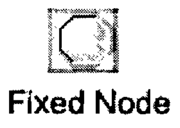


Figure 8. Project Editor - Node in network model

2. Node Model

Node Editor that is used to specify the structure of the device models contains different object types. The nodes existing in the node models are composed of several various types of objects or modules. Each module is a black box with a particular function of node's operation. These modules can be connected by different types of connections such as packet stream to support data flow. Processor is a general purpose programmable object that a process model specifies its behaviour. Queue is similar to a processor in addition to providing internal packet queuing facilities consisting of a bank of sub-queues that are ordered lists of packets. Wireless radio transmitter and receiver allow packets to be sent outside and received of and from the node's boundary. Packet Stream connects an output stream of a source module to the input stream of a destination module. This allows packets to be communicated and buffered between the nodes.

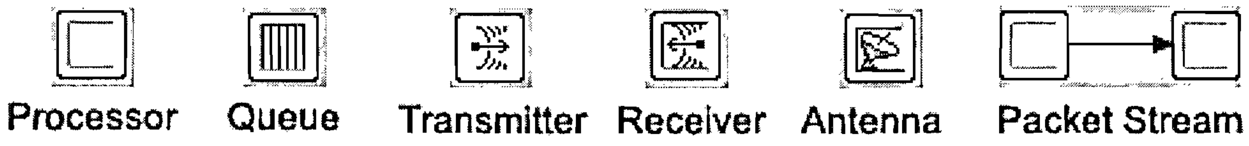


Figure 9. Node Editor – Modules in node model

3. Process Model

The Process Editor is used to specify the behaviour of process models. Process models are instances of processes in the Node Domain that use finite state machine paradigm to illustrate the system behaviour in case of an event. The objects used to build process models are states that represent a process mode. States contain codes that are performed when a state is entered, exited or when an interrupt occurs. These codes include C++ or C program along with the OPNET Porto C language. The OPNET specific language provides predefined procedures such as packet generation, packet send and packet receive.

A red state unlike a green state is an unforced state where the process is blocked immediately upon executing the enter code and the system waits for a new interrupt before continuing. A process can go from a source state to a destination state if a condition is served. An executive statement which specifies an action is performed once the transition is taken. A forced or green state is immediately exited or transited to the next state once it is entered.

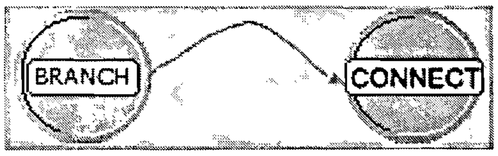


Figure 10. Process Editor – States in process model

CHAPTER V: OPNET SIMULATION

In order to investigate the new proposed protocol, two scenarios are simulated and compared in OPNET. Firstly the μTESLA scheme mentioned in [2] that has improved the original μTESLA technique by facilitating immediate authentication through sending the hash of a future packet with the current packet is implemented in OPNET. Furthermore a second scenario is simulated in the same environment that incorporates the first scenario with the memory efficient μTESLA protocol mentioned in [1] by buffering the MAC of the message instead of the original massage in the resource constrained receiver node. The integration of these two methods is hoped to achieve immediate authentication while at the same time optimally reduce receiver buffering requirements.

As in other μTESLA implementations, several assumptions are made in the creation of these protocols. Firstly, it is assumed that all the nodes are equipped with initial parameters and functions such as key disclosure delay and the hash function and MAC algorithm needed for encryption and decryption purposes. Furthermore, "v" that is the number of packets sent out per time interval is assumed to be constant and equal to two for simplicity. Similarly, key disclosure delay or "d" is arranged to be two, meaning the key needed for decrypting a packet is revealed after two intervals.

Hash functions are used to ensure data integrity. In this simulation, RSHash is the hash function that is used for hashing the key and the message of the future packet for immediate authentication purposes. The function takes a string as a parameter and iteratively applies some additions and multiplications to its character's ASCII code with two random integer numbers. The result number is the encrypted form of the passed string which could be an interval specific key or the hash of the data of a packet in future.

Encryption algorithms are applied to ensure data confidentiality. Much secure HMAC (Hash-based Message Authentication Code) which is a cryptographic hash function with a secret key is used as the encryption algorithm in this simulation. The hash function used in HMAC algorithm is SHA1. Four functions are used in creation of the HMAC-SHA1. As a result, a "Digest" is produced and passed to the packet generation function. Sender encrypts the message with this code and passes the "Digest" along with packet. Once the receiver owns the key and the original message, it applies the HMAC algorithm to the message with the related key. If the two "Digests" are the same, the packet is authenticated.

To investigate the enhancement of the broadcast authentication in sensor networks in OPNET, wireless network technology is employed. One transmitter node and one receiver node are planted in the Project Editor and they are connected wirelessly as shown in figure 11. Two scenarios are designed and simulated in OPNET to analyze and compare the new protocol's efficiency and performance. Each transmitter and receiver node contains series of C++ code to implement the assigned scenarios. General and expanded depiction of the sender and receiver nodes are demonstrated in the next section without considering the scenarios and then detailed description of each scenario is followed. The complete OPNET collected code for the second scenario which is the proposed scheme is given in the Appendices section.

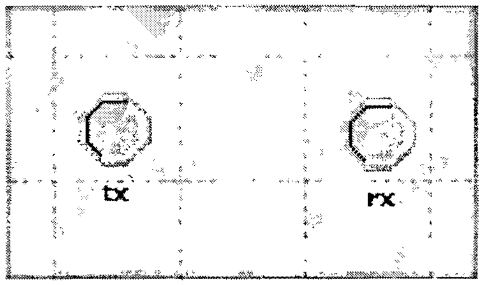


Figure 11. Project Model consisting of a transmitter and a receiver

A. TRANSMITTER

Each transmitter node consists of three modules, a processor, a radio transmitter and an antenna. The processor module in this node is where the simulation is controlled and directed. The underlying strategy is built within the Process Model of the transmitter node. The radio transmitter and the antenna facilitate transition of data in a wireless environment by using radio links.

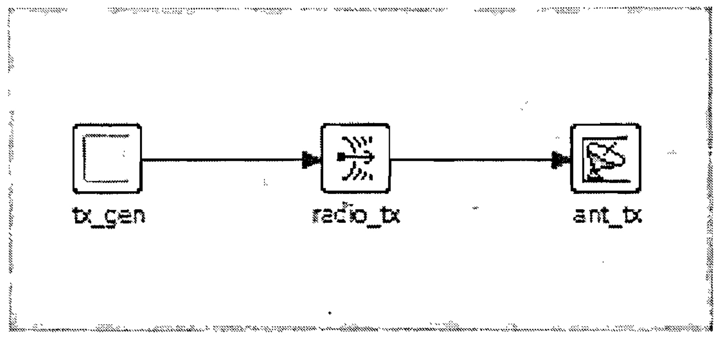
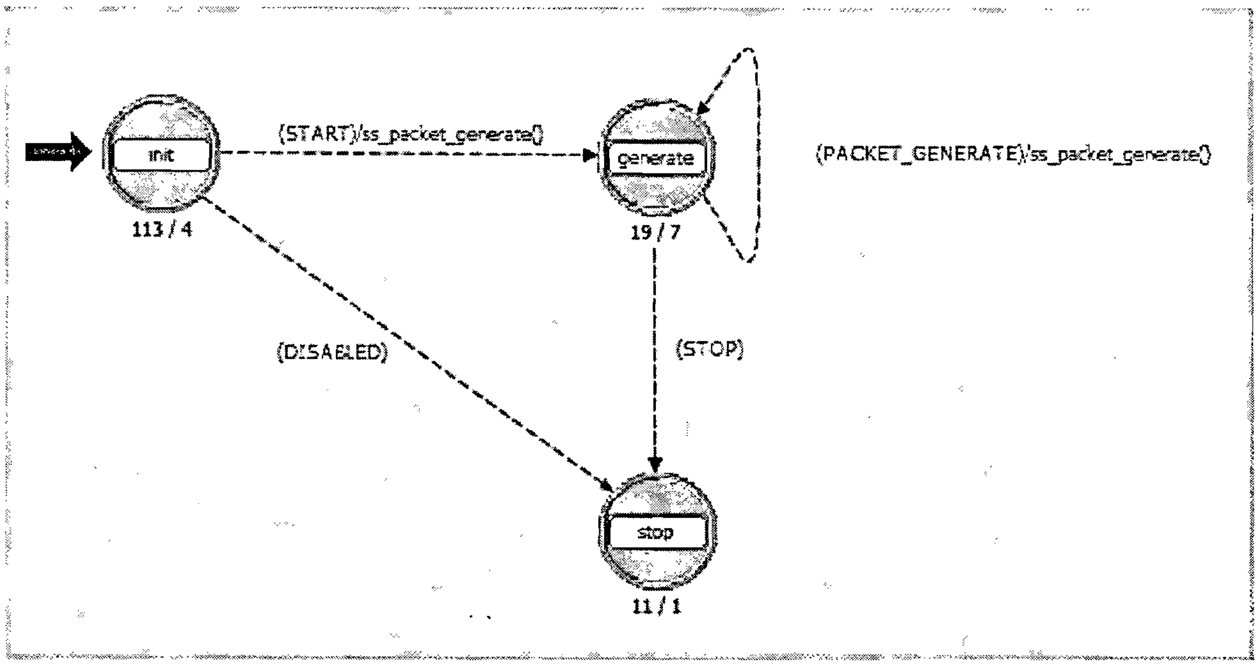


Figure 12. Node Model for the transmitter node

The Process Model of the transmitter processor is composed of three states as illustrated in figure 13. Several initial variables are set in the initial state. If the start condition is true, a packet is generated by calling the function ss_packet_generate() which lies in the function block of the editor. Now the process transits to the "generate" state where packets are continuously generated in a loop until a "stop" interrupt occurs or the simulation time is terminated.

Embedded C or C++ code is placed in various places in Process Model. Header block and function block and both enter and exit parts of each state could contain necessary coding in order for the system to work as desired. The content of header block and function block is available throughout the node and accessible by all the modules of the node. Some variables are initialized in the header block and several functions and procedures are defined in the function block of the sending node of both scenarios.

The hash and MAC functions are implemented in the function block in order to be used for packet formatting during the generation of the packets. The "ss_packet_generate" function is also created in the function block. The formatting of the packets is different in two scenarios; therefore, the code for packet generation at the sender's end and packet processing at the receiver's end differ accordingly and are discussed shortly.



init
113 / 4
(START)/ss_packet_generate()
generate
19 / 7
(PACKET_GENERATE)/ss_packet_generate()
(DISABLED)
(STOP)
stop
11 / 1

Figure 13. Process Model for the transmitter processor module

B. RECEIVER

Each receiver node is composed of a queue, a radio receiver and an antenna as represented in figure 14. The queue processor is the heart of the receiving plot. Packets selectively get inserted in the queue and removed for target authentication. The existing states in this module facilitate optimum and efficient authentication by carrying out the necessary functions. The queue is an OPNET built-in module that performs based on first in first out (FIFO) processing technique.

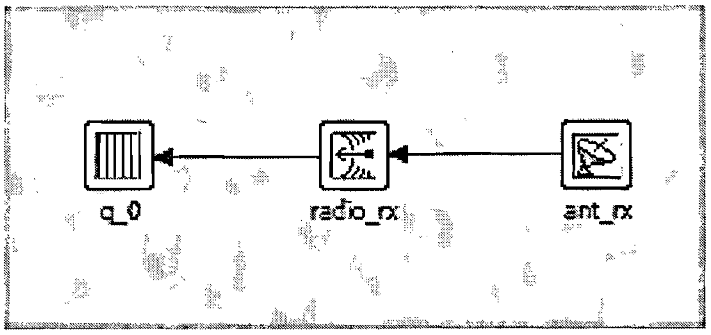


Figure 14. Node Model for the receiver node

As illustrated in figure 15, the initial state is marked with a black arrow. This state accepts packets from the transmitter and since it is a green or forced state, it automatically moves the process to the next state "PROCESS" at the bottom of the figure. This state examines the packet to see if it is a packet containing key, MAC or message.

In case of the first scenario, if the packet is composed of a message, it will be pushed into the queue and the state will be transited back to the initial state. If packet contains a key, the process will be moved to the third state "AUTH" at the top of the figure 15. In the case of the second scenario, the packet is pushed into the queue if it contains a MAC and then transferred to the initial state and it will be moved to the third state if it contains a message. Figure 15 and 16 demonstrate the finite state machine mechanism and the corresponding flowchart illustrating the transitions between states on the receiver side.

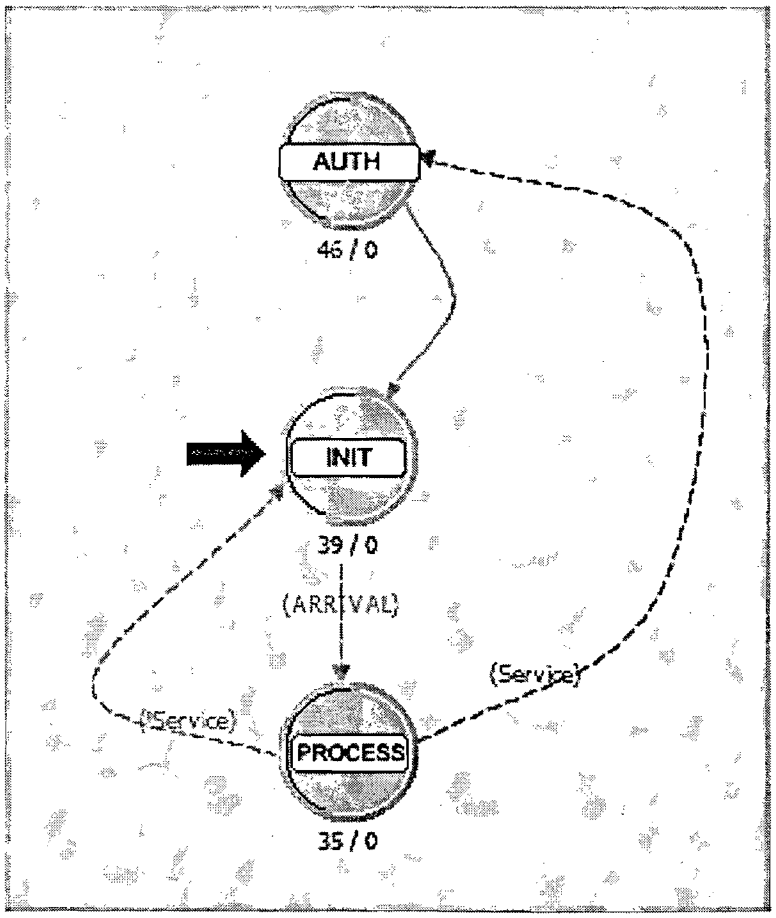


Figure 15. Process Model for the receiver queue module

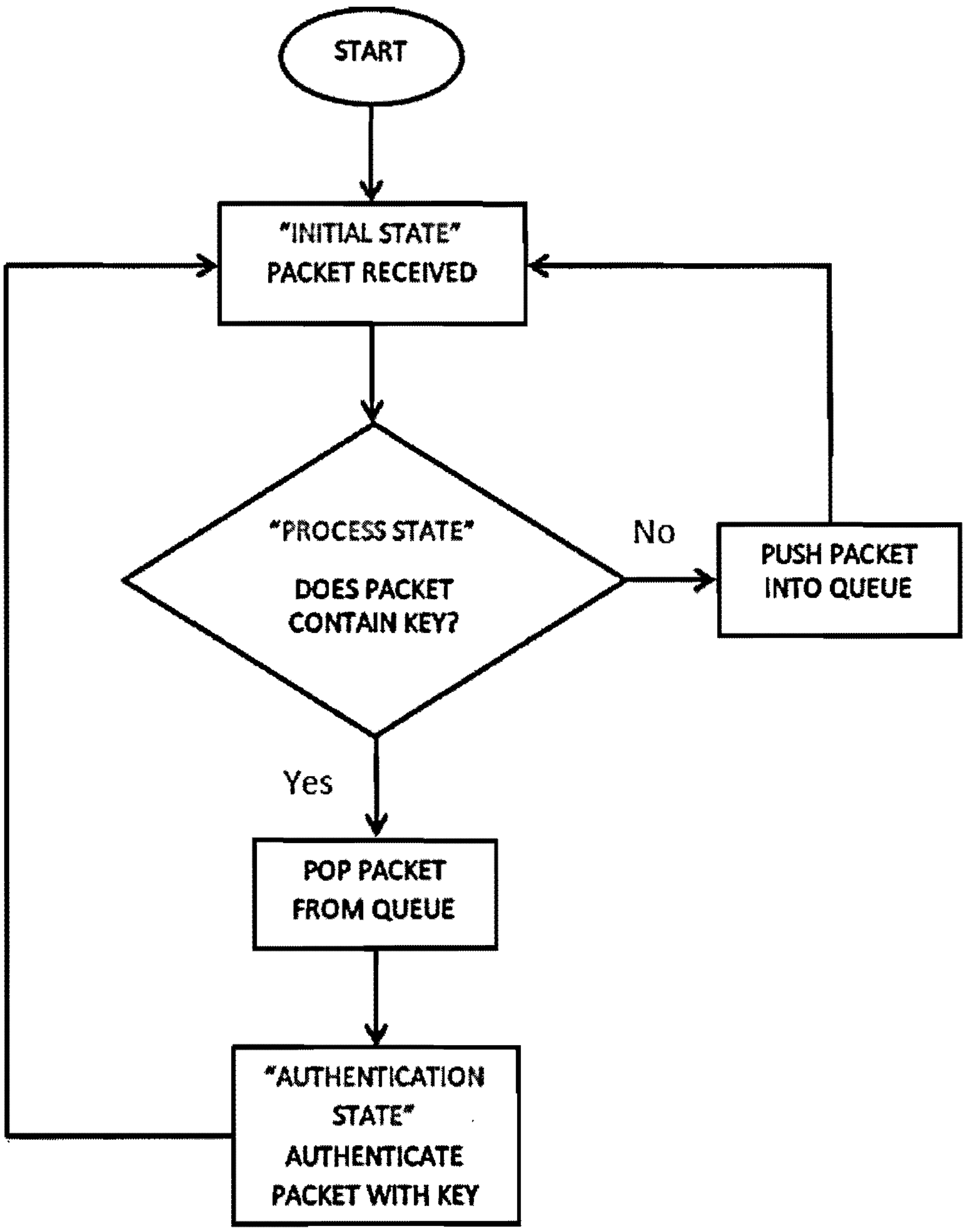

START
"INITIAL STATE" PACKET RECEIVED
"PROCESS STATE" DOES PACKET CONTAIN KEY?
No
PUSH PACKET INTO QUEUE
Yes
POP PACKET FROM QUEUE
"AUTHENTICATION STATE" AUTHENTICATE PACKET WITH KEY

Figure 16. Flowchart of state transition of a receiver node

C. FIRST SCENARIO (IMMEDIATE AUTHENTICATION PROTOCOL)

In this scenario, immediate authentication μTESLA protocol is re-created without considering low memory requirements. In this scheme, in each interval, two data packets are sent along with a separate packet containing the key for the two previous intervals. For the first two intervals the key packet contains no information. As the mechanism of the μTESLA suggests, receiver is supposed to buffer these four packets in queue and wait for the related key in the upcoming two intervals. Once a key is received, packets are popped from the queue in FIFO manner and are authenticated with the related key.

Messages are removed from the queue once they are authenticated. System disposes the packets that are not authenticated. Each packet contains the hash of a future packet according to the formula ($M_{j + vd}$). This facilitates immediate authentication of a packet sent in the future. Formula 2 and 3 are the formats of the two packets sent in each interval and figure 17 is a sketch of packet transmitting first scenario.

(2) P_j : $M_j + H(M_{j+vd}) + MAC(K_j, M_j \mid H(M_{j+vd}))$

(3) $K_i : K_{i-d}$

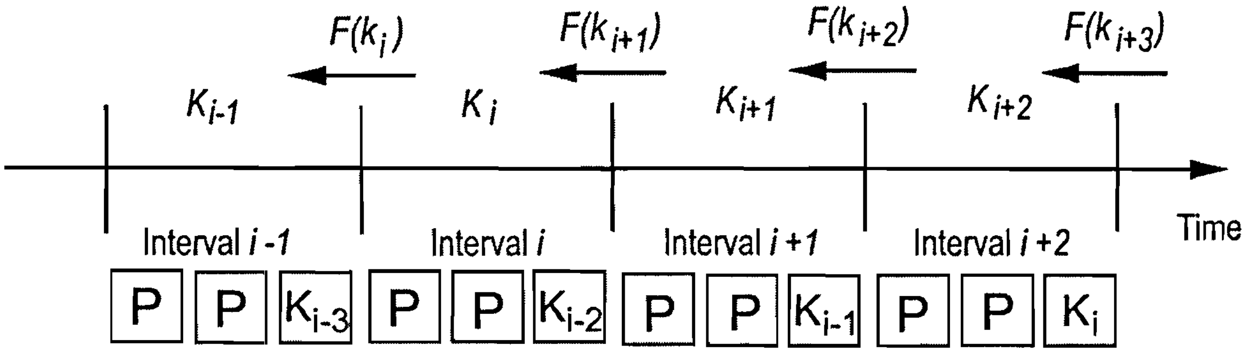


Figure 17. Scenario 1

D. SECOND SCENARIO (PROPOSED PROTOCOL)

The new proposed integrated protocol is implemented as the second scenario. This arrangement is composed of two alternating intervals in terms of packet formatting. Packets alter their format every two intervals. Two data packets are sent in each interval in the first two intervals. In this packet only the MAC of the massage concatenated by the hash of the future packet is sent. In the next two intervals the corresponding messages of the first four MAC are sent along with the key to the previously sent MAC's.

Sending the MAC of the messages first allows less buffering in the receiver side. Once the corresponding message and key is sent, the receiver pops the MAC from the queue in FIFO manner and authenticates the message. As in first scenario, once the current packet is authenticated, a future packet is also authenticated. Packet formats are outlined in the following lines and a sketch of packet forwarding is presented in figure 18.

First alternating format:

(4) $F_1 = P_j : MAC(K_j, M_j \mid H(M_{j+vd}))$

Second alternating format:

(5) $F2 = P_j : M_j + H(M_{j+vd})$

(6) $K_i : K_{i-d}$

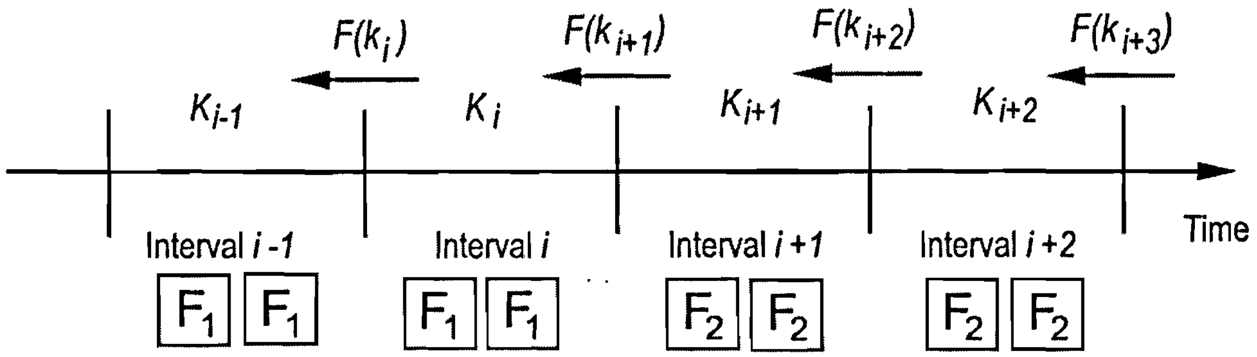
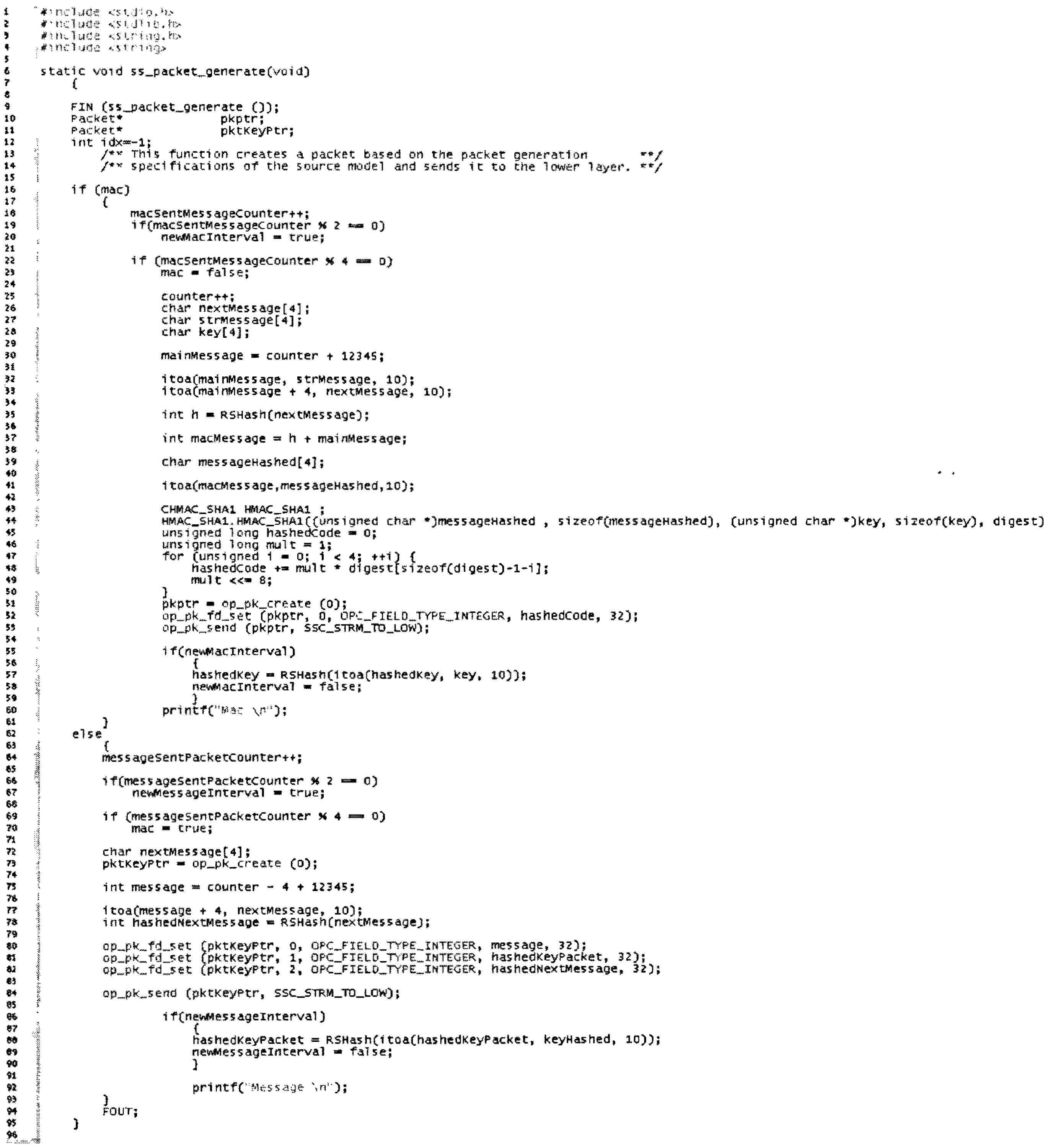


Figure 18. Scenario 2

Similar to first scenario, the function block for the transmitter node consists of implementation of hash function and MAC. In this block, packet generation function employs these procedures when it needs to obtain hash of a message or when it needs to encrypt the message with the MAC algorithm. Figure 19 is a snap shot of part of the function block of the transmitter node in the second scenario. It shows the content of ss_packet_generate() procedure.

In this code, packets are generated according to the design and sent to the receiver node to be processed. This procedure checks to see which format needs to be transmitted. If it is time to send the message, the packet containing the key also needs to be accompanied with the other packets. Otherwise only the MAC of the message concatenated with hash of the future packet is transmitted to the receiver node. This procedure uses formula 4 and 5 to create appropriate packet formats.

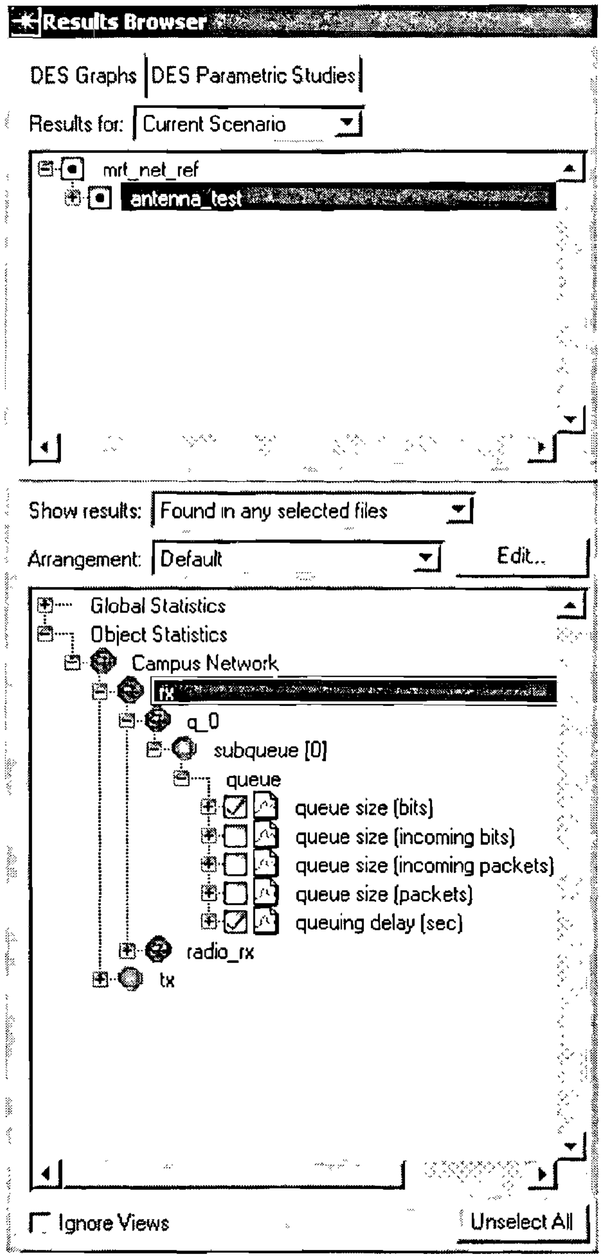


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>

static void ss_packet_generate(void)
    {

    FIN (ss_packet_generate ());
    Packet*             pkptr;
    Packet*             pktKeyPtr;
    int idx=-1;
        /** This function creates a packet based on the packet generation       **/
        /** specifications of the source model and sends it to the lower layer. **/

    if (mac)
        {
            macSentMessageCounter++;
            if(macSentMessageCounter % 2 == 0)
                newMacInterval = true;

            if (macSentMessageCounter % 4 == 0)
                mac = false;

                counter++;
                char nextMessage[4];
                char strMessage[4];
                char key[4];

                mainMessage = counter + 12345;

                itoa(mainMessage, strMessage, 10);
                itoa(mainMessage + 4, nextMessage, 10);

                int h = RSHash(nextMessage);

                int macMessage = h + mainMessage;

                char messageHashed[4];

                itoa(macMessage,messageHashed,10);

                CHMAC_SHA1 HMAC_SHA1 ;
                HMAC_SHA1.HMAC_SHA1((unsigned char *)messageHashed , sizeof(messageHashed), (unsigned char *)key, sizeof(key), digest)
                unsigned long hashedCode = 0;
                unsigned long mult = 1;
                for (unsigned i = 0; i < 4; ++i) {
                    hashedCode += mult * digest[sizeof(digest)-1-i];
                    mult <<= 8;
                }
                pkptr = op_pk_create (0);
                op_pk_fd_set (pkptr, 0, OPC_FIELD_TYPE_INTEGER, hashedCode, 32);
                op_pk_send (pkptr, SSC_STRM_TO_LOW);

                if(newMacInterval)
                    {
                    hashedKey = RSHash(itoa(hashedKey, key, 10));
                    newMacInterval = false;
                    }
                printf("Mac \n");
        }
    else
        {
        messageSentPacketCounter++;

        if(messageSentPacketCounter % 2 == 0)
            newMessageInterval = true;

        if (messageSentPacketCounter % 4 == 0)
            mac = true;

        char nextMessage[4];
        pktKeyPtr = op_pk_create (0);

        int message = counter - 4 + 12345;

        itoa(message + 4, nextMessage, 10);
        int hashedNextMessage = RSHash(nextMessage);

        op_pk_fd_set (pktKeyPtr, 0, OPC_FIELD_TYPE_INTEGER, message, 32);
        op_pk_fd_set (pktKeyPtr, 1, OPC_FIELD_TYPE_INTEGER, hashedKeyPacket, 32);
        op_pk_fd_set (pktKeyPtr, 2, OPC_FIELD_TYPE_INTEGER, hashedNextMessage, 32);

        op_pk_send (pktKeyPtr, SSC_STRM_TO_LOW);

                if(newMessageInterval)
                    {
                    hashedKeyPacket = RSHash(itoa(hashedKeyPacket, keyHashed, 10));
                    newMessageInterval = false;
                    }

                    printf("Message \n");
        }
        FOUT;
    }
```

Figure 19. Scenario 2 - Function block of the transmitter node

CHAPTER VI: SIMULATOIN RESULT

The duration of the simulation and other run related parameters could be set before running the project from the Project Editor. After the simulation is completed different results could be filtered in the result browser.



Results for: Current Scenario

mrt_net_ref
antenna_test

Show results: Found in any selected files

Arrangement: Default

Global Statistics
Object Statistics
Campus Network
rx
q_0
subqueue [0]
queue
✓ queue size (bits)
queue size (incoming bits)
queue size (incoming packets)
queue size (packets)
✓ queuing delay (sec)
radio_rx
tx

Figure 20. Result browser

Following figures illustrate the performance of the proposed node authentication protocol explained in (III-C) as the first scenario along with the protocol stated in section (III-B) as the second scenario in the course of 30 minute simulation. Although immediate authentication is achieved by these protocols, it cannot be captured in a graph. As expected, the following graphs show that the queue performance of the receiving node is improved to a good extent proving that the integration of Low-Buffer μTESLA version with the previous work is optimal.

As explained previously, two separate projects were implemented in OPNET in order to compare and investigate the enhancement of the proposed scheme. The first project implements the immediate authentication μTESLA without considering sensor buffer requirements. The second project is the implementation of the proposed integrated protocol that incorporates the protocol in the first scenario with the Low-Buffer μTESLA scheme in order to improve system performance.

Figures 21 and 22 demonstrate the graphs generated by OPNET. They display the average of queue size in bits in the first and second scenarios respectively. The X axis is time in minutes in these graphs and the Y axis is the average queue size in bits. As illustrated in the following two figures, the number of bits waiting in the queue in the new protocol stabilizes with less than one third of the bits in the queue of the first scenario. This shows an improvement of around 73% in queue size in the new protocol.

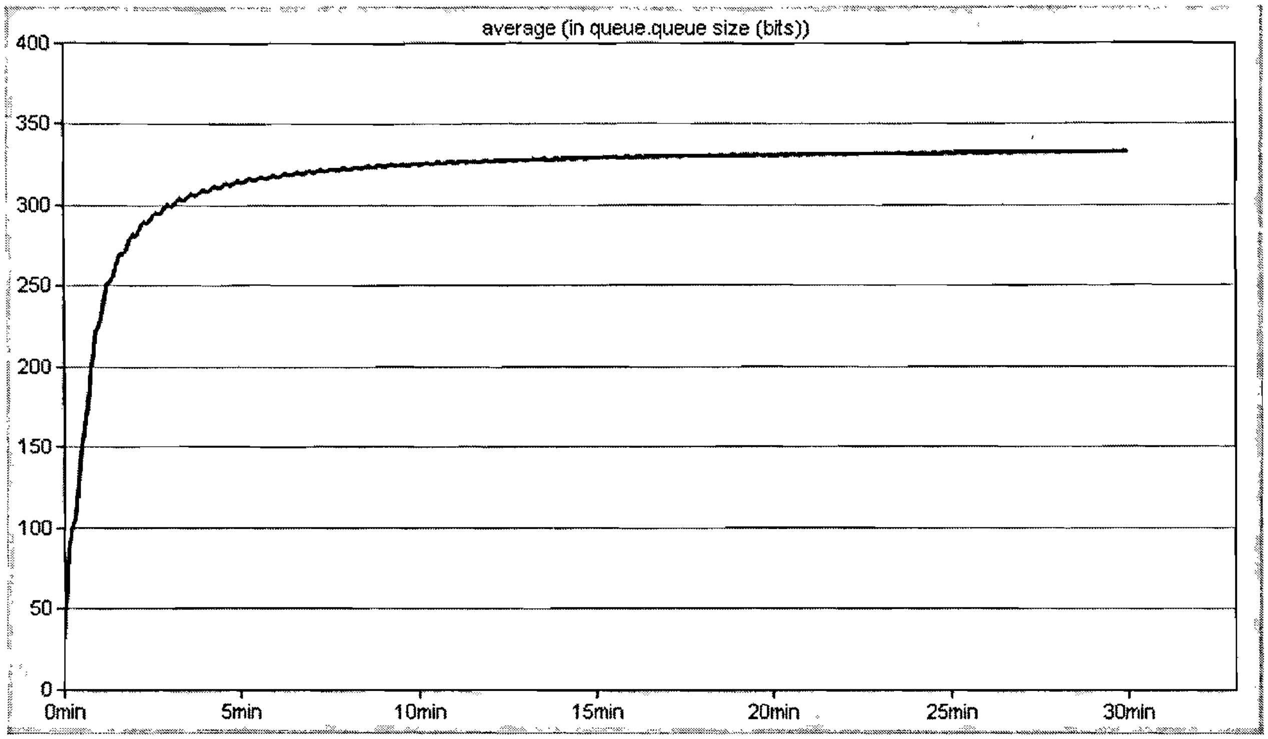


Figure 21. Scenario 1 – Queue size stabilizes at around 330 bits

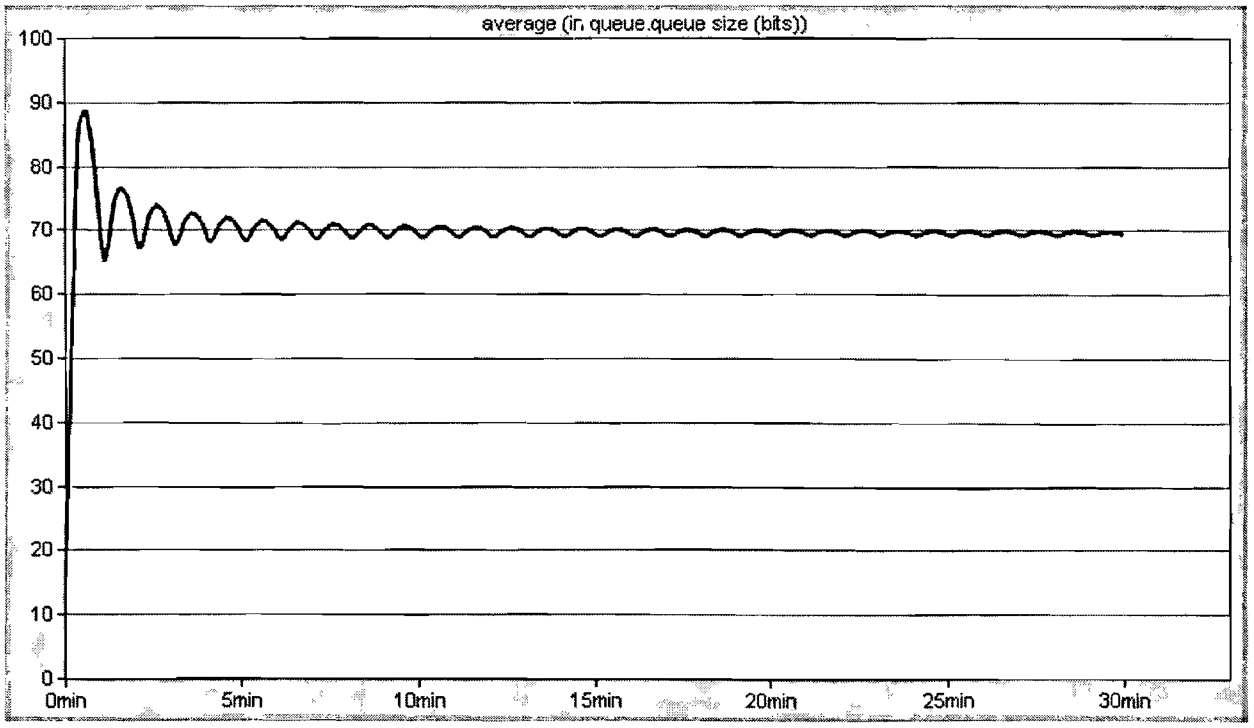


Figure 22. Scenario 2 – Queue size stabilizes at around 70 bits

Figures 23 and 24 also generated by OPNET simulator, illustrate the average of queuing delay in seconds in the first and second scenario respectively. The X axis is time in minutes and the Y axis is the average queue delay in seconds. As represented in figure 23 and 24, the queuing delay in the new protocol stabilizes with a number much less than the one in first scenario which displays an improvement of around 37% in delay time in the new scheme.

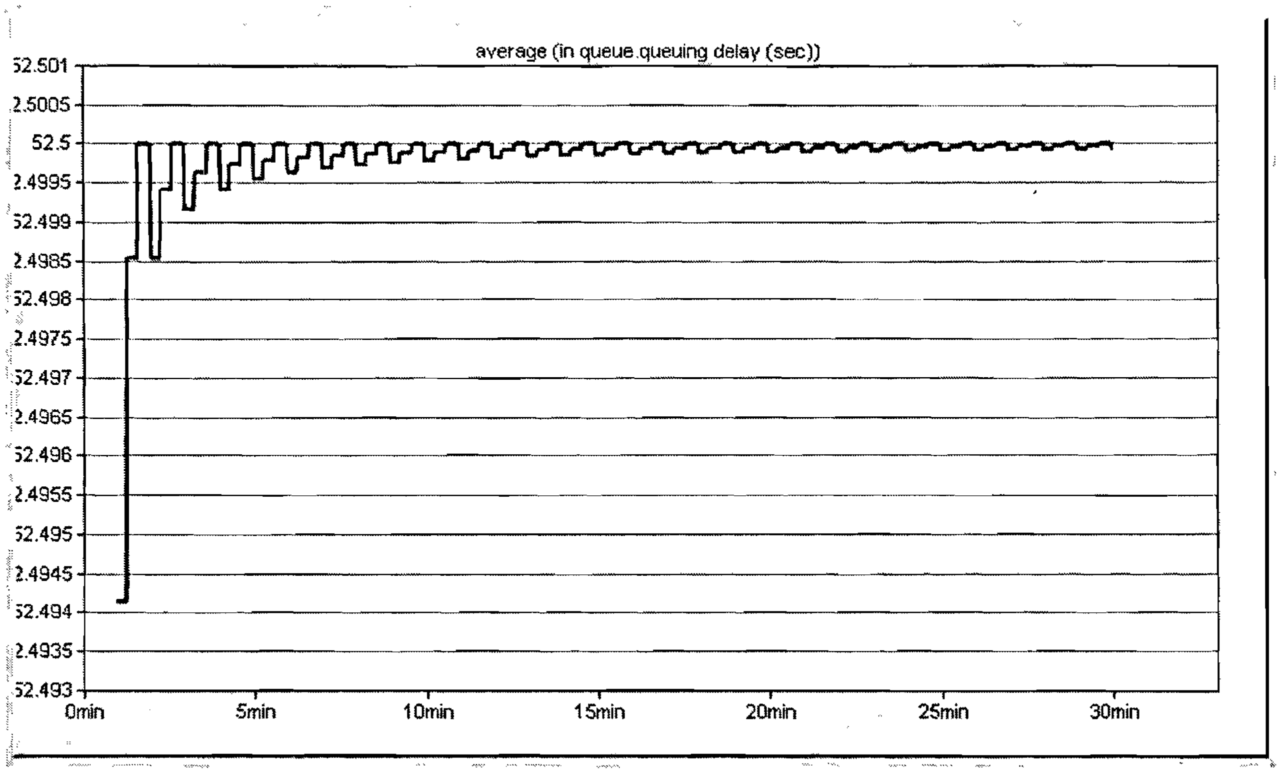


Figure 23. Scenario 1 – Queuing delay stabilizes at 52.5 seconds

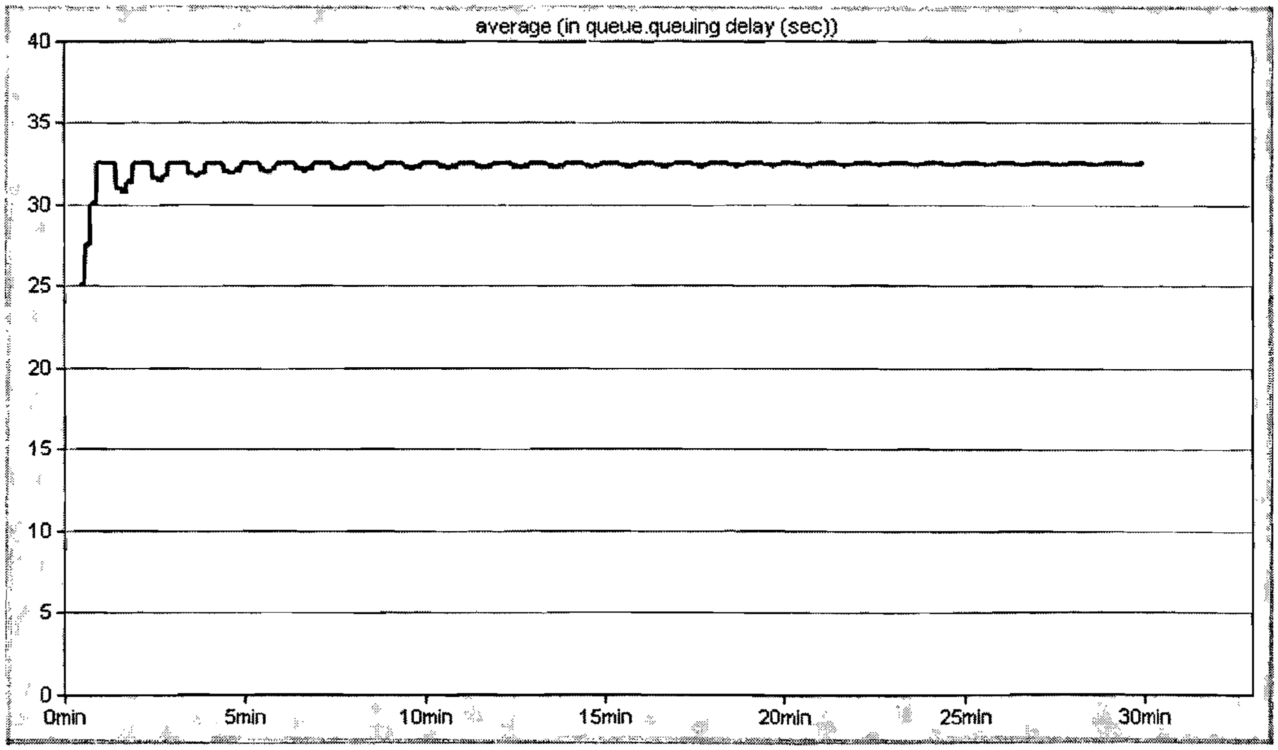


Figure 24. Scenario 2 – Queuing delay stabilizes at 33 seconds

Figure 25 is a combined graph displaying both scenarios' queue size trend in time. Similarly figure 26 is a depiction of both scenarios' queuing delay over time. The two plotted lines in each graph represent the two scenarios initially produced in OPNET. These comparison graphs demonstrate a significant improvement in both buffer size and buffering delay in the receiving node of the proposed scheme.

As the result of this protocol, the buffer requirements of the resource-restricted receiving sensors are well taken care of. Furthermore, the queuing delay which is the time packets need to spent in queue in order to be processed is significantly reduced resulting in a more robust and reliable system. Overall, the new scheme achieved to enhance memory requirements of the low capacity sensing devices while preserving the desirable immediate authentication among sensors.

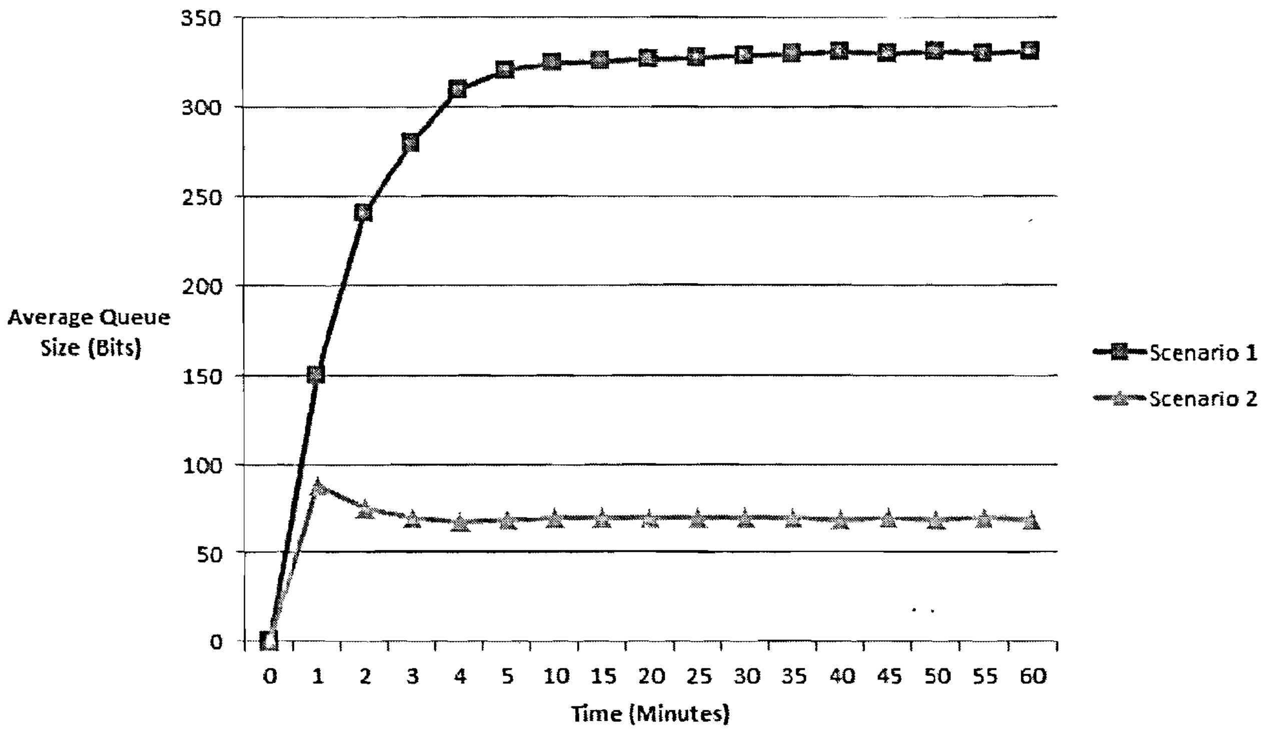


Figure 25. Comparison of average queue size in both scenarios

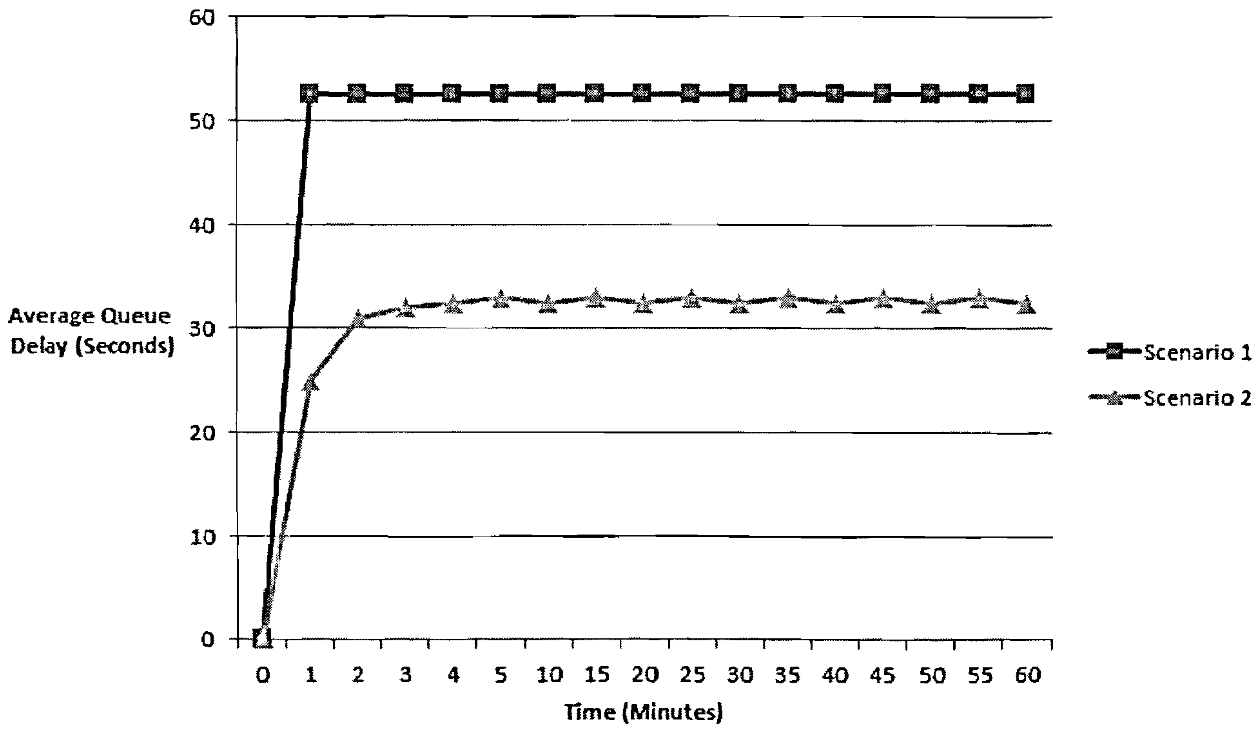


Figure 26. Comparison of average queuing delay in both scenarios

CHAPTER VII: CONCLUSION

This project proposed a new authentication mechanism for a wireless sensor network designed to ensure system integrity and dependability. This scheme applies a simple symmetric cryptography and improves its generally low security capabilities by revealing the cryptographic key with a delay. Since the delay makes the sensor nodes subject to outside attacks, this protocol attempts to remove authentication delay by sending future messages in earlier packets. This might introduce the need for higher buffer capacity which is a commodity in sensor networks. To alleviate this problem, this mechanism is designed to send smaller version of message in form of MAC to enable low-capacity receiving nodes to buffer less information.

This protocol combines two separate authentication mechanisms to provide robust security coupled with high reliability and utilizes affordable data encryption to ensure data confidentiality. Performance and simulation analysis demonstrated that the buffer size requirements and delay of the proposed system are enhanced to a good extent while immediate authentication is achieved to make the whole system highly resistant to outside malicious attacks such as Denial of Service attack.

A. CONTRIBUTIONS OF THIS PROJECT

This project accomplished the objective of incorporating desirable immediate authentication with careful consideration of low-buffer requirements of the sensor nodes in wireless sensor networks. It proposed an integration of two protocols in order to enhance broadcast authentication in sensor networks. This scheme combines a version of μTESLA technique that provides immediate authentication with a reduced buffer version of the same technique.

The proposed integrated protocol is simulated in OPNET modeler along with one of the previously stated scheme to verify the efficiency and the reliability of the new system. Performance and simulation analysis demonstrated that the queue delay and the queue size of the

PROPERTY OF
RYERSON UNIVERSITY LIBRARY

proposed system are enhanced in comparison with the isolated techniques. As expected, the buffer size is reduced due to the elimination of the message buffering and replacing it with the buffering of message MAC instead. At the same time, immediate authentication which is the principal resolution to DoS attacks is achieved to a great extent due to the appending of the hash of the data of the next upcoming packet. Having the hash of the next data content in the current packet allows immediate authentication of the next packet once the current packet is verified. Also addition of this feature is expected for the betterment of the system reliability since both the message and the hash of the message are sent separately in two different intervals. This increases the chance of receiving the message one way or the other in case of channel loss.

B. RECOMMENDATIONS FOR FUTURE WORK

This project proposed a real-time authentication protocol; however several assumptions have been made to simplify the simulation in the modeler. For instance, the number of packets sent out per time interval or the parameter "v" that is assumed constant for simplicity could be attempted to be random and variable to re-create a more realistic network environment.

In a real life wireless sensor networks, all nodes should firstly be deployed with the necessary parameter values such as key disclosure delay, duration of time interval, the initial key and the current time and also the necessary hash functions and MAC algorithms. This initial realm of process is yet another issue in sensor networks that is a base for on-going research. In this simulation, it is assumed that all nodes are equipped with all the required initial parameters. Future simulation could include this initial stage to create a thorough scenario for sensor authentication.

Future work should include a complete implementation of a distributed sensor network with more receiving sensors and more than one base station. Besides the base station broadcasting packets, receiver nodes could also be relaying received information to other nodes. As part of this follow-on work, the traffic analysis of all the nodes in the network could be expanded. Various parallel and individual statistics of all the nodes as well as the system as a whole in longer durations could be investigated.

In the process of simulating the network more accurately, the environment could be programmed to reflect lifelike events such as loss of packets and malicious jamming. Future work could include various wrongful events that might occur in the course of simulation in order to observe system behavior in case of a more realistic plot. A thorough security solution would address different security challenges, such as detecting misbehaving or compromised nodes.

APPENDICES: SOURCE CODE FOR SECOND SCENARIO

APPENDIX A: TRANSMITTER SOURCE CODE

```
/* Process model C++ form file: simple_source.pr.cpp */
/* Portions of this file copyright 1986-2009 by OPNET Technologies, Inc. */


/*
 ========================== NOTE ==========================
 This file is automatically generated from simple_source.pr.m
 during a process model compilation.

 Do NOT manually edit this file.
 Manual edits will be lost during the next compilation.
 ========================== NOTE ==========================
*/

/* This variable carries the header into the object file */
const char simple_source_pr_cpp [] = "MIL_3_Tfile_Hdr_ 150A 30A modeler 7 4D227972
4D227972 1 rye-udcmybdtq7r Administrator 0 0 none none 0 0 none 0 0 0 0 0 0 0 0 21b7 3
";
#include <string.h>



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

/* Header Block */

/* Include files.                                  */

#include <string>
#include        <oms_dist_support.h>
#include "hmac_sha1.hpp"

#include "sha1.hpp"

/* Special attribute values.          */
#define         SSC_INFINITE_TIME              -1.0
#define WIRELESS_STRM                          0
```

```
/* Interrupt code values.                    */
#define      SSC_START                    0
#define      SSC_GENERATE                 1
#define      SSC_STOP                     2

/* Node configuration constants.      */
#define      SSC_STRM_TO_LOW                         0

/* Macro definitions for state          */
/* transitions.                                    */
#define      START                            (intrpt_code == SSC_START)
#define      DISABLED                 (intrpt_code == SSC_STOP)
#define      STOP                     (intrpt_code == SSC_STOP)
#define      PACKET_GENERATE                  (intrpt_code == SSC_GENERATE)
#define      uchar unsigned char

/* Function prototypes.                            */
static void      ss_packet_generate (void);
void HashString(BYTE[100], int);
void UHash24 (uchar *msg, uchar *secret, int len, uchar *result);
unsigned char * IntToByteArray(int value);


unsigned char convertedMessage[11];

int packetSentCount=0;
//bool iskeyInterval=false;

int macSentMessageCounter=0;
int messageSentPacketCounter=0;
int mainMessage=0;
int hashedKey=110;
int hashedKeyPacket=110;
char key[4];
char keyHashed[4];
bool mac=true;
bool newMacInterval=false;
bool newMessageInterval = false;

/*typedef struct  {
unsigned char key[20];
unsigned char name[11];
unsigned char message[11];
int isPacketInserted;//1 = buffer the packet  0 = do not buffer the packet
}
```

```
Custom_DS;
*/

unsigned int RSHash1  (char* str, unsigned int len);
int packetNumber;
int rpt;
int counter=0;
int pktCounter=0;
int interval=0;
int keyInterval=1;

unsigned int db;



        BYTE digest[4];

int idx=0;
/* allocate an empty list */
List *lst;

/* ************************************************************************
 *                                              *
 *        General Purpose Hash Function Algorithms Library            *
 *                                              *
 * Author: Arash Partow - 2002                          *
 * URL: http://www.partow.net                            *
 * URL: http://www.partow.net/programming/hashfunctions/index.html       *
 *                                              *
 * Copyright notice:                                *
 * Free use of the General Purpose Hash Function Algorithms Library is    *
 * permitted under the guidelines and in accordance with the most current *
 * version of the Common Public License.                    *
 * http://www.opensource.org/licenses/cpl1.0.php                 *
 *                                              *
 ************************************************************************/

#ifndef INCLUDE_GENERALHASHFUNCTION_CPP_H
#define INCLUDE_GENERALHASHFUNCTION_CPP_H

typedef unsigned int (*HashFunction)(const std::string&);


 int RSHash  (const std::string& str);//unsigned

#endif
```

```
/* End of Header Block */

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



/* State variable definitions */
class simple_source_state
        {
        private:
                /* Internal state tracking for FSM */
                FSM_SYS_STATE

        public:
                simple_source_state (void);

                /* Destructor contains Termination Block */
                ~simple_source_state (void);

                /* State Variables */
                Objid                   own_id                          ;   /* Object ID
of the surrounding module. */
                char                    format_str [64]                  ; /* Format of
the packets generated by this source. */
                double                  start_time                      ;   /* Time when
this source will start its packet generation */
                                                                            /* activities.
*/
                double                  stop_time                       ;   /* Time when
this source will stop its packet generation */
                                                                            /* activities.
*/
                OmsT_Dist_Handle                interarrival_dist_ptr                ;
        /* PDF used to determine the interarrival times of */
                                                                            /* generated
packets.                */
```

```
		OmsT_Dist_Handle		pksize_dist_ptr			; /* PDF
used to determine the sizes of generated packets. */
		Boolean				generate_unformatted			;
	/* Flag that indicates whether the source will generate */
											/* unformatted
or formatted packets.		*/
		Evhandle			next_pk_evh				; /*
Event handle for the arrival of next packet. */
		double			next_intarr_time			;/* Time
between the generation of the last packet and the */
											/* next packet.
*/
		Stathandle			bits_sent_hndl				; /*
Statistic handle for "Traffic Sent (bits/sec)" statistic. */
		Stathandle			packets_sent_hndl				;
	/* Statistic handle for "Traffic Sent (packets/sec)" statistic. */
		Stathandle			packet_size_hndl				;
	/* Statistic handle for "Packet Size (bits)" statistic. */
		Stathandle			interarrivals_hndl				;/*
Statistic handle for "Packet Interaarival Time (secs)" */
											/* statistic.
*/
		int				max_packet_count			;	/*
Number of Packets to process */

		/* FSM code */
		void simple_source (OP_SIM_CONTEXT_ARG_OPT);
		/* Diagnostic Block */
		void _op_simple_source_diag (OP_SIM_CONTEXT_ARG_OPT);

#if defined (VOSD_NEW_BAD_ALLOC)
		void * operator new (size_t) throw (VOSD_BAD_ALLOC);
#else
		void * operator new (size_t);
#endif
		void operator delete (void *);

		/* Memory management */
		static VosT_Obtype obtype;
	};

VosT_Obtype simple_source_state::obtype = (VosT_Obtype)OPC_NIL;

#define own_id				op_sv_ptr->own_id
#define format_str			op_sv_ptr->format_str
#define start_time			op_sv_ptr->start_time
```

```
#define stop_time                op_sv_ptr->stop_time
#define interarrival_dist_ptr    op_sv_ptr->interarrival_dist_ptr
#define pksize_dist_ptr          op_sv_ptr->pksize_dist_ptr
#define generate_unformatted             op_sv_ptr->generate_unformatted
#define next_pk_evh              op_sv_ptr->next_pk_evh
#define next_intarr_time         op_sv_ptr->next_intarr_time
#define bits_sent_hndl           op_sv_ptr->bits_sent_hndl
#define packets_sent_hndl                op_sv_ptr->packets_sent_hndl
#define packet_size_hndl                 op_sv_ptr->packet_size_hndl
#define interarrivals_hndl       op_sv_ptr->interarrivals_hndl
#define max_packet_count                 op_sv_ptr->max_packet_count

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


/* Function Block */

#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ + 2};
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>

static void ss_packet_generate(void)
     {

     FIN (ss_packet_generate ());
     Packet*                          pkptr;
     Packet*                     pktKeyPtr;
     int idx=-1;
          /** This function creates a packet based on the packet generation          **/
          /** specifications of the source model and sends it to the lower layer.    **/
     if (mac)
          {
                macSentMessageCounter++;
                if(macSentMessageCounter % 2 == 0)
                        newMacInterval = true;

                if (macSentMessageCounter % 4 == 0)
                        mac = false;

                        counter++;
                        char nextMessage[4];
                        char strMessage[4];
                        char key[4];

                        mainMessage = counter + 12345;

                        itoa(mainMessage, strMessage, 10);
                        itoa(mainMessage + 4, nextMessage, 10);//4th message after the
current message
                        int h = RSHash(nextMessage);

                        int macMessage = h + mainMessage;

                        char messageHashed[4];

                        itoa(macMessage,messageHashed,10);

                        CHMAC_SHA1 HMAC_SHA1 ;
                        HMAC_SHA1.HMAC_SHA1((unsigned char *)messageHashed ,
sizeof(messageHashed), (unsigned char *)key, sizeof(key), digest) ;
                        unsigned long hashedCode = 0;
                        unsigned long mult = 1;
                        for (unsigned i = 0; i < 4; ++i) {
                                hashedCode += mult * digest[sizeof(digest)-1-i];
                                mult <<= 8;
                        }
                        pkptr = op_pk_create (0);
                        op_pk_fd_set (pkptr, 0, OPC_FIELD_TYPE_INTEGER,
hashedCode, 32);
                        op_pk_send (pkptr, SSC_STRM_TO_LOW);

                        if(newMacInterval)
                                {
                                hashedKey = RSHash(itoa(hashedKey, key, 10));
                                newMacInterval = false;
                                }
                        printf("Mac \n");
```

```
            }
        else
            {
            messageSentPacketCounter++;

            if(messageSentPacketCounter % 2 == 0)
                newMessageInterval = true;

            if (messageSentPacketCounter % 4 == 0)
                mac = true;

            char nextMessage[4];
            pktKeyPtr = op_pk_create (0);

            int message = counter - 4 + 12345;

            itoa(message + 4, nextMessage, 10);//4th message after the current message

            int hashedNextMessage = RSHash(nextMessage);


            op_pk_fd_set (pktKeyPtr, 0, OPC_FIELD_TYPE_INTEGER, message, 32);
            op_pk_fd_set (pktKeyPtr, 1, OPC_FIELD_TYPE_INTEGER, hashedKeyPacket,
32);
            op_pk_fd_set (pktKeyPtr, 2, OPC_FIELD_TYPE_INTEGER, message + 4, 32);

            op_pk_send (pktKeyPtr, SSC_STRM_TO_LOW);

                        if(newMessageInterval)
                            {
                            hashedKeyPacket = RSHash(itoa(hashedKeyPacket,
keyHashed, 10));
                            newMessageInterval = false;
                            }

                                printf("Message \n");
            }
            FOUT;
        }


 int RSHash(const std::string& str)//unsigned
{
  int b    = 3785;//51;
  int a    = 63689;
  int hash = 0;
  for(int i = 0; i < str.length(); i++)//std::size_t
  {
    hash = hash * a + str[i];
    a    = a * b;

  }
  return (hash & 0x7FFFFFFF); //Returns the hashed string
}
/* End Of RS Hash Function */

void CHMAC_SHA1::HMAC_SHA1(BYTE *text, int text_len, BYTE *key, int key_len, BYTE
*digest)
{

        memset(SHA1_Key, 0, SHA1_BLOCK_SIZE);

        /* repeated 64 times for values in ipad and opad */
        memset(m_ipad, 0x36, sizeof(m_ipad));
        memset(m_opad, 0x5c, sizeof(m_opad));
        /* STEP 1 */
        if (key_len > SHA1_BLOCK_SIZE)
        {
                CSHA1::Reset();
                CSHA1::Update((UINT_8 *)key, key_len);
                CSHA1::Final();
                CSHA1::GetHash((UINT_8 *)SHA1_Key);
        }
        else
                memcpy(SHA1_Key, key, key_len);

        /* STEP 2 */
        for (int i=0; i<sizeof(m_ipad); i++)
        {
                m_ipad[i] ^= SHA1_Key[i];
        }

        /* STEP 3 */
        memcpy(AppendBuf1, m_ipad, sizeof(m_ipad));
        memcpy(AppendBuf1 + sizeof(m_ipad), text, text_len);

        /* STEP 4 */
        CSHA1::Reset();
        CSHA1::Update((UINT_8 *)AppendBuf1, sizeof(m_ipad) + text_len);
        CSHA1::Final();
        CSHA1::GetHash((UINT_8 *)szReport);
        /* STEP 5 */
        for (int j=0; j<sizeof(m_opad); j++)
        {
                m_opad[j] ^= SHA1_Key[j];
        }

        /* STEP 6 */
        memcpy(AppendBuf2, m_opad, sizeof(m_opad));
        memcpy(AppendBuf2 + sizeof(m_opad), szReport, SHA1_DIGEST_LENGTH);

        /*STEP 7 */
        CSHA1::Reset();
        CSHA1::Update((UINT_8 *)AppendBuf2, sizeof(m_opad) +
SHA1_DIGEST_LENGTH);
        CSHA1::Final();

        CSHA1::GetHash((UINT_8 *)digest);
}

#ifdef SHA1_UTILITY_FUNCTIONS
#define SHA1_MAX_FILE_BUFFER 8000
#endif

// Rotate x bits to the left
#ifndef ROL32
#ifdef _MSC_VER
#define ROL32(_val32, _nBits) _rotl(_val32, _nBits)
#else
#define ROL32(_val32, _nBits) (((_val32)<<(_nBits))|((_val32)>>(32-(_nBits))))
#endif
#endif

#ifdef SHA1_LITTLE_ENDIAN
#define SHABLK0(i) (m_block->l[i] = \
        (ROL32(m_block->l[i],24) & 0xFF00FF00) | (ROL32(m_block->l[i],8) & 0x00FF00FF))
#else
#define SHABLK0(i) (m_block->l[i])
#endif

#define SHABLK(i) (m_block->l[i&15] = ROL32(m_block->l[(i+13)&15] ^ m_block-
>l[(i+8)&15] \
        ^ m_block->l[(i+2)&15] ^ m_block->l[i&15],1))

// SHA-1 rounds
```

```
#define _R0(v,w,x,y,z,i) { z+=((w&(x^y))^y)+SHABLK0(i)+0x5A827999+ROL32(v,5);
w=ROL32(w,30); }
#define _R1(v,w,x,y,z,i) { z+=((w&(x^y))^y)+SHABLK(i)+0x5A827999+ROL32(v,5);
w=ROL32(w,30); }
#define _R2(v,w,x,y,z,i) { z+=(w^x^y)+SHABLK(i)+0x6ED9EBA1+ROL32(v,5);
w=ROL32(w,30); }
#define _R3(v,w,x,y,z,i) { z+=(((w|x)&y)|(w&x))+SHABLK(i)+0x8F1BBCDC+ROL32(v,5);
w=ROL32(w,30); }
#define _R4(v,w,x,y,z,i) { z+=(w^x^y)+SHABLK(i)+0xCA62C1D6+ROL32(v,5);
w=ROL32(w,30); }

CSHA1::CSHA1()
{
        m_block = (SHA1_WORKSPACE_BLOCK *)m_workspace;

        Reset();
}

CSHA1::~CSHA1()
{
        Reset();
}

void CSHA1::Reset()
{
        // SHA1 initialization constants
        m_state[0] = 0x67452301;
        m_state[1] = 0xEFCDAB89;
        m_state[2] = 0x98BADCFE;
        m_state[3] = 0x10325476;
        m_state[4] = 0xC3D2E1F0;

        m_count[0] = 0;
        m_count[1] = 0;
}

void CSHA1::Transform(UINT_32 *state, UINT_8 *buffer)
{
        // Copy state[] to working vars
        UINT_32 a = state[0], b = state[1], c = state[2], d = state[3], e = state[4];

        memcpy(m_block, buffer, 64);

        // 4 rounds of 20 operations each. Loop unrolled.
        _R0(a,b,c,d,e, 0); _R0(e,a,b,c,d, 1); _R0(d,e,a,b,c, 2); _R0(c,d,e,a,b, 3);
        _R0(b,c,d,e,a, 4); _R0(a,b,c,d,e, 5); _R0(e,a,b,c,d, 6); _R0(d,e,a,b,c, 7);
```

```
	_R0(c,d,e,a,b, 8); _R0(b,c,d,e,a, 9); _R0(a,b,c,d,e,10); _R0(e,a,b,c,d,11);
	_R0(d,e,a,b,c,12); _R0(c,d,e,a,b,13); _R0(b,c,d,e,a,14); _R0(a,b,c,d,e,15);
	_R1(e,a,b,c,d,16); _R1(d,e,a,b,c,17); _R1(c,d,e,a,b,18); _R1(b,c,d,e,a,19);
	_R2(a,b,c,d,e,20); _R2(e,a,b,c,d,21); _R2(d,e,a,b,c,22); _R2(c,d,e,a,b,23);
	_R2(b,c,d,e,a,24); _R2(a,b,c,d,e,25); _R2(e,a,b,c,d,26); _R2(d,e,a,b,c,27);
	_R2(c,d,e,a,b,28); _R2(b,c,d,e,a,29); _R2(a,b,c,d,e,30); _R2(e,a,b,c,d,31);
	_R2(d,e,a,b,c,32); _R2(c,d,e,a,b,33); _R2(b,c,d,e,a,34); _R2(a,b,c,d,e,35);
	_R2(e,a,b,c,d,36); _R2(d,e,a,b,c,37); _R2(c,d,e,a,b,38); _R2(b,c,d,e,a,39);
	_R3(a,b,c,d,e,40); _R3(e,a,b,c,d,41); _R3(d,e,a,b,c,42); _R3(c,d,e,a,b,43);
	_R3(b,c,d,e,a,44); _R3(a,b,c,d,e,45); _R3(e,a,b,c,d,46); _R3(d,e,a,b,c,47);
	_R3(c,d,e,a,b,48); _R3(b,c,d,e,a,49); _R3(a,b,c,d,e,50); _R3(e,a,b,c,d,51);
	_R3(d,e,a,b,c,52); _R3(c,d,e,a,b,53); _R3(b,c,d,e,a,54); _R3(a,b,c,d,e,55);
	_R3(e,a,b,c,d,56); _R3(d,e,a,b,c,57); _R3(c,d,e,a,b,58); _R3(b,c,d,e,a,59);
	_R4(a,b,c,d,e,60); _R4(e,a,b,c,d,61); _R4(d,e,a,b,c,62); _R4(c,d,e,a,b,63);
	_R4(b,c,d,e,a,64); _R4(a,b,c,d,e,65); _R4(e,a,b,c,d,66); _R4(d,e,a,b,c,67);
	_R4(c,d,e,a,b,68); _R4(b,c,d,e,a,69); _R4(a,b,c,d,e,70); _R4(e,a,b,c,d,71);
	_R4(d,e,a,b,c,72); _R4(c,d,e,a,b,73); _R4(b,c,d,e,a,74); _R4(a,b,c,d,e,75);
	_R4(e,a,b,c,d,76); _R4(d,e,a,b,c,77); _R4(c,d,e,a,b,78); _R4(b,c,d,e,a,79);

	// Add the working vars back into state
	state[0] += a;
	state[1] += b;
	state[2] += c;
	state[3] += d;
	state[4] += e;

	// Wipe variables
#ifdef SHA1_WIPE_VARIABLES
	a = b = c = d = e = 0;
#endif
}

// Use this function to hash in binary data and strings
void CSHA1::Update(UINT_8 *data, UINT_32 len)
{
	UINT_32 i, j;

	j = (m_count[0] >> 3) & 63;

	if((m_count[0] += len << 3) < (len << 3)) m_count[1]++;

	m_count[1] += (len >> 29);

	if((j + len) > 63)
	{
		i = 64 - j;
```

```
                memcpy(&m_buffer[j], data, i);
                Transform(m_state, m_buffer);

                for(; i + 63 < len; i += 64) Transform(m_state, &data[i]);

                j = 0;
        }
        else i = 0;

        memcpy(&m_buffer[j], &data[i], len - i);
}

#ifdef SHA1_UTILITY_FUNCTIONS
// Hash in file contents
bool CSHA1::HashFile(char *szFileName)
{
        unsigned long ulFileSize, ulRest, ulBlocks;
        unsigned long i;
        UINT_8 uData[SHA1_MAX_FILE_BUFFER];
        FILE *fIn;

        if(szFileName == NULL) return false;

        fIn = fopen(szFileName, "rb");
        if(fIn == NULL) return false;

        fseek(fIn, 0, SEEK_END);
        ulFileSize = (unsigned long)ftell(fIn);
        fseek(fIn, 0, SEEK_SET);

        if(ulFileSize != 0)
        {
                ulBlocks = ulFileSize / SHA1_MAX_FILE_BUFFER;
                ulRest = ulFileSize % SHA1_MAX_FILE_BUFFER;
        }
        else
        {
                ulBlocks = 0;
                ulRest = 0;
        }

        for(i = 0; i < ulBlocks; i++)
        {
                fread(uData, 1, SHA1_MAX_FILE_BUFFER, fIn);
                Update((UINT_8 *)uData, SHA1_MAX_FILE_BUFFER);
        }
        if(ulRest != 0)
        {
                fread(uData, 1, ulRest, fIn);
                Update((UINT_8 *)uData, ulRest);
        }

        fclose(fIn); fIn = NULL;
        return true;
}
#endif

void CSHA1::Final()
{
        UINT_32 i;
        UINT_8 finalcount[8];

        for(i = 0; i < 8; i++)
                finalcount[i] = (UINT_8)((m_count[((i >= 4) ? 0 : 1)]
                        >> ((3 - (i & 3)) * 8) ) & 255); // Endian independent

        Update((UINT_8 *)"\200", 1);

        while ((m_count[0] & 504) != 448)
                Update((UINT_8 *)"\0", 1);

        Update(finalcount, 8); // Cause a SHA1Transform()

        for(i = 0; i < 20; i++)
        {
                m_digest[i] = (UINT_8)((m_state[i >> 2] >> ((3 - (i & 3)) * 8) ) & 255);
        }


        // Wipe variables for security reasons
#ifdef SHA1_WIPE_VARIABLES
        i = 0;
        memset(m_buffer, 0, 64);
        memset(m_state, 0, 20);
        memset(m_count, 0, 8);
        memset(finalcount, 0, 8);
        Transform(m_state, m_buffer);
#endif
}

#ifdef SHA1_UTILITY_FUNCTIONS
```

```
// Get the final hash as a pre-formatted string
void CSHA1::ReportHash(char *szReport, unsigned char uReportType)
{
        unsigned char i;
        char szTemp[16];

        if(szReport == NULL) return;

        if(uReportType == REPORT_HEX)
        {
                sprintf(szTemp, "%02X", m_digest[0]);
                strcat(szReport, szTemp);

                for(i = 1; i < 20; i++)
                {
                        sprintf(szTemp, " %02X", m_digest[i]);
                        strcat(szReport, szTemp);
                }
        }
        else if(uReportType == REPORT_DIGIT)
        {
                sprintf(szTemp, "%u", m_digest[0]);
                strcat(szReport, szTemp);

                for(i = 1; i < 20; i++)
                {
                        sprintf(szTemp, " %u", m_digest[i]);
                        strcat(szReport, szTemp);
                }
        }
        else strcpy(szReport, "Error: Unknown report type!");
}
#endif


// Get the raw message digest
void CSHA1::GetHash(UINT_8 *puDest)
{
        memcpy(puDest, m_digest, 20);
}
```

```
void UHash24 (uchar *msg, uchar *secret, int len, uchar *result)
{
 uchar r1 = 0, r2 = 0, r3 = 0, s1, s2, s3, byteCnt = 0, bitCnt, byte;

 while (len-- > 0) {
  if (byteCnt-- == 0) {
   s1 = *secret++;
   s2 = *secret++;
   s3 = *secret++;
   byteCnt = 2;
  }
  byte = *msg++;
  for (bitCnt = 0; bitCnt < 8; bitCnt++) {
   if (byte & 1) { /* msg not divisible by x */
    r1 ^= s1; /* so add s * 1 */
    r2 ^= s2;
    r3 ^= s3;
   }
   byte >>= 1; /* divide message by x */
   if (s3 & 0x80) { /* and multiply secret with x, subtracting
      the polynomial when necessary to keep its order under 24 */
    s3 <<= 1;
    if (s2 & 0x80) s3 |= 1;
    s2 <<= 1;
    if (s1 & 0x80) s2 |= 1;
    s1 <<= 1;

    s1 ^= 0x1B; /* x^24 + x^4 + x^3 + x + 1 */
   }
   else {
    s3 <<= 1;
    if (s2 & 0x80) s3 |= 1;
    s2 <<= 1;
    if (s1 & 0x80) s2 |= 1;
    s1 <<= 1;
   }
  } /* for each bit in the message */
 } /* for each byte in the message */
 *result++ ^= r1;
 *result++ ^= r2;
 *result++ ^= r3;
 printf("result %s= \n", result);
}


/* End of Function Block */
/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.              */
#undef FIN_TRACING
#define FIN_TRACING

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

/* Undefine shortcuts to state variables because the */
/* following functions are part of the state class */
#undef own_id
#undef format_str
#undef start_time
#undef stop_time
#undef interarrival_dist_ptr
#undef pksize_dist_ptr
#undef generate_unformatted
#undef next_pk_evh
#undef next_intarr_time
#undef bits_sent_hndl
#undef packets_sent_hndl
#undef packet_size_hndl
#undef interarrivals_hndl
#undef max_packet_count

/* Access from C kernel using C linkage */
extern "C"
{
        VosT_Obtype _op_simple_source_init (int * init_block_ptr);
        VosT_Address _op_simple_source_alloc (VosT_Obtype, int);
        void simple_source (OP_SIM_CONTEXT_ARG_OPT)
                {
                ((simple_source_state *)(OP_SIM_CONTEXT_PTR->_op_mod_state_ptr))-
>simple_source (OP_SIM_CONTEXT_PTR_OPT);
                }

        void _op_simple_source_svar (void *, const char *, void **);

        void _op_simple_source_diag (OP_SIM_CONTEXT_ARG_OPT)
                {
                ((simple_source_state *)(OP_SIM_CONTEXT_PTR->_op_mod_state_ptr))-
>_op_simple_source_diag (OP_SIM_CONTEXT_PTR_OPT);
                }
        void _op_simple_source_terminate (OP_SIM_CONTEXT_ARG_OPT)
                {
                /* The destructor is the Termination Block */
                delete (simple_source_state *)(OP_SIM_CONTEXT_PTR->_op_mod_state_ptr);
                }


} /* end of 'extern "C"' */


/* Process model interrupt handling procedure */


void
simple_source_state::simple_source (OP_SIM_CONTEXT_ARG_OPT)
        {
#if !defined (VOSD_NO_FIN)
        int _op_block_origin = 0;
#endif
        FIN_MT (simple_source_state::simple_source ());
        try
                {
                /* Temporary Variables */
                /* Variables used in the "init" state.          */
                char            interarrival_str [128];
                char            size_str [128];
                Prg_List*       pk_format_names_lptr;
                char*           found_format_str;
                Boolean                 format_found;
                int                     i;

                /* Variables used in state transitions.         */
                int                     intrpt_code;
                /* End of Temporary Variables */


                FSM_ENTER ("simple_source")

                FSM_BLOCK_SWITCH
                        {
                        /*---------------------------------------------------------*/
                        /** state (init) enter executives **/
                        FSM_STATE_ENTER_UNFORCED_NOLABEL (0, "init",
"simple_source [init enter execs]")
                                FSM_PROFILE_SECTION_IN ("simple_source [init enter
execs]", state0_enter_exec)
```

```
                                        {
                                        /* At this initial state, we read the values of source attributes
        */
                                        /* and schedule a selt interrupt that will indicate our start time
        */
                                        /* for packet generation.
                                                */

                                        /* Obtain the object id of the surrounding module.
                */
                                        own_id = op_id_self ();

                                        //Initialize hashedKey
                                        hashedKey = RSHash(itoa(hashedKey, key, 10));
                                        hashedKeyPacket = RSHash(itoa(hashedKeyPacket, keyHashed,
10));
                                    . . /* Read the values of the packet generation parameters, i.e. the
        */
                                        /* attribute values of the surrounding module.
                        */
                                        op_ima_obj_attr_get (own_id, "Packet Interarrival Time",
interarrival_str);
                                        op_ima_obj_attr_get (own_id, "Packet Size",         size_str);
                                        op_ima_obj_attr_get (own_id, "Packet Format",
format_str);
                                        op_ima_obj_attr_get (own_id, "Start Time",          &start_time);
                                        op_ima_obj_attr_get (own_id, "Stop Time",
&stop_time);

                                        /* Get the maximum packet count, */
                                        /* set at simulation run-time */
                                        op_ima_sim_attr_get_int32 ("Maximum Packet",
                                                    &max_packet_count);

                                        /* Load the PDFs that will be used in computing the packet
        */
                                        /* interarrival times and packet sizes.
                        */
                                        interarrival_dist_ptr = oms_dist_load_from_string
(interarrival_str);
                                        pksize_dist_ptr     = oms_dist_load_from_string (size_str);

                                        /* Verify the existence of the packet format to be used for
        */
                                        /* generated packets.
                                                */
```

```
                              if (strcmp (format_str, "NONE") == 0)
                                    {
                                    /* We will generate unformatted packets. Set the flag.
            */
                                    generate_unformatted = OPC_TRUE;
                                    }
                              else
                                    {
                                    /* We will generate formatted packets. Turn off the flag.
      */
                                    generate_unformatted = OPC_FALSE;

                                    /* Get the list of all available packet formats.
            */
                                    pk_format_names_lptr = prg_tfile_name_list_get
(PrgC_Tfile_Type_Packet_Format);

                                    /* Search the list for the requested packet format.
            */
                                    format_found = OPC_FALSE;
                                    for (i = prg_list_size (pk_format_names_lptr);
((format_found == OPC_FALSE) && (i > 0)); i--)
                                          {
                                          /* Access the next format name and compare with
requested      */
                                          /* format name.
                                                */
                                          found_format_str = (char *) prg_list_access
(pk_format_names_lptr, i - 1);
                                          if (strcmp (found_format_str, format_str) == 0)
                                                format_found = OPC_TRUE;
                                          }

                                if (format_found == OPC_FALSE)
                                          {
                                          /* The requested format does not exist. Generate
            */
                                          /* unformatted packets.
                                          */
                                          generate_unformatted = OPC_TRUE;

                                          /* Display an appropriate warning.
                              */
                                          op_prg_odb_print_major ("Warning from simple
packet generator model (simple_source):",
                                                                                    "The
specified packet format", format_str,
                                                                                    "is not
found. Generating unformatted packets instead.", OPC_NIL);
                                              }

                                    /* Destroy the lits and its elements since we don't need it
      */
                                    /* anymore.
                                                      */
                                    prg_list_free (pk_format_names_lptr);
                                    prg_mem_free  (pk_format_names_lptr);
                                    }


                              /* Make sure we have valid start and stop times, i.e. stop time is
      */
                              /* not earlier than start time.
                              */
                              if ((stop_time <= start_time) && (stop_time !=
SSC_INFINITE_TIME))
                                    {
                                    /* Stop time is earlier than start time. Disable the source.
      */
                                    start_time = SSC_INFINITE_TIME;

                                    /* Display an appropriate warning.
                              */
                                    op_prg_odb_print_major ("Warning from simple packet
generator model (simple_source):",
                                                                              "Although the
generator is not disabled (start time is set to a finite value),",
                                                                              "a stop time
that is not later than the start time is specified.",
                                                                              "Disabling the
generator.", OPC_NIL);
                                    }

                              /* Schedule a self interrupt that will indicate our start time for
      */
                              /* packet generation activities. If the source is disabled,
      */
                              /* schedule it at current time with the appropriate code value.
      */
                              if (start_time == SSC_INFINITE_TIME)
                                    {
                                        op_intrpt_schedule_self (op_sim_time (), SSC_STOP);
                                        }
                                else
                                        {
                                        op_intrpt_schedule_self (start_time, SSC_START);

                                        /* In this case, also schedule the interrupt when we will
stop    */
                                        /* generating packets, unless we are configured to run until
        */
                                        /* the end of the simulation.
                                */
                                        if (stop_time != SSC_INFINITE_TIME)
                                                {
                                                op_intrpt_schedule_self (stop_time, SSC_STOP);
                                                }
                                        }

                                rpt=0;
                                /* Register the statistics that will be maintained by this model.
        */
                                bits_sent_hndl          = op_stat_reg ("Generator.Traffic Sent
(bits/sec)",                    OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                                packets_sent_hndl  = op_stat_reg ("Generator.Traffic Sent
(packets/sec)",         OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                                packet_size_hndl    = op_stat_reg ("Generator.Packet Size (bits)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                                interarrivals_hndl  = op_stat_reg ("Generator.Packet Interarrival
Time (secs)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

                                }
                                FSM_PROFILE_SECTION_OUT (state0_enter_exec)

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


                        /** state (init) exit executives **/
                        FSM_STATE_EXIT_UNFORCED (0, "init", "simple_source [init exit
execs]")
                                FSM_PROFILE_SECTION_IN ("simple_source [init exit execs]",
state0_exit_exec)
                                {
                                /* Determine the code of the interrupt, which is used in evaluating
        */
```

```
                              /* state transition conditions.
                              */
                              intrpt_code = op_intrpt_code ();
                              }
                              FSM_PROFILE_SECTION_OUT (state0_exit_exec)


                    /** state (init) transition processing **/
                    FSM_PROFILE_SECTION_IN ("simple_source [init trans conditions]",
state0_trans_conds)
                    FSM_INIT_COND (START)
                    FSM_TEST_COND (DISABLED)
                    FSM_TEST_LOGIC ("init")
                    FSM_PROFILE_SECTION_OUT (state0_trans_conds)

                    FSM_TRANSIT_SWITCH
                              {
                              FSM_CASE_TRANSIT (0, 1, state1_enter_exec,
ss_packet_generate();, "START", "ss_packet_generate()", "init", "generate", "tr_0",
"simple_source [init -> generate : START / ss_packet_generate()]")
                              FSM_CASE_TRANSIT (1, 2, state2_enter_exec, ;, "DISABLED",
"", "init", "stop", "tr_1", "simple_source [init -> stop : DISABLED / ]")
                              }
                              /*---------------------------------------------------------*/



                    /** state (generate) enter executives **/
                    FSM_STATE_ENTER_UNFORCED (1, "generate", state1_enter_exec,
"simple_source [generate enter execs]")
                              FSM_PROFILE_SECTION_IN ("simple_source [generate enter
execs]", state1_enter_exec)
                              {
                              /* At the enter execs of the "generate" state we schedule the
     */
                              /* arrival of the next packet.
                              */
                              next_intarr_time = oms_dist_outcome (interarrival_dist_ptr);
                              rpt++;
                              /* Make sure that interarrival time is not negative. In that case it */
                              /* will be set to 0.
                                        */
                              if (next_intarr_time <0)
                                        {
                                        next_intarr_time = 0;
                                        }
                                        next_pk_evh      = op_intrpt_schedule_self (op_sim_time ()
+ next_intarr_time, SSC_GENERATE);

                                op_stat_write (interarrivals_hndl, next_intarr_time);
                                }
                                FSM_PROFILE_SECTION_OUT (state1_enter_exec)

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


                        /** state (generate) exit executives **/
                        FSM_STATE_EXIT_UNFORCED (1, "generate", "simple_source
[generate exit execs]")
                                FSM_PROFILE_SECTION_IN ("simple_source [generate exit
execs]", state1_exit_exec)
                                {
                                /* Determine the code of the interrupt, which is used in evaluating
       */

                                /* state transition conditions.
                                */
                                intrpt_code = op_intrpt_code ();


                                }
                                FSM_PROFILE_SECTION_OUT (state1_exit_exec)


                        /** state (generate) transition processing **/
                        FSM_PROFILE_SECTION_IN ("simple_source [generate trans
conditions]", state1_trans_conds)
                        FSM_INIT_COND (STOP)
                        FSM_TEST_COND (PACKET_GENERATE)
                        FSM_TEST_LOGIC ("generate")
                        FSM_PROFILE_SECTION_OUT (state1_trans_conds)

                        FSM_TRANSIT_SWITCH
                                {
                                FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;, "STOP", "",
"generate", "stop", "tr_2", "simple_source [generate -> stop : STOP / ]")
                                FSM_CASE_TRANSIT (1, 1, state1_enter_exec,
ss_packet_generate();, "PACKET_GENERATE", "ss_packet_generate()", "generate",
"generate", "tr_3", "simple_source [generate -> generate : PACKET_GENERATE /
ss_packet_generate()]")
                                }
```

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


                /** state (stop) enter executives **/
                FSM_STATE_ENTER_UNFORCED (2, "stop", state2_enter_exec,
"simple_source [stop enter execs]")
                    FSM_PROFILE_SECTION_IN ("simple_source [stop enter
execs]", state2_enter_exec)
                    {
                    /* When we enter into the "stop" state, it is the time for us to
    */
                    /* stop generating traffic. We simply cancel the generation of the
    */
                    /* next packet and go into a silent mode by not scheduling
anything     */
                    /* else.
                    */
                    if (op_ev_valid (next_pk_evh) == OPC_TRUE)
                        {
                        op_ev_cancel (next_pk_evh);
                        }

                    }
                    FSM_PROFILE_SECTION_OUT (state2_enter_exec)

                /** blocking after enter executives of unforced state. **/
                FSM_EXIT (5,"simple_source")


                /** state (stop) exit executives **/
                FSM_STATE_EXIT_UNFORCED (2, "stop", "simple_source [stop exit
execs]")


                /** state (stop) transition processing **/
                FSM_TRANSIT_MISSING ("stop")
                    /*------------------------------------------------------*/



                }


            FSM_EXIT (0,"simple_source")
            }
```

```
        catch (...)
                {
                Vos_Error_Print (VOSC_ERROR_ABORT,
                        (const char *)VOSC_NIL,
                        "Unhandled C++ exception in process model (simple_source)",
                        (const char *)VOSC_NIL, (const char *)VOSC_NIL);
                }
        }




void
simple_source_state::_op_simple_source_diag (OP_SIM_CONTEXT_ARG_OPT)
        {
        /* No Diagnostic Block */
        }

void
simple_source_state::operator delete (void* ptr)
        {
        FIN (simple_source_state::operator delete (ptr));
        Vos_Poolmem_Dealloc (ptr);
        FOUT
        }

simple_source_state::~simple_source_state (void)
        {

        FIN (simple_source_state::~simple_source_state ())


        /* No Termination Block */


        FOUT
        }


#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

void *
```

```
simple_source_state::operator new (size_t)
#if defined (VOSD_NEW_BAD_ALLOC)
                throw (VOSD_BAD_ALLOC)
#endif
        {
        void * new_ptr;

        FIN_MT (simple_source_state::operator new ());

        new_ptr = Vos_Alloc_Object (simple_source_state::obtype);
#if defined (VOSD_NEW_BAD_ALLOC)
        if (new_ptr == VOSC_NIL) throw VOSD_BAD_ALLOC();
#endif
        FRET (new_ptr)
        }

/* State constructor initializes FSM handling */
/* by setting the initial state to the first */
/* block of code to enter. */

simple_source_state::simple_source_state (void) :
                _op_current_block (0)
        {
#if defined (OPD_ALLOW_ODB)
                _op_current_state = "simple_source [init enter execs]";
#endif
        }

VosT_Obtype
_op_simple_source_init (int * init_block_ptr)
        {
        FIN_MT (_op_simple_source_init (init_block_ptr))

        simple_source_state::obtype = Vos_Define_Object_Prstate ("proc state vars
(simple_source)",
                sizeof (simple_source_state));
        *init_block_ptr = 0;

        FRET (simple_source_state::obtype)
        }

VosT_Address
_op_simple_source_alloc (VosT_Obtype, int)
        {
#if !defined (VOSD_NO_FIN)
        int _op_block_origin = 0;
```

```
#endif
        simple_source_state * ptr;
        FIN_MT (_op_simple_source_alloc ())

        /* New instance will have FSM handling initialized */
#if defined (VOSD_NEW_BAD_ALLOC)
        try {
                ptr = new simple_source_state;
        } catch (const VOSD_BAD_ALLOC &) {
                ptr = VOSC_NIL;
        }
#else
        ptr = new simple_source_state;
#endif
        FRET ((VosT_Address)ptr)
        }



void
_op_simple_source_svar (void * gen_ptr, const char * var_name, void ** var_p_ptr)
        {
        simple_source_state             *prs_ptr;

        FIN_MT (_op_simple_source_svar (gen_ptr, var_name, var_p_ptr))

        if (var_name == OPC_NIL)
                {
                *var_p_ptr = (void *)OPC_NIL;
                FOUT
                }
        prs_ptr = (simple_source_state *)gen_ptr;

        if (strcmp ("own_id" , var_name) == 0)
                {
                *var_p_ptr = (void *) (&prs_ptr->own_id);
                FOUT
                }
        if (strcmp ("format_str" , var_name) == 0)
                {
                *var_p_ptr = (void *) (prs_ptr->format_str);
                FOUT
                }
        if (strcmp ("start_time" , var_name) == 0)
                {
                *var_p_ptr = (void *) (&prs_ptr->start_time);
```

```
		FOUT
		}
	if (strcmp ("stop_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->stop_time);
		FOUT
		}
	if (strcmp ("interarrival_dist_ptr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->interarrival_dist_ptr);
		FOUT
		}
	if (strcmp ("pksize_dist_ptr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->pksize_dist_ptr);
		FOUT
		}
	if (strcmp ("generate_unformatted" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->generate_unformatted);
		FOUT
		}
	if (strcmp ("next_pk_evh" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->next_pk_evh);
		FOUT
		}
	if (strcmp ("next_intarr_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->next_intarr_time);
		FOUT
		}
	if (strcmp ("bits_sent_hndl" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->bits_sent_hndl);
		FOUT
		}
	if (strcmp ("packets_sent_hndl" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->packets_sent_hndl);
		FOUT
		}
	if (strcmp ("packet_size_hndl" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->packet_size_hndl);
		FOUT
```

```
        }
if (strcmp ("interarrivals_hndl" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->interarrivals_hndl);
        FOUT
        }
if (strcmp ("max_packet_count" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->max_packet_count);
        FOUT
        }
*var_p_ptr = (void *)OPC_NIL;

FOUT
}
```

APPENDIX B: RECEIVER SOURCE CODE

```
/* Process model C++ form file: SensorNetwork_Fifo.pr.cpp */
/* Portions of this file copyright 1986-2009 by OPNET Technologies, Inc. */


/*
 =========================== NOTE ==========================
 This file is automatically generated from SensorNetwork_Fifo.pr.m
 during a process model compilation.

 Do NOT manually edit this file.
 Manual edits will be lost during the next compilation.
 =========================== NOTE ==========================
*/



/* This variable carries the header into the object file */
const char SensorNetwork_Fifo_pr_cpp [] = "MIL_3_Tfile_Hdr_ 150A 30A modeler 7
4D227751 4D227751 1 rye-udcmybdtq7r Administrator 0 0 none none 0 0 none 0 0 0 0 0 0 0 0
21b7 3
";
#include <string.h>



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



/* Header Block */


#include <string>
#include        <oms_dist_support.h>

#include "hmac_sha1.hpp"
#include "sha1.hpp"

#define ARRIVAL     op_intrpt_type () == OPC_INTRPT_STRM
/* Node configuration constants.        */
#define         SSC_STRM_TO_LOW                          0

int num_pkts, num_pkts1;
int q_index;
int num_subqs, num_subqs1;
int packetCount=0;
int ind=-1;
int Service=0;
Packet *pkptr3 = OPC_NIL;
int receiverCountrPacket=0;
int queuePositionHead = OPC_QPOS_HEAD;


int indx;

static bool CompareMessages(int message, int KeySentByTransmitter, int nextmessage, int
macenqueued);


static void AccessQueue();
BYTE packetKeyEnqueued[14];
List* listOfStructures = OPC_NIL;

BYTE digestReceiver[4];

#ifndef INCLUDE_GENERALHASHFUNCTION_CPP_H
#define INCLUDE_GENERALHASHFUNCTION_CPP_H

typedef unsigned int (*HashFunction)(const std::string&);


int RSHashReceiver (const std::string& str);

#endif

/* End of Header Block */

#if !defined (VOSD_NO_FIN)
#undef BIN
#undef BOUT
#defineBIN          FIN_LOCAL_FIELD(_op_last_line_passed) = __LINE__ -
_op_block_origin;
#defineBOUT BIN
#defineBINIT FIN_LOCAL_FIELD(_op_last_line_passed) = 0; _op_block_origin = __LINE__;
#else
#defineBINIT
#endif /* #if !defined (VOSD_NO_FIN) */
```

```
/* State variable definitions */
class SensorNetwork_Fifo_state
        {
        private:
                /* Internal state tracking for FSM */
                FSM_SYS_STATE

        public:
                SensorNetwork_Fifo_state (void);

                /* Destructor contains Termination Block */
                ~SensorNetwork_Fifo_state (void);

                /* State Variables */
                Stathandle                    bits_rcvd_stathandle              ;
                Stathandle                    bitssec_rcvd_stathandle           ;
                Stathandle                    pkts_rcvd_stathandle              ;
                Stathandle                    pktssec_rcvd_stathandle           ;
                Stathandle                    ete_delay_stathandle              ;
                Stathandle                    bits_rcvd_gstathandle             ;
                Stathandle                    bitssec_rcvd_gstathandle          ;
                Stathandle                    pkts_rcvd_gstathandle             ;
                Stathandle                    pktssec_rcvd_gstathandle          ;
                Stathandle                    ete_delay_gstathandle             ;

                /* FSM code */
                void SensorNetwork_Fifo (OP_SIM_CONTEXT_ARG_OPT);
                /* Diagnostic Block */
                void _op_SensorNetwork_Fifo_diag (OP_SIM_CONTEXT_ARG_OPT);

#if defined (VOSD_NEW_BAD_ALLOC)
                void * operator new (size_t) throw (VOSD_BAD_ALLOC);
#else
                void * operator new (size_t);
#endif
                void operator delete (void *);

                /* Memory management */
                static VosT_Obtype obtype;
        };

VosT_Obtype SensorNetwork_Fifo_state::obtype = (VosT_Obtype)OPC_NIL;

#define bits_rcvd_stathandle            op_sv_ptr->bits_rcvd_stathandle
#define bitssec_rcvd_stathandle         op_sv_ptr->bitssec_rcvd_stathandle
```

```
#define pkts_rcvd_stathandle              op_sv_ptr->pkts_rcvd_stathandle
#define pktssec_rcvd_stathandle           op_sv_ptr->pktssec_rcvd_stathandle
#define ete_delay_stathandle              op_sv_ptr->ete_delay_stathandle
#define bits_rcvd_gstathandle             op_sv_ptr->bits_rcvd_gstathandle
#define bitssec_rcvd_gstathandle          op_sv_ptr->bitssec_rcvd_gstathandle
#define pkts_rcvd_gstathandle             op_sv_ptr->pkts_rcvd_gstathandle
#define pktssec_rcvd_gstathandle          op_sv_ptr->pktssec_rcvd_gstathandle
#define ete_delay_gstathandle             op_sv_ptr->ete_delay_gstathandle

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


/* Function Block */

#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ + 2};
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>


static bool CompareMessages (int message, int KeySentByTransmitter, int nextmessage, int
macenqueued)
        {
//      FIN (CompareMessages(int packetKey, int enqueuedMessage, int enqueuedMac, int
enqueuedNextMessage));
        FIN (CompareMessages(int message, int KeySentByTransmitter, int nextmessage, int
macenqueued));

                char nextMessageArray[4];
                char messageArray[4];
                char keyArray[4];

    itoa(message, messageArray, 10);
```

```
                              //itoa(message[++counter],nextMessage,10);
                    itoa(nextmessage, nextMessageArray, 10);

                    int h = RSHashReceiver(nextMessageArray);
                    int macMessage = h + message;

                    char messageHashed[4];

                    itoa(macMessage,messageHashed,10);
                    CHMAC_SHAReceiver HMAC_SHA1 ;
                    HMAC_SHA1.HMAC_SHAReceiver((unsigned char *)messageHashed ,
sizeof(messageHashed), (unsigned char *)itoa(KeySentByTransmitter,keyArray,10),
sizeof(keyArray), digestReceiver) ;


                    unsigned long hashedCode = 0;
                    unsigned long mult = 1;
                    for (unsigned i = 0; i < 4; ++i) {
                         hashedCode += mult * digestReceiver[sizeof(digestReceiver)-1-i];
                         mult <<= 8;
                    }
          bool rtn = true;
          FRET(rtn);

          }

int RSHashReceiver(const std::string& str)
{
   int b    = 3785;
   int a    = 63689;
   int hash = 0;

  for(int i = 0; i < str.length(); i++)
  {
    hash = hash * a + str[i];
    a    = a * b;

  }
          return (hash & 0x7FFFFFFF); //Returns the hashed string}
/* End Of RS Hash Function */
}

void CHMAC_SHAReceiver::HMAC_SHAReceiver(BYTE *text, int text_len, BYTE *key, int
key_len, BYTE *digest)
{
}

memset(SHA1_Key, 0, SHA1_BLOCK_SIZE);

/* repeated 64 times for values in ipad and opad */
memset(m_ipad, 0x36, sizeof(m_ipad));
memset(m_opad, 0x5c, sizeof(m_opad));


/* STEP 1 */
if (key_len > SHA1_BLOCK_SIZE)
{
        CSHAReceiver::ResetReceiver();
        CSHAReceiver::UpdateReceiver((UINT_8 *)key, key_len);
        CSHAReceiver::FinalReceiver();

        CSHAReceiver::GetHashReceiver((UINT_8 *)SHA1_Key);

}
else
        memcpy(SHA1_Key, key, key_len);

/* STEP 2 */
for (int i=0; i<sizeof(m_ipad); i++)
{
        m_ipad[i] ^= SHA1_Key[i];
}

/* STEP 3 */
memcpy(AppendBuf1, m_ipad, sizeof(m_ipad));
memcpy(AppendBuf1 + sizeof(m_ipad), text, text_len);

/* STEP 4 */
CSHAReceiver::ResetReceiver();
CSHAReceiver::UpdateReceiver((UINT_8 *)AppendBuf1, sizeof(m_ipad) + text_len);
CSHAReceiver::FinalReceiver();
CSHAReceiver::GetHashReceiver((UINT_8 *)szReport);


/* STEP 5 */
for (int j=0; j<sizeof(m_opad); j++)
{
        m_opad[j] ^= SHA1_Key[j];
}

/* STEP 6 */
memcpy(AppendBuf2, m_opad, sizeof(m_opad));
memcpy(AppendBuf2 + sizeof(m_opad), szReport, SHA1_DIGEST_LENGTH);
```

```
        /*STEP 7 */
        CSHAReceiver::ResetReceiver();
        CSHAReceiver::UpdateReceiver((UINT_8 *)AppendBuf2, sizeof(m_opad) +
SHA1_DIGEST_LENGTH);
        CSHAReceiver::FinalReceiver();

        CSHAReceiver::GetHashReceiver((UINT_8 *)digest);
}

#ifdef SHA1_UTILITY_FUNCTIONS
#define SHA1_MAX_FILE_BUFFER 8000
#endif

// Rotate x bits to the left
#ifndef ROL32
#ifdef _MSC_VER
#define ROL32(_val32, _nBits) _rotl(_val32, _nBits)
#else
#define ROL32(_val32, _nBits) (((_val32)<<(_nBits))|((_val32)>>(32-(_nBits))))
#endif
#endif

#ifdef SHA1_LITTLE_ENDIAN
#define SHABLK0(i) (m_block->l[i] = \
        (ROL32(m_block->l[i],24) & 0xFF00FF00) | (ROL32(m_block->l[i],8) & 0x00FF00FF))
#else
#define SHABLK0(i) (m_block->l[i])
#endif

#define SHABLK(i) (m_block->l[i&15] = ROL32(m_block->l[(i+13)&15] ^ m_block-
>l[(i+8)&15] \
        ^ m_block->l[(i+2)&15] ^ m_block->l[i&15],1))

// SHA-1 rounds
#define _R0(v,w,x,y,z,i) { z+=((w&(x^y))^y)+SHABLK0(i)+0x5A827999+ROL32(v,5);
w=ROL32(w,30); }
#define _R1(v,w,x,y,z,i) { z+=((w&(x^y))^y)+SHABLK(i)+0x5A827999+ROL32(v,5);
w=ROL32(w,30); }
#define _R2(v,w,x,y,z,i) { z+=(w^x^y)+SHABLK(i)+0x6ED9EBA1+ROL32(v,5);
w=ROL32(w,30); }
#define _R3(v,w,x,y,z,i) { z+=(((w|x)&y)|(w&x))+SHABLK(i)+0x8F1BBCDC+ROL32(v,5);
w=ROL32(w,30); }
#define _R4(v,w,x,y,z,i) { z+=(w^x^y)+SHABLK(i)+0xCA62C1D6+ROL32(v,5);
w=ROL32(w,30); }
```

```
CSHAReceiver::CSHAReceiver()
{
        m_block = (SHA1_WORKSPACE_BLOCK *)m_workspace;

        ResetReceiver();
}

CSHAReceiver::~CSHAReceiver()
{
        ResetReceiver();
}

void CSHAReceiver::ResetReceiver()
{
        // SHA1 initialization constants
        m_state[0] = 0x67452301;
        m_state[1] = 0xEFCDAB89;
        m_state[2] = 0x98BADCFE;
        m_state[3] = 0x10325476;
        m_state[4] = 0xC3D2E1F0;

        m_count[0] = 0;
        m_count[1] = 0;
}

void CSHAReceiver::TransformReceiver(UINT_32 *state, UINT_8 *buffer)
{
        // Copy state[] to working vars
        UINT_32 a = state[0], b = state[1], c = state[2], d = state[3], e = state[4];

        memcpy(m_block, buffer, 64);

        // 4 rounds of 20 operations each. Loop unrolled.
        _R0(a,b,c,d,e, 0); _R0(e,a,b,c,d, 1); _R0(d,e,a,b,c, 2); _R0(c,d,e,a,b, 3);
        _R0(b,c,d,e,a, 4); _R0(a,b,c,d,e, 5); _R0(e,a,b,c,d, 6); _R0(d,e,a,b,c, 7);
        _R0(c,d,e,a,b, 8); _R0(b,c,d,e,a, 9); _R0(a,b,c,d,e,10); _R0(e,a,b,c,d,11);
        _R0(d,e,a,b,c,12); _R0(c,d,e,a,b,13); _R0(b,c,d,e,a,14); _R0(a,b,c,d,e,15);
        _R1(e,a,b,c,d,16); _R1(d,e,a,b,c,17); _R1(c,d,e,a,b,18); _R1(b,c,d,e,a,19);
        _R2(a,b,c,d,e,20); _R2(e,a,b,c,d,21); _R2(d,e,a,b,c,22); _R2(c,d,e,a,b,23);
        _R2(b,c,d,e,a,24); _R2(a,b,c,d,e,25); _R2(e,a,b,c,d,26); _R2(d,e,a,b,c,27);
        _R2(c,d,e,a,b,28); _R2(b,c,d,e,a,29); _R2(a,b,c,d,e,30); _R2(e,a,b,c,d,31);
        _R2(d,e,a,b,c,32); _R2(c,d,e,a,b,33); _R2(b,c,d,e,a,34); _R2(a,b,c,d,e,35);
        _R2(e,a,b,c,d,36); _R2(d,e,a,b,c,37); _R2(c,d,e,a,b,38); _R2(b,c,d,e,a,39);
        _R3(a,b,c,d,e,40); _R3(e,a,b,c,d,41); _R3(d,e,a,b,c,42); _R3(c,d,e,a,b,43);
        _R3(b,c,d,e,a,44); _R3(a,b,c,d,e,45); _R3(e,a,b,c,d,46); _R3(d,e,a,b,c,47);
        _R3(c,d,e,a,b,48); _R3(b,c,d,e,a,49); _R3(a,b,c,d,e,50); _R3(e,a,b,c,d,51);
        _R3(d,e,a,b,c,52); _R3(c,d,e,a,b,53); _R3(b,c,d,e,a,54); _R3(a,b,c,d,e,55);
        _R3(e,a,b,c,d,56); _R3(d,e,a,b,c,57); _R3(c,d,e,a,b,58); _R3(b,c,d,e,a,59);
        _R4(a,b,c,d,e,60); _R4(e,a,b,c,d,61); _R4(d,e,a,b,c,62); _R4(c,d,e,a,b,63);
        _R4(b,c,d,e,a,64); _R4(a,b,c,d,e,65); _R4(e,a,b,c,d,66); _R4(d,e,a,b,c,67);
        _R4(c,d,e,a,b,68); _R4(b,c,d,e,a,69); _R4(a,b,c,d,e,70); _R4(e,a,b,c,d,71);
        _R4(d,e,a,b,c,72); _R4(c,d,e,a,b,73); _R4(b,c,d,e,a,74); _R4(a,b,c,d,e,75);
        _R4(e,a,b,c,d,76); _R4(d,e,a,b,c,77); _R4(c,d,e,a,b,78); _R4(b,c,d,e,a,79);

        // Add the working vars back into state
        state[0] += a;
        state[1] += b;
        state[2] += c;
        state[3] += d;
        state[4] += e;

        // Wipe variables
#ifdef SHA1_WIPE_VARIABLES
        a = b = c = d = e = 0;
#endif
}

// Use this function to hash in binary data and strings
void CSHAReceiver::UpdateReceiver(UINT_8 *data, UINT_32 len)
{
        UINT_32 i, j;

        j = (m_count[0] >> 3) & 63;

        if((m_count[0] += len << 3) < (len << 3)) m_count[1]++;

        m_count[1] += (len >> 29);

        if((j + len) > 63)
        {
                i = 64 - j;
                memcpy(&m_buffer[j], data, i);
                TransformReceiver(m_state, m_buffer);

                for(; i + 63 < len; i += 64) TransformReceiver(m_state, &data[i]);

                j = 0;
        }
        else i = 0;

        memcpy(&m_buffer[j], &data[i], len - i);
}
```

```
#ifdef SHA1_UTILITY_FUNCTIONS
// Hash in file contents
bool CSHAReceiver::HashFileReceiver(char *szFileName)
{
        unsigned long ulFileSize, ulRest, ulBlocks;
        unsigned long i;
        UINT_8 uData[SHA1_MAX_FILE_BUFFER];
        FILE *fIn;

        if(szFileName == NULL) return false;

        fIn = fopen(szFileName, "rb");
        if(fIn == NULL) return false;

        fseek(fIn, 0, SEEK_END);
        ulFileSize = (unsigned long)ftell(fIn);
        fseek(fIn, 0, SEEK_SET);

        if(ulFileSize != 0)
        {
                ulBlocks = ulFileSize / SHA1_MAX_FILE_BUFFER;
                ulRest = ulFileSize % SHA1_MAX_FILE_BUFFER;
        }
        else
        {
                ulBlocks = 0;
                ulRest = 0;
        }

        for(i = 0; i < ulBlocks; i++)
        {
                fread(uData, 1, SHA1_MAX_FILE_BUFFER, fIn);
                UpdateReceiver((UINT_8 *)uData, SHA1_MAX_FILE_BUFFER);
        }

        if(ulRest != 0)
        {
                fread(uData, 1, ulRest, fIn);
                UpdateReceiver((UINT_8 *)uData, ulRest);
        }

        fclose(fIn); fIn = NULL;
        return true;
}
#endif
```

```
void CSHAReceiver::FinalReceiver()
{
        UINT_32 i;
        UINT_8 finalcount[8];

        for(i = 0; i < 8; i++)
                finalcount[i] = (UINT_8)((m_count[((i >= 4) ? 0 : 1)]
                        >> ((3 - (i & 3)) * 8) ) & 255); // Endian independent

        UpdateReceiver((UINT_8 *)"\200", 1);

        while ((m_count[0] & 504) != 448)
                UpdateReceiver((UINT_8 *)"\0", 1);

        UpdateReceiver(finalcount, 8); // Cause a SHA1Transform()

        for(i = 0; i < 20; i++)
        {
                m_digest[i] = (UINT_8)((m_state[i >> 2] >> ((3 - (i & 3)) * 8) ) & 255);
        }


        // Wipe variables for security reasons
#ifdef SHA1_WIPE_VARIABLES
        i = 0;
        memset(m_buffer, 0, 64);
        memset(m_state, 0, 20);
        memset(m_count, 0, 8);
        memset(finalcount, 0, 8);
        TransformReceiver(m_state, m_buffer);
#endif
}

#ifdef SHA1_UTILITY_FUNCTIONS
// Get the final hash as a pre-formatted string
void CSHAReceiver::ReportHashReceiver(char *szReport, unsigned char uReportType)
{
        unsigned char i;
        char szTemp[16];

        if(szReport == NULL) return;

        if(uReportType == REPORT_HEX)
        {
                sprintf(szTemp, "%02X", m_digest[0]);
                strcat(szReport, szTemp);

                for(i = 1; i < 20; i++)
                {
                        sprintf(szTemp, " %02X", m_digest[i]);
                        strcat(szReport, szTemp);
                }
        }
        else if(uReportType == REPORT_DIGIT)
        {
                sprintf(szTemp, "%u", m_digest[0]);
                strcat(szReport, szTemp);

                for(i = 1; i < 20; i++)
                {
                        sprintf(szTemp, " %u", m_digest[i]);
                        strcat(szReport, szTemp);
                }
        }
        else strcpy(szReport, "Error: Unknown report type!");
}
#endif


// Get the raw message digest
void CSHAReceiver::GetHashReceiver(UINT_8 *puDest)
{
        memcpy(puDest, m_digest, 20);
}

/* End of Function Block */

/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.              */
#undef FIN_TRACING
#define FIN_TRACING

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

/* Undefine shortcuts to state variables because the */
/* following functions are part of the state class */
#undef bits_rcvd_stathandle
#undef bitssec_rcvd_stathandle
#undef pkts_rcvd_stathandle
```

```
#undef pktssec_rcvd_stathandle
#undef ete_delay_stathandle
#undef bits_rcvd_gstathandle
#undef bitssec_rcvd_gstathandle
#undef pkts_rcvd_gstathandle
#undef pktssec_rcvd_gstathandle
#undef ete_delay_gstathandle

/* Access from C kernel using C linkage */
extern "C"
{
	VosT_Obtype _op_SensorNetwork_Fifo_init (int * init_block_ptr);
	VosT_Address _op_SensorNetwork_Fifo_alloc (VosT_Obtype, int);
	void SensorNetwork_Fifo (OP_SIM_CONTEXT_ARG_OPT)
		{
		((SensorNetwork_Fifo_state *)(OP_SIM_CONTEXT_PTR-
>_op_mod_state_ptr))->SensorNetwork_Fifo (OP_SIM_CONTEXT_PTR_OPT);
		}

	void _op_SensorNetwork_Fifo_svar (void *, const char *, void **);

	void _op_SensorNetwork_Fifo_diag (OP_SIM_CONTEXT_ARG_OPT)
		{
		((SensorNetwork_Fifo_state *)(OP_SIM_CONTEXT_PTR-
>_op_mod_state_ptr))->_op_SensorNetwork_Fifo_diag (OP_SIM_CONTEXT_PTR_OPT);
		}

	void _op_SensorNetwork_Fifo_terminate (OP_SIM_CONTEXT_ARG_OPT)
		{
		/* The destructor is the Termination Block */
		delete (SensorNetwork_Fifo_state *)(OP_SIM_CONTEXT_PTR-
>_op_mod_state_ptr);
		}


} /* end of 'extern "C"' */




/* Process model interrupt handling procedure */


void
SensorNetwork_Fifo_state::SensorNetwork_Fifo (OP_SIM_CONTEXT_ARG_OPT)
	{
```

```
#if !defined (VOSD_NO_FIN)
        int _op_block_origin = 0;
#endif
        FIN_MT (SensorNetwork_Fifo_state::SensorNetwork_Fifo ());
        try
                {
                /* Temporary Variables */
                //Packet*                 pkptr;
                double          pk_size;
                double          ete_delay;
                /* End of Temporary Variables */


                FSM_ENTER ("SensorNetwork_Fifo")

                FSM_BLOCK_SWITCH
                        {
                        /*---------------------------------------------------------*/
                        /** state (INS_TAIL) enter executives **/
                        FSM_STATE_ENTER_UNFORCED (0, "INS_TAIL", state0_enter_exec,
"SensorNetwork_Fifo [INS_TAIL enter execs]")
                                FSM_PROFILE_SECTION_IN ("SensorNetwork_Fifo
[INS_TAIL enter execs]", state0_enter_exec)
                                {


                                /* Obtain the incoming packet.          */
                                pkptr3 = op_pk_get (op_intrpt_strm ());
                                ++packetCount;

                                /* Caclulate metrics to be updated.            */
                                pk_size = (double) op_pk_total_size_get (pkptr3);
                                ete_delay = op_subq_stat (0, OPC_QSTAT_DELAY) ;

                                if( pk_size == 96)

                                        {
                                                Service = 1;
                                        }
                                else
                                        {

                                        if (op_subq_pk_insert (0, pkptr3, OPC_QPOS_TAIL) !=
OPC_QINS_OK)
                                                {
                                                        op_pk_destroy (pkptr3);
                                                printf ("Queue - op_pk_destroyed");
                                        }
                                                Service = 0;
                                }

                        }
                        FSM_PROFILE_SECTION_OUT (state0_enter_exec)

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


                /** state (INS_TAIL) exit executives **/
                FSM_STATE_EXIT_UNFORCED (0, "INS_TAIL",
"SensorNetwork_Fifo [INS_TAIL exit execs]")


                /** state (INS_TAIL) transition processing **/
                FSM_PROFILE_SECTION_IN ("SensorNetwork_Fifo [INS_TAIL trans
conditions]", state0_trans_conds)
                FSM_INIT_COND (Service)
                FSM_TEST_COND (!Service)
                FSM_TEST_LOGIC ("INS_TAIL")
                FSM_PROFILE_SECTION_OUT (state0_trans_conds)

                FSM_TRANSIT_SWITCH
                        {
                        FSM_CASE_TRANSIT (0, 1, state1_enter_exec, ;, "Service", "",
"INS_TAIL", "SEND_HEAD", "tr_4", "SensorNetwork_Fifo [INS_TAIL -> SEND_HEAD :
Service / ]")
                        FSM_CASE_TRANSIT (1, 2, state2_enter_exec, ;, "!Service", "",
"INS_TAIL", "BRANCH", "tr_5", "SensorNetwork_Fifo [INS_TAIL -> BRANCH : !Service /
]")
                        }
                        /*---------------------------------------------------------*/



                /** state (SEND_HEAD) enter executives **/
                FSM_STATE_ENTER_UNFORCED (1, "SEND_HEAD",
state1_enter_exec, "SensorNetwork_Fifo [SEND_HEAD enter execs]")
                        FSM_PROFILE_SECTION_IN ("SensorNetwork_Fifo
[SEND_HEAD enter execs]", state1_enter_exec)
                        {
                        Objid subq_objid;
                        int privateKey=0;
```

```
                                        int message;
                                        int nextmessage;
                                        int macenqueued=0;
                                        Packet *queuedPacket;

                                        op_ima_obj_attr_get (op_id_self (), "subqueue", &subq_objid);
                                        num_subqs = op_topo_child_count (subq_objid,
OPC_OBJTYPE_SUBQ);
                                        //op_subq_pk_remove (0, OPC_QPOS_HEAD);

                                        op_pk_fd_get (pkptr3, 0, &message);
                                        op_pk_fd_get (pkptr3, 1, &privateKey);
                                        op_pk_fd_get (pkptr3, 2, &nextmessage);

                                        queuedPacket = op_subq_pk_access (0, queuePositionHead);
                                        op_pk_fd_get (queuedPacket, 0, &macenqueued);


                                        if (CompareMessages(message, privateKey, nextmessage,
macenqueued))
                                                        op_subq_pk_remove (0, OPC_QPOS_HEAD);
                                        else
                                                queuePositionHead += OPC_QPOS_HEAD;


                                        }
                                        FSM_PROFILE_SECTION_OUT (state1_enter_exec)

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


                                /** state (SEND_HEAD) exit executives **/
                                FSM_STATE_EXIT_UNFORCED (1, "SEND_HEAD",
"SensorNetwork_Fifo [SEND_HEAD exit execs]")


                                /** state (SEND_HEAD) transition processing **/
                                FSM_TRANSIT_FORCE (2, state2_enter_exec, ;, "default", "",
"SEND_HEAD", "BRANCH", " _0", "SensorNetwork_Fifo [SEND_HEAD -> BRANCH :
default / ]")
                                        /*---------------------------------------------------------*/

                                /** state (BRANCH) enter executives **/
                                FSM_STATE_ENTER_FORCED (2, "BRANCH", state2_enter_exec,
"SensorNetwork_Fifo [BRANCH enter execs]")
```

```
                        FSM_PROFILE_SECTION_IN ("SensorNetwork_Fifo [BRANCH
enter execs]", state2_enter_exec)
                        {

                        /* Initilaize the statistic handles to keep        */
                        /* track of traffic sinked by this process.*/


                        bits_rcvd_stathandle          = op_stat_reg ("Traffic Received
(bits)",            OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        bitssec_rcvd_stathandle       = op_stat_reg ("Traffic Received
(bits/sec)",        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        pkts_rcvd_stathandle          = op_stat_reg ("Traffic Received
(packets)",         OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        pktssec_rcvd_stathandle       = op_stat_reg ("Traffic Received
(packets/sec)",OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        ete_delay_stathandle          = op_stat_reg ("End-to-End Delay
(seconds)",         OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        //head = OPC_QPOS_HEAD;
                        }
                        FSM_PROFILE_SECTION_OUT (state2_enter_exec)

                    /** state (BRANCH) exit executives **/
                    FSM_STATE_EXIT_FORCED (2, "BRANCH", "SensorNetwork_Fifo
[BRANCH exit execs]")


                    /** state (BRANCH) transition processing **/
                    FSM_TRANSIT_ONLY ((ARRIVAL), 0, state0_enter_exec, ;,
BRANCH, "ARRIVAL", "", "BRANCH", "INS_TAIL", " _2", "SensorNetwork_Fifo
[BRANCH -> INS_TAIL : ARRIVAL / ]")
                        /*---------------------------------------------------*/

                    }

                FSM_EXIT (2,"SensorNetwork_Fifo")
                }
        catch (...)
                {
                Vos_Error_Print (VOSC_ERROR_ABORT,
                    (const char *)VOSC_NIL,
                    "Unhandled C++ exception in process model (SensorNetwork_Fifo)",
                    (const char *)VOSC_NIL, (const char *)VOSC_NIL);
                }
        }
```

```
void
SensorNetwork_Fifo_state::_op_SensorNetwork_Fifo_diag (OP_SIM_CONTEXT_ARG_OPT)
	{
	/* No Diagnostic Block */
	}

void
SensorNetwork_Fifo_state::operator delete (void* ptr)
	{
	FIN (SensorNetwork_Fifo_state::operator delete (ptr));
	Vos_Poolmem_Dealloc (ptr);
	FOUT
	}

SensorNetwork_Fifo_state::~SensorNetwork_Fifo_state (void)
	{

	FIN (SensorNetwork_Fifo_state::~SensorNetwork_Fifo_state ())


	/* No Termination Block */


	FOUT
	}


#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

void *
SensorNetwork_Fifo_state::operator new (size_t)
#if defined (VOSD_NEW_BAD_ALLOC)
		throw (VOSD_BAD_ALLOC)
#endif
	{
	void * new_ptr;

	FIN_MT (SensorNetwork_Fifo_state::operator new ());

	new_ptr = Vos_Alloc_Object (SensorNetwork_Fifo_state::obtype);
#if defined (VOSD_NEW_BAD_ALLOC)
	if (new_ptr == VOSC_NIL) throw VOSD_BAD_ALLOC();
```

```
#endif
        FRET (new_ptr)
        }

/* State constructor initializes FSM handling */
/* by setting the initial state to the first */
/* block of code to enter. */

SensorNetwork_Fifo_state::SensorNetwork_Fifo_state (void) :
                _op_current_block (4)
        {
#if defined (OPD_ALLOW_ODB)
                _op_current_state = "SensorNetwork_Fifo [BRANCH enter execs]";
#endif
        }

VosT_Obtype
_op_SensorNetwork_Fifo_init (int * init_block_ptr)
        {
        FIN_MT (_op_SensorNetwork_Fifo_init (init_block_ptr))

        SensorNetwork_Fifo_state::obtype = Vos_Define_Object_Prstate ("proc state vars
(SensorNetwork_Fifo)",
                sizeof (SensorNetwork_Fifo_state));
        *init_block_ptr = 4;

        FRET (SensorNetwork_Fifo_state::obtype)
        }

VosT_Address
_op_SensorNetwork_Fifo_alloc (VosT_Obtype, int)
        {
#if !defined (VOSD_NO_FIN)
        int _op_block_origin = 0;
#endif
        SensorNetwork_Fifo_state * ptr;
        FIN_MT (_op_SensorNetwork_Fifo_alloc ())

        /* New instance will have FSM handling initialized */
#if defined (VOSD_NEW_BAD_ALLOC)
        try {
                ptr = new SensorNetwork_Fifo_state;
        } catch (const VOSD_BAD_ALLOC &) {
                ptr = VOSC_NIL;
        }
#else
```

```
        ptr = new SensorNetwork_Fifo_state;
#endif
        FRET ((VosT_Address)ptr)
        }



void
_op_SensorNetwork_Fifo_svar (void * gen_ptr, const char * var_name, void ** var_p_ptr)
        {
        SensorNetwork_Fifo_state            *prs_ptr;

        FIN_MT (_op_SensorNetwork_Fifo_svar (gen_ptr, var_name, var_p_ptr))

        if (var_name == OPC_NIL)
                {
                *var_p_ptr = (void *)OPC_NIL;
                FOUT
                }
        prs_ptr = (SensorNetwork_Fifo_state *)gen_ptr;

        if (strcmp ("bits_rcvd_stathandle" , var_name) == 0)
                {
                *var_p_ptr = (void *) (&prs_ptr->bits_rcvd_stathandle);
                FOUT
                }
        if (strcmp ("bitssec_rcvd_stathandle" , var_name) == 0)
                {
                *var_p_ptr = (void *) (&prs_ptr->bitssec_rcvd_stathandle);
                FOUT
                }
        if (strcmp ("pkts_rcvd_stathandle" , var_name) == 0)
                {
                *var_p_ptr = (void *) (&prs_ptr->pkts_rcvd_stathandle);
                FOUT
                }
        if (strcmp ("pktssec_rcvd_stathandle" , var_name) == 0)
                {
                *var_p_ptr = (void *) (&prs_ptr->pktssec_rcvd_stathandle);
                FOUT
                }
        if (strcmp ("ete_delay_stathandle" , var_name) == 0)
                {
                *var_p_ptr = (void *) (&prs_ptr->ete_delay_stathandle);
                FOUT
                }
```

```
if (strcmp ("bits_rcvd_gstathandle" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->bits_rcvd_gstathandle);
        FOUT
        }
if (strcmp ("bitssec_rcvd_gstathandle" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->bitssec_rcvd_gstathandle);
        FOUT
        }
if (strcmp ("pkts_rcvd_gstathandle" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->pkts_rcvd_gstathandle);
        FOUT
        }
if (strcmp ("pktssec_rcvd_gstathandle" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->pktssec_rcvd_gstathandle);
        FOUT
        }
if (strcmp ("ete_delay_gstathandle" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->ete_delay_gstathandle);
        FOUT
        }
*var_p_ptr = (void *)OPC_NIL;

FOUT
}
```

LIST OF REFERENCES

1. A.E. Hegazy, A.M. Darwish, R. El-Fouly, "Reducing μTESLA memory requirements", Second International Conference on Systems and Networks Communications, ICSNC, pp.33, 2007
2. Adrian Perrig, Ran Canetti, Dawn Song, J.D. Tygar, "Efficient and Secure Source Authentication for Multicast", In Network and Distributed System Security Symposium, NDSS, 35 – 46, 2001
3. Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, J. D. Tygar, "SPINS: Security Protocols for Sensor Networks, Wireless Networks", 8, 521.534, 2002
4. Adrian Perrig, Ran Canetti, J.D. Tygar, Dawn Song, "The TESLA Broadcast Authentication Protocol", RSA CryptoBytes, volume 5, 2002
5. Mark Luk, Adrian Perrig, Bram Whillock, " Seven Cardinal Properties of Sensor Network Broadcast Authentication", Proceedings of the Fourth ACM Workshop on Security of ad hoc and Sensor Networks, SASN. pp. 147-156, 2006
6. Donggang Liu, Peng Ning, Sencun Zhu, Sushil Jajodia, "Practical Broadcast Authentication in Sensor Networks", The Second Annual International Conference in, 118 – 129, July 2005
7. Kui Ren, *Member, IEEE*, Wenjing Lou, *Member, IEEE*, Kai Zeng, *Student Member, IEEE*, and Patrick J. Moran, "On Broadcast Authentication in Wireless Sensor Networks", IEEE Transcations on Wireless Communications, Vol. 6, No. 11, November 2007
8. Peng Ning, An Liu, Wenliang Du, "Mitigating DoS Attacks against Broadcast Authentication in Wireless Sensor Networks", ACM Transactions on Sensor Networks, Volume 4, Issue 1, January 2008
9. [9] Donggang Liu and Peng Ning, "Multi-Level μTESLA: Broadcast Authentication for Distributed Sensor Networks", ACM Transactions on Embedded Computing Systems, Volume 3 Issue 4, November 2004
10. El Kaissi, Rouba Zakaria, "DAWWSEN: A Defense Mechanism Against Wormhole Attacks in Wireless Sensor Networks", The Second International Conference on Innovations in Information Technology, 2005

11. Grigorios Katsis, “Multistage security mechanism for hybrid, large scale wireless sensor networks”, Master's thesis, Naval Postgraduate School, June 2007
12. Arayeh Norouzi, Abdolreza Abhari, Truman Yang, "Enhancing Broadcast Authentication in Sensor Networks", Proceedings of the 2010 Spring Simulation Multi-conference, 2010
13. Wenliang Du, Ronghua Wang, Peng Ning, “An Efficient Scheme for Authenticating Public Keys in Sensor Networks”, Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing, 2005
14. Wen-Huei Chen and Yu-Jen Chen, “A Bootstrapping Scheme for Inter-Sensor Authentication within Sensor Networks”, IEEE Communication Letters, Vol. 9, No. 10, October 2005
15. Yih-Chun Hu, Adrian Perrig, David B. Johnson, “Wormhole Attacks in Wireless Networks”, Selected Areas in Communications, IEEE Journal, Volume: 24 Issue: 2, 370 - 380, Feb. 2006
16. Yih-Chun Hu, Adrian Perrig, David B. Johnson, “Packet Leashes: A Defense against Wormhole Attacks in Wireless Ad Hoc Networks”, Joint Conference of the IEEE Computer and Communications, 1976 - 1986 vol.3, April 2003
17. David R. Raymond, Scott F. Midkiff, “Denial-of-Service in Wireless Sensor Networks: Attacks and Defenses”, IEEE Pervasive Computing 7 (1), pp. 74-81, March 2008
18. Veronika Kondratieva and Seung-Woo Seo, *Member, IEEE,* "Optimized Hash Tree for Authentication in Sensor Networks”, IEEE Communication Letters, Vol. 11, No. 2, February 2007
19. Elain Shi, Agrian Perrig, “Designing Secure Sensor Networks”, Wireless Communications, IEEE, Volume: 11 Issue:6, Pages: 38 - 43, Dec 2004
20. Wen-Huei Chen, Yu-Jen Chen, “A C-μTesla Protocol for Sensor Networks”, Journal of Informatics & Electronics, Vol.2, No.2, pp.29-32, March 2008
21. Zinaida Benenson, Nils Gedicke, Ossi Raivio, “Realizing Robust User Authentication in Sensor Networks”, 2005
22. Mathias Bohge, Wade Trappe, “TESLA Certificates: An Authentication Tool for Networks of Compute-Constrained Devices”, In Proc. Of 2003 workshop on Wireless Security, August 2003

23. Zinaida Benenson, Felix G¨artner, Dogan Kesdogan,"User Authentication in Sensor Networks (Extended Abstract)", In Proceedings of Informatik 2004, Workshop on Sensor Networks, 2004
24. Roberto Di Pietro, Luigi V. Mancini, Alessandro Mei, "Energy efficient node-to-node authentication and communication confidentiality in wireless sensor networks", Wireless Networks 12 (6), pp. 709-721, November 2006
25. Huei-Ru Tseng, Rong-Hong Jan, and Wuu Yang, "An Improved Dynamic User Authentication Scheme for Wireless Sensor Networks", Gobal Telecommunications Conference, IEEE, 86 - 990, November 2007
26. [26] Jeffery Undercoffer, Sasikanth Avancha, Anupam Joshi, John Pinkston, "Security for Sensor Networks", Chapter 12, 253-275, 2004
27. Kui Ren, Wenjing Lou, Yanchao Zhang, "Mulit-user Broadcast Authentication in Wireless Sensor Networks", Sensor, Mesh and Ad Hoc Communications and Networks, 2007, 223 - 232, June 2007
28. Ronghua Wang, Wenliang Du, Peng Ning, "Containing Denial-of-Service Attacks in Broadcast Authentication in Sensor Networks", Proceedings of the 8th ACM international symposium on Mobile ad hoc networking and computing, 2007
29. Xiaojian Tian, Duncan S. Wong, *Member, IEEE,* and Robert W. Zhu, *Student Member, IEEE,* "Analysis and Improvement of an Authenticated Key exchange Protocol for Sensor Networks", IEEE Communications Letters, Vol. 9, No. 11, November 2005
30. Jian Wang, Z Y Xia, Lein Harn, "Storage-Optimal Key Sharing with Authentication in Sensor Networks", IET Conference Publications (496 CP), pp. 134, 2005
31. Xuefei Cao , Weidong Kou, Lanjun Dang, Bin Zhao, "IMBAS: Identity-based multi-user broadcast authentication in wireless sensor networks", Computer Communications 31, 659–667, 2008
32. F. L. Lewis, "Wireless Sensor Networks", To appear in Smart Environments: Technologies, Protocols, and Applications ed. D.J. Cook and S.K. Das, John Wiley, New York, 2004
33. Mike Chen, Weidong Cui, Victor Wen, "Security and Deployment Issues in a Sensor Network", International Journal Of Communications, Issue 1, Volume 2, 2008

34. Taojun Wu, Yi Cui, Brano Kusy, Akos Ledeczi, Janos Sallai, Nathan Skirvin, Jan Werner, Yuan Xue, "A Fast and Efficient Source Authentication Solution for Broadcasting in Wireless Sensor Networks", Proceedings of New Technologies, Mobility and Security, ifip, IEEE, May, 2007
35. Harald Vogt, "Exploring Message Authentication in Sensor Networks", Lecture Notes in Computer Science 3313, pp. 19-30, 2005
36. Li Zhou, Chinya V. Ravishankar, "Dynamic Merkle Trees for Verifying Privileges in Sensor Networks", Communications, 2006. ICC '06, IEEE, Volume: 5, 2276 - 2282, June 2006
37. Truman Yang, "Computer Network Security Lecture Notes", Ryerson University, 2008
38. Tommy Svensson, Alex Popescu, "OPNET Modeler", Master Thesis, Blekinge Institute of Technology, June 2003
39. OPNET Modeler Help