University of North Florida
UNF Digital Commons

UNF Graduate Theses and Dissertations | Student Scholarship

2011

Performance of BFSA Based Anti-Collision Protocols for RFID Networks Supporting Identical Tags

Kirti Chemburkar
University of North Florida

Follow this and additional works at: https://digitalcommons.unf.edu/etd

 Part of the Computer Sciences Commons

Suggested Citation

Chemburkar, Kirti, "Performance of BFSA Based Anti-Collision Protocols for RFID Networks Supporting Identical Tags" (2011). *UNF Graduate Theses and Dissertations*. 124.
https://digitalcommons.unf.edu/etd/124

This Master's Thesis is brought to you for free and open access by the Student Scholarship at UNF Digital Commons. It has been accepted for inclusion in UNF Graduate Theses and Dissertations by an authorized administrator of UNF Digital Commons. For more information, please contact Digital Projects.
© 2011 All Rights Reserved

PERFORMANCE OF BFSA BASED ANTI-COLLISION PROTOCOLS FOR RFID NETWORKS SUPPORTING IDENTICAL TAGS

by

Kirti Chemburkar

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

August 2011

Copyright (©) 2011 by Kirti Chemburkar

All rights reserved. Reproduction in whole or in part in any form requires the prior written permission of Kirti Chemburkar or designated representative.

The thesis "Performance of BFSA Based Anti-Collision Protocols for RFID Networks Supporting Identical Tags" submitted by Kirti Chemburkar in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

6-7-11

Zornitza Genova Prodanoff, Ph.D.
Thesis Advisor and Committee Chairperson

Signature Deleted

6-7-11

Kenneth E. Martin, Ph.D.

Signature Deleted

6-7-11

Susan Vasana, Ph.D.

Accepted for the School of Computing:

Signature Deleted

8.5.11

Asai Asaithambi, Ph.D.
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

Signature Deleted

8/8/11

Mark Tumeo, Ph.D.
Dean of the College

Accepted for the University:

Signature Deleted

8/15/11

Len Roberson, Ph.D.
Dean of the Graduate School

ACKNOWLEDGEMENT

I wish to thank my spouse for his unwavering support and understanding during the many hours I dedicated to achieving this milestone in my life and career. I also wish to thank my thesis advisor Dr. Prodanoff for her valuable insights into the thesis topic and her inspiring and uplifting spirit.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Radio Frequency Identification (RFID) is a powerful emerging technology widely used for asset tracking, supply chain management, animal identification, military applications, payment systems, and access control. Over the years, RFID has emerged as a popular technology in various industries because of its ability to track moving objects. As RFID is becoming less expensive and more robust, many companies and vendors are developing tags to track objects. Multiple vendors manufacture RFID tags worldwide. Therefore, it is quite possible that they manufacture tags with the same identification code (ID) as vendor ID code data sets may not be synchronized or may be subject to tag id errors. Due to this drawback, there is the possibility that non-unique tags exist along with unique tags in the same RFID system. As existing implementations optimize the performance of RFID systems performance based on the assumption of unique tags, it is important to study the effect of non-unique tags on RFID systems.

This thesis focuses on a formal analysis of the Basic Frame Slotted ALOHA (BFSA) Muting RFID system with non-unique tags. An RFID network was modeled with OPNET Modeler 14.5. An evaluation model was built to measure the total census delay, optimal frame size, and network throughput for an RFID network based on a BFSA protocol for non-unique tags and support for muting. The evaluation results are in agreement with results obtained from the evaluation of a similar model for unique tags [Kang08]. Comparing total census delay for unique and non-unique tags for variable frame sizes showed an increase in total census delay with an increase in the number of tags. Comparing minimum network

throughput, mean network throughput, and maximum network throughput for unique and non-unique tags for variable frame sizes showed a decrease in network throughput with an increase in the number of tags.

Chapter 1

INTRODUCTION

Over the last 30 years, supply chain management and logistics companies have successfully implemented Radio Frequency Identification (RFID) systems all over the world. Due to the declining cost of the implementation of RFID systems, many companies have recognized the benefits of implementing them. RFID technology uses radio frequency waves to transfer data between a reader and an item to identify, to track, or to locate. The advantage of RFID technology is that the reader can read tags without contact and beyond the line-of-sight. The reader can read tags through a variety of visually and environmentally challenging conditions such as snow, ice, fog, paint, grime, inside containers and vehicles, and while in storage [Roberts06]. The versatile applicability of the RFID systems have led companies to implement them.

1.1 Radio Frequency Identification (RFID) System

Figure 1 shows an RFID system that consists of three components – the first component is an application installed on a computer to manage readers, the second component is a reader device that collects information from tags and transmits it to the computer application, and the third component is a tag attached to an item to identify. RFID is a technology in which the reader (or interrogator) broadcasts radio signals to sense the presence of tags and identify items with attached RFID tags. The reader identifies tags through a census. The census is the process of reading tags within an interrogation zone

[Prodanoff10]. The main advantage of an RFID system over a barcode and smartcard system is that tag identification is possible beyond the line of sight of the reader and without any contact with the reader. An RFID system can also successfully identify tags up to three meters for a far-field reader, unlike the barcode and smartcards [Want06]. Some commercial RFID tags can store data from about 16 Kbytes to 64 Kbytes unlike the barcode and smartcards [Finkenzeller03]. RFID is a relatively new and emerging technology to overcome deficiencies in barcode and smartcard systems.

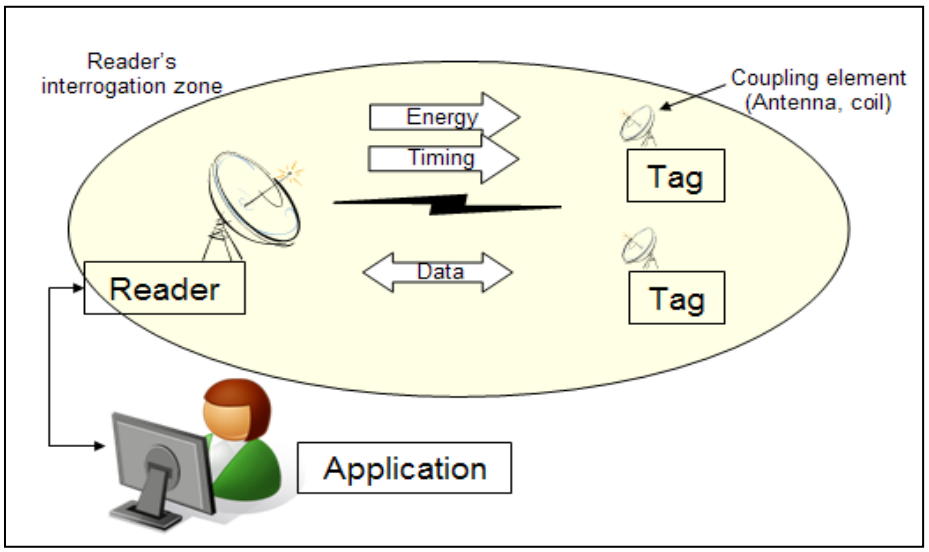

Figure 1: RFID Physical Composition [Finkenzeller03]

RFID tags can be either passive or active. A passive tag does not require an internal power source, whereas an active tag requires an internal power source such as a battery. In the case of passive tags, the reader broadcasts radio signals which the antenna of each tag within the interrogation zone absorb (also referred to as inductive coupling). Additionally, the reader typically charges tags within its range. After converting the received signal to electrical current, the on-board capacitor stores the electric current. In

case of near-field coupling, this technique is called load modulation and back scattering for far-field coupling. [Bin05]

1.2 Multi-Tag Coordination

Since transmission of tag ids and a reader's capability to receive the transmitted data from multiple tags occur at the same time, tags require a coordination scheme to resolve collisions. Time Division Multiple Access (TDMA) is the most frequently used technique to identify multiple tags simultaneously. In this technique, each tag uses uplink to send data during a read cycle. TDMA for RFID has two protocol specifications [ISO 18000-6:2004].

- ALOHA protocol: This is a tag-driven stochastic TDMA protocol. In this protocol, multiple tags transmit data packets at random intervals. If data packet collision occurs, the tags wait for a random time before resending a data packet.

- Binary tree protocol: This is a reader-driven stochastic TDMA protocol. In this protocol, if data packet collision occurs, the reader resolves each collision one bit at a time with a binary search tree algorithm. Each tag has an ID associated with it. A reader specifies the range of IDs that must reply, while other tag IDs must not respond. If two tags transmit their ID at the same time, which results in a collision, then the reader can detect the exact bit at which the collision occurred. The reader is able to read every tag by using a sophisticated binary search tree algorithm [Hassan06].

1.3 ALOHA Protocol

There are three ALOHA protocols that provide for collision resolution applicable to RFID systems: (Pure) ALOHA, Slotted-ALOHA and Frame-Slotted ALOHA (FSA) [Zürich04]. In Pure ALOHA, a tag responds after a random delay, and continues until identified. In Slotted-ALOHA, a tag responds in synchronized slots after random delay. However, in FSA a tag randomly selects a slot to respond only once in a frame. If there is a collision, tags respond in the next frame. In FSA with variation, the frame size may vary over time. Due to its simplicity, the Framed-Slotted ALOHA collision resolution algorithm is widely implemented in existing protocols [ISO 18000-6:2004, ISO15693-3:2000].

1.3.1 Frame-Slotted ALOHA (FSA)

In Frame-Slotted ALOHA, slots divide the read cycle for an RFID system. Each slot has frames. A slot is a discrete interval of time that allows a tag to transmit its ID number along with the Cyclic Redundancy Check (CRC) code. The reader synchronizes tags timers to ensure no collisions are partial. A read cycle is the time interval between two REQUEST commands and is repeated until identification of all tags in the interrogation range is completed. Figure 2 shows the beginning of a census for four tags, with three slots per frame. When the frame size is fixed, each transmitted frame has the same number of slots. Initially, RESET and CALIBRATION commands sent by the reader activate the passive tags. Then the reader broadcasts a REQUEST command. On receiving a REQUEST command tags synchronize their timers and then transmit their ID along with the CRC code. [Prodanoff10]

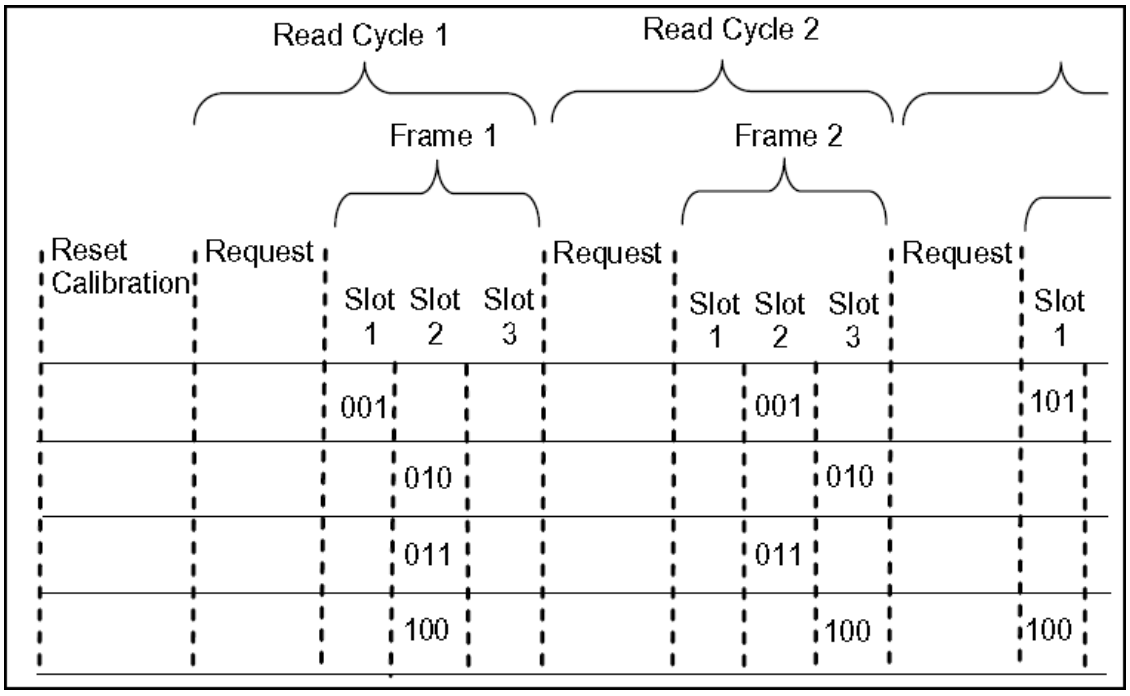

Figure 2: Frame-Slotted ALOHA Algorithm [Prodanoff10]

FSA is classified as Basic FSA (BFSA) and Dynamic FSA (DFSA) based on whether it uses a fixed or variable frame size. BFSA and DFSA are further classified based on whether they support muting and early end. In the muting feature, the reader has to silence successfully identified tags. In the early end feature, the reader closes the idle slots or the no response slots early. [Klair07]

Chapter 2

PREVIOUS WORK

2.1 FSA RFID Anti-Collision Protocol Study

The experiment written about in [Klair07] studied the energy consumption of Frame-Slotted ALOHA (FSA) based anti-collision protocols to determine an appropriate anti-collision protocol suitable for enhanced RFID wireless sensor networks. The experimental setup involved a reader with design features of SkyeTek's M1-Mini RFID reader to model the RFID system with *n* tags in an interrogation zone. The tags transmitted data at a rate of 26 kilo bytes per second (kbps) [Klair07]. In order to evaluate the energy consumption of FSA protocols, the experiment analyzed the success delay, the collision delay, and the idle listening delay.

2.1.1 Delays Incurred by BFSA Non-Muting

Equation 1 calculates the probability of *r* tags responding in a slot in the i^{th} read cycle, where N is the given frame size (slots) and *n* is the number of tags to be read in the i^{th} read cycle [Bin05] .

$$p_r(i) = \binom{n}{r}\left(\frac{1}{N}\right)^r\left(1-\frac{1}{N}\right)^{n-r} \qquad (1)$$

From equation 1, the probability of having one or more idle slots in the i^{th} read cycle ($p_0(i)$) is calculated in equation 2, the probability of successful slots in the i^{th} read cycle ($p_1(i)$) is

calculated in equation 3, and the probability of collide slots in the i^{th} read cycle ($p_k(i)$) is calculated in equation 4.

$$p_0(i) = \left(1 - \frac{1}{N}\right)^n \tag{2}$$

$$p_1(i) = \frac{n}{N}\left(1 - \frac{1}{N}\right)^{n-1} \tag{3}$$

$$p_k(i) = 1 - p_0(i) - p_1(i) \tag{4}$$

Then the expected number of the successful transmissions in the i^{th} read cycle becomes $Np_1(i)$, since a read cycle has N slots [Bin05]. Equation 5 calculates the probability of having an unread tag after R reads [Klair07].

$$p_{miss}(i) = \prod_{i=1}^{R}\left(1 - \frac{Np_1(i)}{n}\right) = 1 - \alpha \tag{5}$$

R represents the number of required read cycles to identify a set of tags with a confidence level α. Because the number of tags n and the frame size N are the same for all read cycles, $p_1(i)$ remains constant. As a result, the equation 5 transforms into equation 6.

$$\left(1 - \frac{Np_1}{n}\right)^R = 1 - \alpha \tag{6}$$

Solving equation 6 for R yields [Klair07]

$$R \geq \left| \frac{\log(1-\alpha)}{\log\left(1-\frac{Np_1}{n}\right)} \right| \quad (7)$$

The ceiling function represents the integral value of *R*. Using *R,* the theoretical delay of success (D_{Succ_BFSA}), idle (D_{Idle_BFSA}), and collision (D_{Coll_BFSA}) can be obtained from equations 8-10. In equations 8-10, *N* is a frame size and *T* is slot duration [Klair07].

$$D_{Succ\_BFSA} = NRT \quad (8)$$

$$D_{Idle\_BFSA} = Np_0RT \quad (9)$$

$$D_{Coll\_BFSA} = NRT(1-p_0-p_1) \quad (10)$$

The summation of these three delays is the total census delay.

2.1.2 Delays Incurred by BFSA Muting

During muting, after the reader successfully identifies the tags, these identified tags will not respond to the reader. As a result, the number of responses reduces after each identification cycle. Equation 11 calculates the number of tags in the $(i+1)^{th}$ round [Bin05].

$$n(i+1) = n(i) - p_1(i) \times N(i) \quad (11)$$

In equation 11, $p_1(i) \times N(i)$ is the number of tags identified in a read round. Based on equation 11, the algorithm shown in Figure 3 calculates the total delay, collision delay, and idle delay [Klair07].

```
BEGIN ;
  Initialize unread tags  = actual number of tags ;
  While True do
        Perform a read cycle for unread tags;
        Store the number of identified tags;
        Store the number of slots filled with collisions;
        Store the number of slots filled with idle responses;
        Store current frame size;
         IF (No Collisions) then
                Break;
         Else
                Unread tags = actual – identified tags;
         end
  end
  Total delay = T X ∑ stored frames;
  Collision Delay = T X ∑ stored collision slots;
  Idle Delay = T X ∑ stored Idle slots;
END;
```

Figure 3: Pseudo-Code for BFSA Muting

2.1.3 Delays Incurred by BFSA Non-Muting Early End

If the reader does not detect any response from the tags, then the reader closes the slot in BFSA Non-Muting early end. Time *t* denotes the time after which the reader closes a slot and N_{Idle_early} denotes the number of no response slots. Equation 13 and 14 calculate the success delay (D_{Succ_early}) after substituting equation 8 in equation 12, and the idle delay (D_{Idle_early}) for BFSA early end [Klair07]. The collision delay remains unchanged in BFSA-non muting early end, as the probability of collision is not dependent on slot duration.

$$D_{Succ\_early} = D_{Succ\_BFSA} - (T - t)N_{Idle\_early} \quad (12)$$

$$D_{Succ\_early} = NR(T - (T - t)p_0) \quad (13)$$

$$D_{Idle\_early} = Np_0Rt \quad (14)$$

2.2 BFSA Muting Protocol Simulation Using OPNET

A model simulation of the BFSA Muting Protocol with OPNET IT Guru 14.0 [Kang08] measures the total census delay and network throughput. A comparison of the model simulation and analytical results from the algorithm by Klair [Klair07] show that the results agree. Figure 4 shows the minimum total census delay for the simulation and the minimum total census delay for the algorithm by Klair [Klair07] both delays increase linearly with the number of tags.

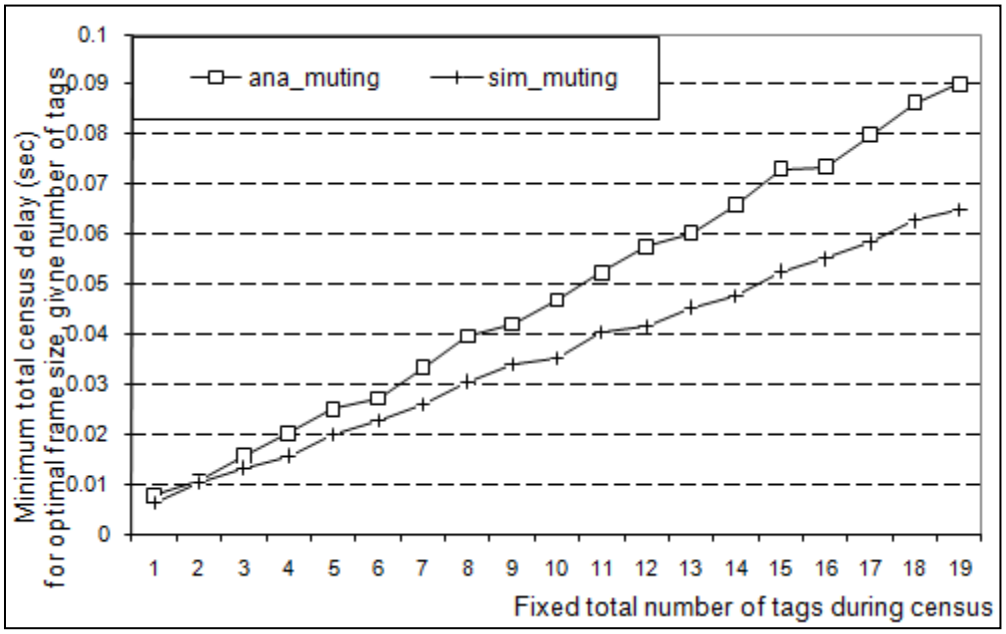

Figure 4: Total Census Delay for BFSA Muting [Kang08]

Figure 5 shows the network throughput comparison for the BFSA muting simulation results and the BFSA muting analytical results. The analytical network throughput and the simulation network throughput decrease with the increase in the number of tags.

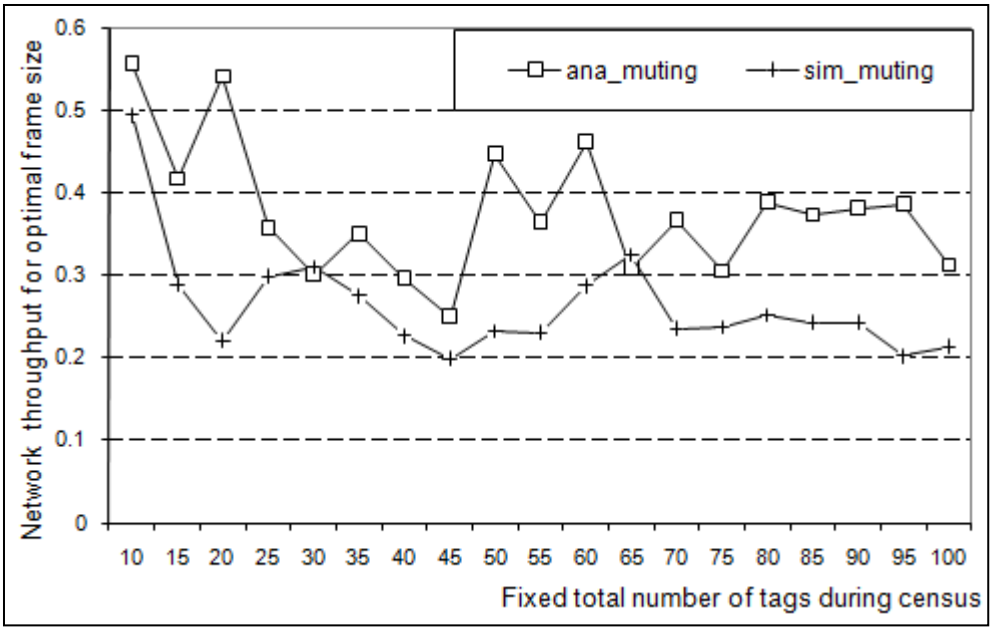

Figure 5: Network Throughput for BFSA Muting [Kang08]

Chapter 3

FORMAL ANALYSIS

3.1 RFID System and Non-Unique Tags

Ports and airport terminals integrate RFID technology to increase the efficiency and effectiveness of container systems. Ports mainly use RFID technology for container tracking, wherein the reader can read tags from multiple vendors [Mullen05]. At any port/terminal, hundreds of containers arrive from around the globe with RFID tags manufactured by various vendors located around the world. It is quite possible that RFID tags manufactured by various vendors have the same tag ID, because ID code data sets of the vendors are not synchronized or may be subject to tag errors during production. In such situations, it is legitimate to consider all of the tags of different vendors.

Existing implementations of RFID systems mostly use unique tags; hence, most multi-tag anti-collision protocols do not support non-unique tag identification. For instance, Philips I-Code1 cannot identify RFID tags if non-unique tags are present, as non-unique tags will always occupy the same time slot in every read cycle [I-CODE1]. However, ISO/IEC the 18000-6 protocol can be used to generate different time slot even for non-unique tag IDs [ISO 18000-6:2004].

3.1.1 Time Slot Calculation

Philips I-Code1 calculates the time slot based on the tag ID itself, as a result of which non-unique tags will always generate the same time slot [I-CODE1]. ISO 18000-6 has the following two options to generate the time slot for a tag ID: pseudo-random number generation; sub unique ID calculation. The main purpose of pseudo-random number generation in ISO 18000-6 is to increase the probability of detecting a collision between two or more tags whose responses contain the same data. The pseudo-random number detects a collision when a 4-bit tag signature combines with an 8-bit random number to generate a unique 12-bit random number. A 4-bit pseudo-random number generator can use a part of the tag's UID, the tag's CRC, or a circuit that measures the tag's excitation level. The vendor decides the appropriate way of generating the tag signature. The generation of the tag signature and the random number are independent of each other. Hence, the ISO 18000-6 protocol can be used for identifying non-unique tags without any modification. [ISO 18000-6:2004].

3.2 Extending Algorithms

A review of readily available literature shows a lack of research on RFID systems with non-unique RFID tag identification. Hence, it is important to study the effect of non-unique tags on the performance of the system and the effect on the total census delay. This thesis analyzes BFSA muting for non-unique tags by extending the algorithms proposed in [Klair07]. By extending the algorithms for RFID systems with non-unique tags, it is possible to determine total delay, collision delay, idle delay, and the total time to identify all tags in the system. An RFID system with the BFSA non-muting technique is not practical, as the

reader cannot determine whether it is the same tag that is responding or there exists another tag with the same tag ID.

Figure 6 shows pseudo-code for the extended algorithm to determine the total time required to find non-unique tags using the BFSA muting technique. Time *T* corresponds to the duration of a slot and is defined by the rate of data transfer of the tags ID, in bits per second.

```
BEGIN;
Initialize unread tags  = actual number of tags ;
Initialize non-unique tags  = 0;
While True do
        Perform a read cycle for unread tags;
        Store the number of identified tags;
       Store the number of slots filled with collisions;
       Store the number of slots filled with idle responses;
       Store current frame size;
          If( non-unique tags are identified) then
                Non-unique tags = Determine the non-unique tags identified in read
cycle;
                Store the Non-unique Tags;
           end
        IF (No Collisions) then
                Break;
        Else
                Unread tags = actual – identified tags;
        end
end
Total delay = T X ∑ stored frames;
Collision Delay = T X ∑ stored collision slots;
Idle Delay = T X ∑ stored Idle slots;
Total time for non-unique tags = T X ∑ Non-unique tags;
END;
```

Figure 6: Pseudo-Code for RFID System with Non-Unique Tags in BFSA Muting

Figure 7 shows the pseudo-code for the extended algorithm to determine the total time required to find non-unique tags in the BFSA muting early-end technique. If the reader does

not detect responses from tags, time *t* corresponds to the duration of time after which the reader closes a slot ($t < T$).

```
 BEGIN;
Initialize unread tags  = actual number of tags ;
Initialize non-unique tags  = 0;
While True do
        Perform a read cycle for unread tags;
        Store the number of identified tags;
       Store the number of slots filled with collisions;
       Store the number of slots filled with idle responses;
       if slots are idle for time t then
          close the slot;
          update the frame size;
      end
       Store current frame size;
          If( non-unique tags are identified) then
                Non-unique tags = Determine the non-unique tags identified in read cycle;
             Store the Non-unique Tags
          end
        IF (No Collisions) then
                Break
        Else
                Unread tags = actual – identified tags;
        end
end
Total delay = T X ∑ stored frames;
Collision Delay = T X ∑ stored collision slots;
Idle Delay = T X ∑ stored Idle slots;
Total time for non-unique tags = T X ∑ Non-unique tags;
END;
```

Figure 7: Pseudo-Code for RFID System with Non-Unique Tags in BFSA Muting Early End

Figure 8 shows the pseudo-code for an algorithm to determine the time when the count of each tag *d* matches exactly with non-unique tags of a single ID in the range of the reader.

```
BEGIN
Initialize unread tags = actual number of tags
While True do
        Perform a read cycle for unread tags
        Store the number of identified tags
        If( non-unique tags are identified) then
                For each non-unique tag calculate the count
                    Insert/Update details for that tag
                        Store the count of non-unique tag
                        Store Time taken for that tag
                End for
        End
        IF (No Collisions) then
                Break
        Else
                Unread tags = actual – identified tags
        end
Print the tag details, which include the count of that tag and time taken to determine
it.
END
```

Figure 8: Pseudo-Code to Determined Non-Unique Tags of a Single Tag in BFSA Muting

The time complexity for the above three algorithms depends on the number of read cycles each one takes to read all *n* tags. For instance, say *R* is the duration of a single read cycle and *m* is the number of cycles required to read *n* tags, then the time complexity for first algorithm (Figure 6) is the total number of read cycles *O(mR)*. For the same RFID system, R will be a constant, but m will vary for the same number of RFID tags or a different number of RFID tags. For the second algorithm (Figure 7), it will take *(m-t)* read cycles to read. Therefore, the time complexity for the second algorithm is *O((m-t)R)*. For the last algorithm (Figure 8), additional time is required to calculate the count for each non-unique tag. Therefore, the time complexity of the last algorithm is *O(mR+ n)*.

Chapter 4

METHODOLOGY

4.1 Evaluating BFSA Muting Model with Non-Unique Tags

The analytical model built as part of this thesis with OPNET Modeler 14.5 analyzed the Basic Frame-Slotted ALOHA protocol supporting non-unique tags of a passive RFID network system. This model is a partial implementation of a real-world RFID system, as we assume slot duration and frame size are constant and no tags or readers ever fail. We also assume that there are no signal anomalies caused by the environment, such as attenuation and distortion. The sample size for the BFSA muting RFID system varied from 10 to 100 tags.

The model simulated the behavior of non-unique tags, by implementing the BFSA protocol to predict the total census delay, network throughput, and optimal frame-size. The simulation results from this study were compared with a previous study conducted by [Kang08] for unique tags.

4.1.1 Evaluating Delays

The total census delay consisted of success delay, collision delay, and idle delay [Cappelletti06].

Equation 15 calculated the slot delay duration *T* in seconds (sec), where *ID* (bits) was the size of the packet containing the tag's *ID* and *data_rate* (bps) was the rate of data transfer from the tag to the reader.

$$T = \frac{ID\ (bits)}{data\_rate\ (bps)} \tag{15}$$

Equation 16 calculated the total census delay [Klair07], where *T* was slot duration defined by response packet size by data rate and $\sum stored\_frames$ was the sum of all frame sizes.

$$Total\ delay = T \times \sum stored\_frames \tag{16}$$

4.1.2 Evaluating Network Throughput

The ratio of the number of successfully transmitted packets (one per tag) to the total number of packets sent by tags during the census defines network throughput [Cappelletti06]. The model measured the number of required read cycles and the number of unread tags. These parameters determined the total number of packets (tags) transmitted during the census. Equation 17 calculated network throughput, where *S[n]* was network throughput, *n* was total number of identified tags, α was the assurance level, and *P[n]* was the total number of packets sent by tags during the census.

$$S[n] = \frac{\alpha\ n}{P[n]} \tag{17}$$

The assurance level is the probability of identifying all tags in the reader's interrogation range. [Vogt02]. A value of 0.99 for α in the model indicates less than 1% of the tags are missing.

Chapter 5

VERIFICATION OF THE MODEL

5.1 Simulation Conditions

For simplicity, consider a reader and eight tags with non-unique tags as shown in Figure 9.

Table 1 lists the values of the parameters used to simulate the model.

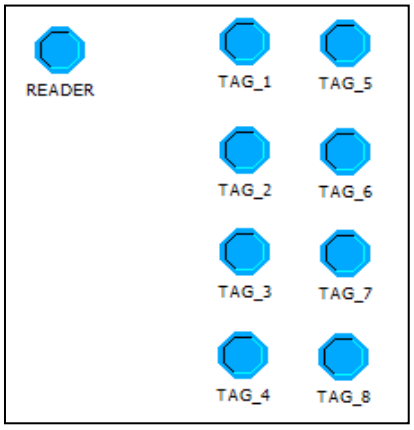

Figure 9 : A Reader and Tags

List	Value
Assurance Level	0.99
Data rate (between reader and tags)	500000(bps)
Frame size (number of slots in a read cycle)	8
Duration of single slot	0.00016 sec
Duration of read cycle	0.00128 sec
REQUEST packet size (from reader to tag)	88 bits
SELECT packet size (from reader to tag)	72 bits
RESPONSE packet size (from tag to reader)	80 bits

Table 1: Simulation Condition

The time required for packet transmission calculated using a given packet size and data rate is shown in Table 2.

List	Value
REQUEST packet	0.000176 sec
SELECT packet	0.000144 sec
RESPONSE packet	0.00016 sec

Table 2: Packet Transmission Time

5.1.1 BFSA Muting with Non-Unique Tags

At the beginning of the census, the number of unread tags was initialized to the actual number of tags in the range. On performing the census to identify the unread tags, the number of identified tags, the slots with collision, the idle slots, the current frame size, and the number of non-unique tags if any were stored. If there were no collisions, then the total delay, collision delay, and idle delay were calculated. Figure 6 shows the pseudo code for the RFID system with non-unique tags in BFSA muting. Appendix A documents the log from the BFSA muting simulation with non-unique tags. Figures 10-14 show the sequence of events during the BFSA muting for non-unique tags.

As shown in Figure 10, a frame consisted of eight slots. At the beginning of the census, the reader broadcast the REQUEST packet to all the tags. As seen from Table 1, the transmission delay for a REQUEST packet is 0.000176. Assume the propagation delay was negligible. On receiving the REQUEST packet, the tags synchronized their timers to avoid partial collision. Tags were allowed to transmit only once per read cycle, as per the FSA protocol. Tags randomly selected one of the slots to transmit a RESPONSE packet. Each tag

had an ID, which could be non-unique to one or more tags, and the unique ID was a calculated number unique for each tag. When multiple tags transmitted their ID's to the reader within the same slot, a collision occurred and the reader could not identify any tags.

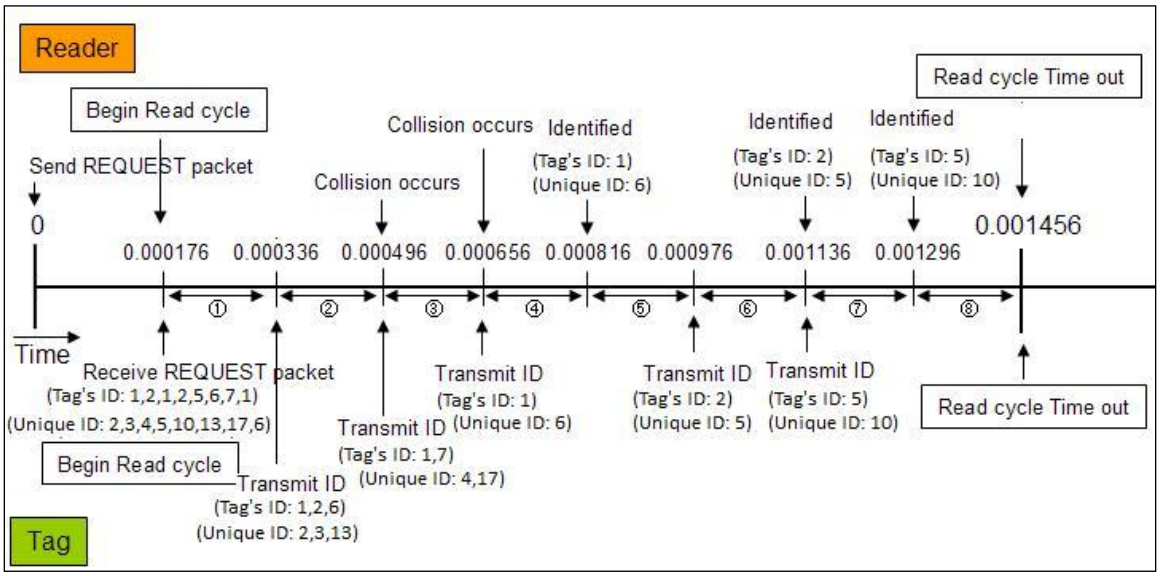

Figure 10: First Read Cycle

Figure 10 shows two collisions occurred in the first read cycle. Three tags with IDs: 1, 2, and 6 (corresponding Unique IDs: 2, 3, and 13) were transmitted by occupying the second slot. Also tags with IDs: 1 and 7 (corresponding Unique IDs: 4, and 17) were transmitted by occupying the third slot. Because of the collision, the second slot and third slot were rejected. As can be seen in the fourth, sixth, and seventh slots, the reader successfully identified a single tag transmission without a collision. The first, fifth, and eighth slots were idle slots in the first read cycle.

After the end of a read cycle, the reader computed and stored the number of identified tags, number of non-unique tags, slots with collisions, and idle slots. If there were no collisions in the read cycle, then the reader completed a census. For the identified tags,

the reader transmitted SELECT packets with unique IDs as shown in Figure 11. As soon as a tag with a matching unique ID received the SELECT packet, the reader stopped the tag from transmitting the ID by muting the tag. The reader transmitted the SELECT packets to three identified tags from the previous read cycle, as shown in Figure 11. Then the reader broadcast the REQUEST packet to tags. Only, unread tags will respond to the REQUEST packet.

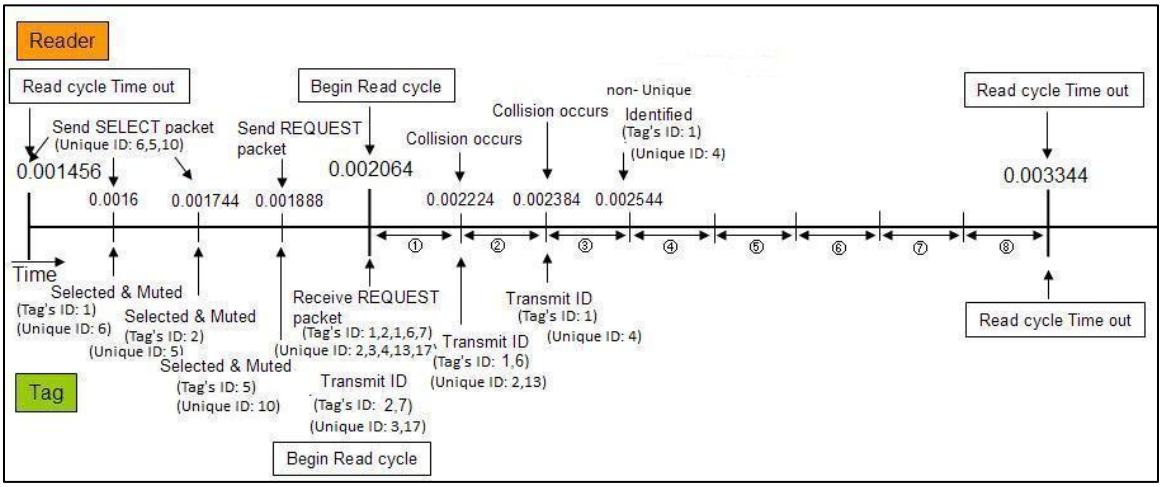

Figure 11: Second Read Cycle

Figure 11 shows two collisions occurred in the second read cycle. Two tags with IDs: 2 and 7 (corresponding Unique IDs: 3 and 17) were transmitted by occupying the first slot and two tags with IDs: 1 and 6 (corresponding Unique ID: 2 and 13) were transmitted by occupying the second slot. Because of the collision, first slot and second slot were rejected. In the third slot, the reader successfully identified a single tag transmission without a collision. As the tag identified in the third slot had a non-unique tag ID to a tag previously identified, the reader updated the non-unique tags count.

In Figures 12-14, on delivery of a REQUEST packet to the tags the read cycle began. In every read cycle, the procedure for transmitting tag ID's, the detection of a collision, and the identification of a non-unique or unique tag was the same as in the previous read cycle. Figure 14 shows the fifth read cycle where no collisions occurred so the census finished.

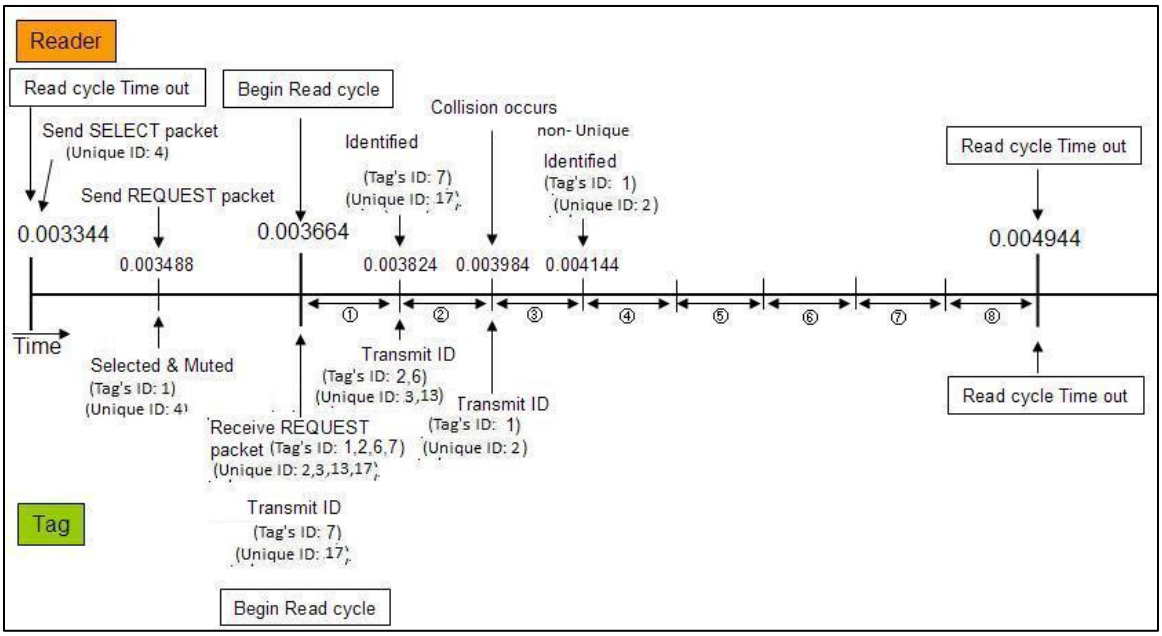

Figure 12: Third Read Cycle

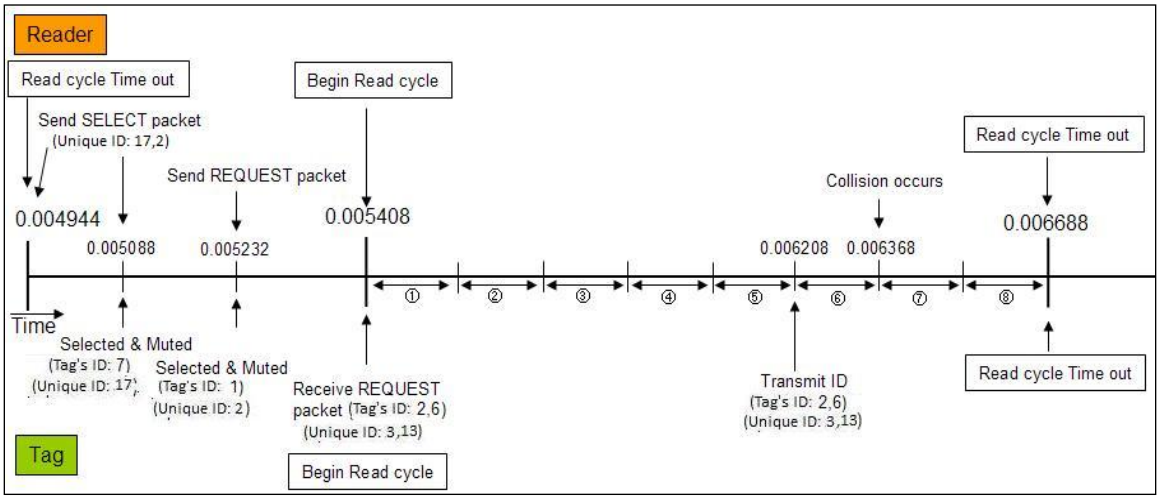

Figure 13: Fourth Read Cycle

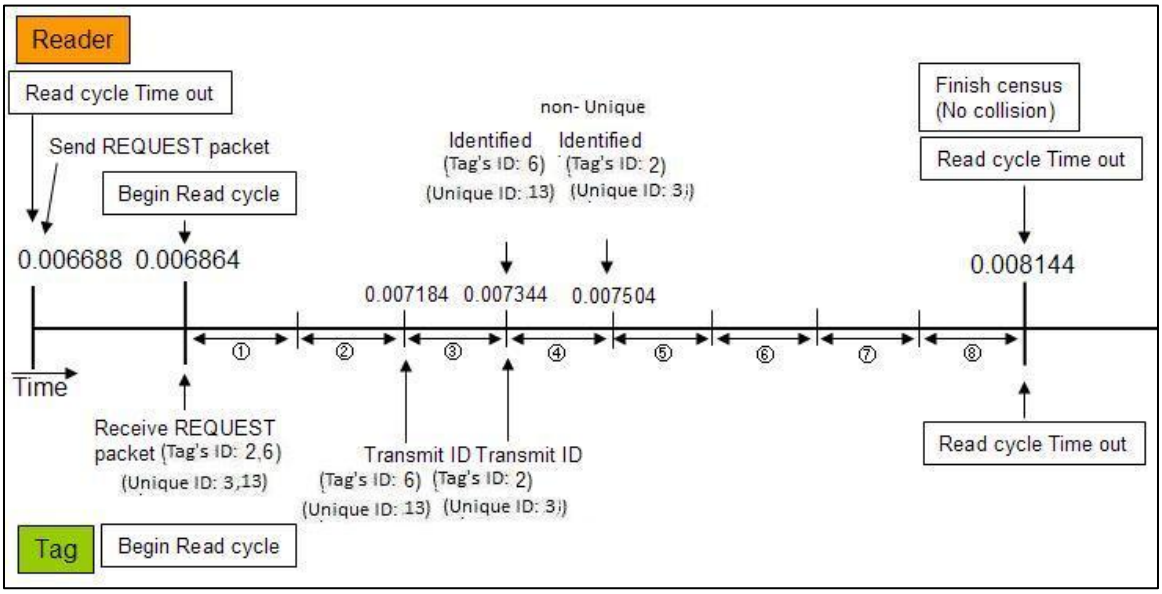

Figure 14: Fifth Read Cycle

Chapter 6

EVALUATION

This study compared simulation results with the results from a previous study for unique tags conducted by [Kang08] using statistical tests. We performed the evaluation of total census delay and other metrics of interest under the premise that the scope of this work was theoretical, assuming ideal conditions. By *Ideal conditions* we imply a constant frame size and slot duration for the total census and no failing tags or readers with no signal anomalies caused by the environment, such as attenuation and distortion. There is no closed form expression for the computations associated in the theoretical solution in this section. We arrived at the results iteratively, by using the algorithms presented in Chapter 3.

6.1 Total Census Delay

The main purpose of this evaluation was to compute census delay under ideal conditions for the case of BFSA muting. The total census delay depended on the frame size and the actual number of tags. The total census delay would be longer, if the frame size was larger or smaller. If the frame size was larger, there would be too many idle slots resulting in a longer total census delay. Similarly, if the frame size was too small, there would be too many collisions resulting in a longer total census delay. The calculation of minimum total census delay required an optimal frame size obtained from the results. In order to determine the optimal frame size for a given number of tags, the frame size

varied from 10 to 120, in increments of five, for ten censuses each [Kang08]. The model executed for tag sets from 10 to 100, using an increment of five.

Since variability in results can have a great effect on network performance, we compared the variability between the two populations of non-unique and unique tags in all tests described in this chapter. We first performed a two-sample *F*-test, a statistical test for variance that compares the variance between the minimum total census delay for non-unique tags and unique tags. We chose to perform a one-sided *p*–value test, since the expectation was that the non-unique tags variation in results would have a larger mean. We assumed two independent random samples and Gaussian populations (normally distributed). The reader should note that if this assumption of normality does not hold, it may be necessary to perform other statistical tests. We anticipate that more realistic performance evaluation studies that relax some of the described assumptions (e.g. no signal interference) may require other tests to demonstrate the statistical significance of the results.

Figure 15 shows the minimum total census delay increased linearly with the number of tags for both BFSA muting with non-unique tags and BFSA muting tags with unique tags. The total census delay for BFSA muting for 100 non-unique tags was 0.056 seconds, while the total census delay for BFSA muting for 100 unique tags was 0.0648 seconds. This shows that BFSA muting with non-unique tags had a slightly lower total delay than BFSA muting for unique tags.

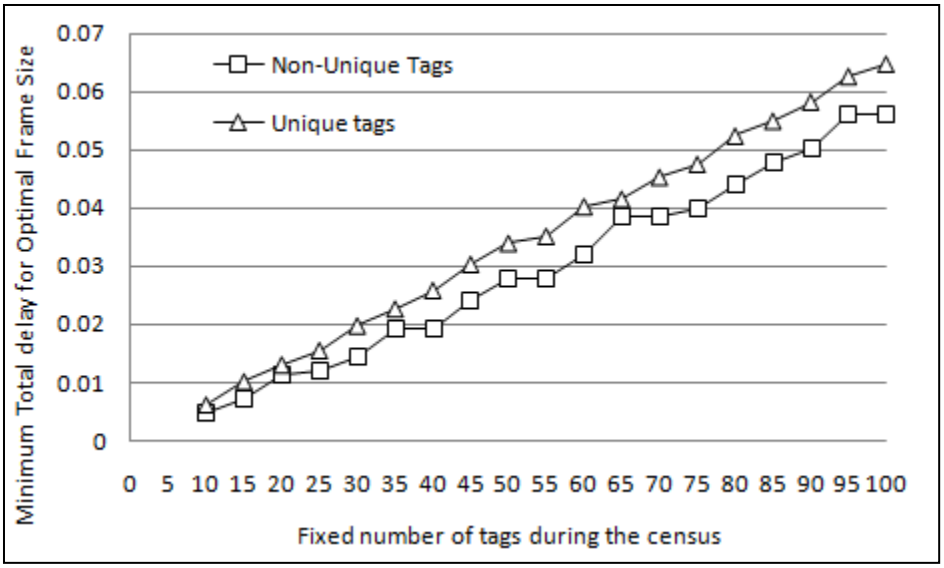

Figure 15: Minimum Total Census Delay for BFSA Muting

In this and in all other tests described in this chapter, the null hypothesis states that the two population variances are equal and the alternative hypothesis is that non-unique tags have greater variance in results. Assume the level of significance α as 0.05. The *F*-test calculates the *F*-statistic, *F*-critical value, and *p* value. According to the *F*-test results, when *F*-statistic < *F*-critical and $p > \alpha$, then the variances are equal for two samples.

Table 3 shows the *F*-test for variance results for minimum total census delay for non-unique tags and unique tags. Since *F*-statistic > *F*-critical (0.81279457 >0.45101989), using a lower tail critical value, and $p > \alpha$ (0.33241392 >0.05), we failed to reject the null hypothesis. With a *p*-value of 0.33, we cannot reject the assumption that the variance in a population of unique tags is equal to the variance in a population of non-unique tags.

	Non-Unique	*Unique*
Mean	0.03006316	0.035869
Variance	0.00027145	0.000334
Observations	19	19
df	18	18
F	0.81279457	
P(F<=f) one-tail	0.33241392	
F Critical one-tail	0.45101989	

Table 3: F-Test for Total Minimum Census Delay for Non-Unique and Unique Tags

Figure 16 shows a comparison of optimal frame size for BFSA muting with non-unique tags and BFSA muting with unique tags. Figure 16 shows that the optimal frame size increased approximately linearly with the number of tags, irrespective of the non-unique or unique type of protocol.

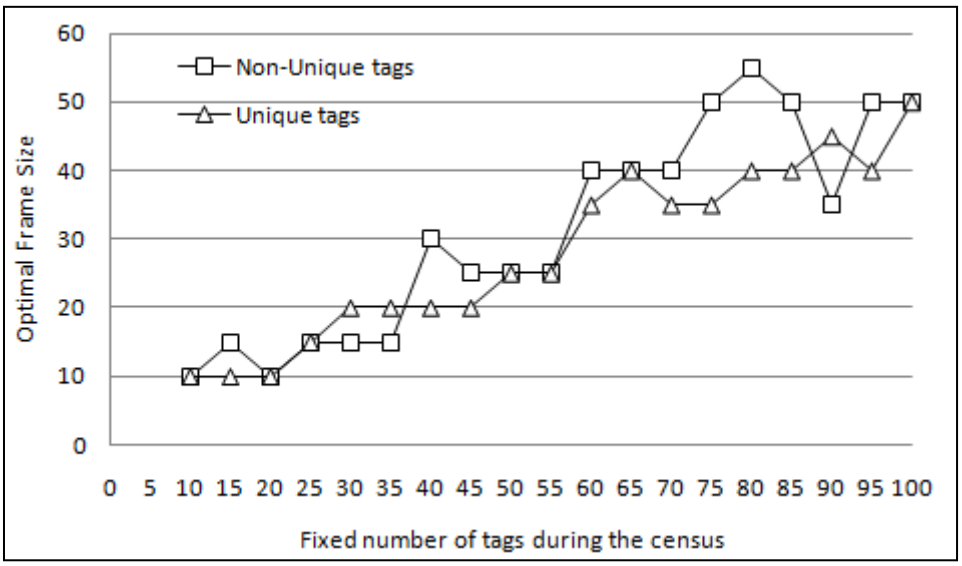

Figure 16: Optimal Frame Size for BFSA Muting

Table 4 shows the *F*-test for variance results for optimal frame size for non-unique tags and unique tags. Since *F*-statistic < *F*-critical (1.47468354 < 2.21719713) and $p > \alpha$ (0.20892521 > 0.05), we failed to reject the null hypothesis. With a *p*-value of about

0.21, we cannot reject the assumption that the variance in optimal frame size in a population of unique tags is equal to the variance in a population of the non-unique tags.

	Non-Unique	*Unique*
Mean	31.3157895	28.15789
Variance	238.450292	161.6959
Observations	19	19
df	18	18
F	1.47468354	
P(F<=f) one-tail	0.20892521	
F Critical one-tail	2.21719713	

Table 4: F- Test for Optimal Frame Size for Non-Unique and Unique Tags

6.2 Network Throughput

This section describes the evaluation of network throughput when the frame size is optimal, maximum throughput, minimum throughput, and average throughput. Figure 17 shows network throughput when the frame size is optimal for BFSA muting with non-unique tags and BFSA with unique tags. Figure 17 shows that network throughput decreases with increase in number of tags for non-unique and unique versions of the protocol.

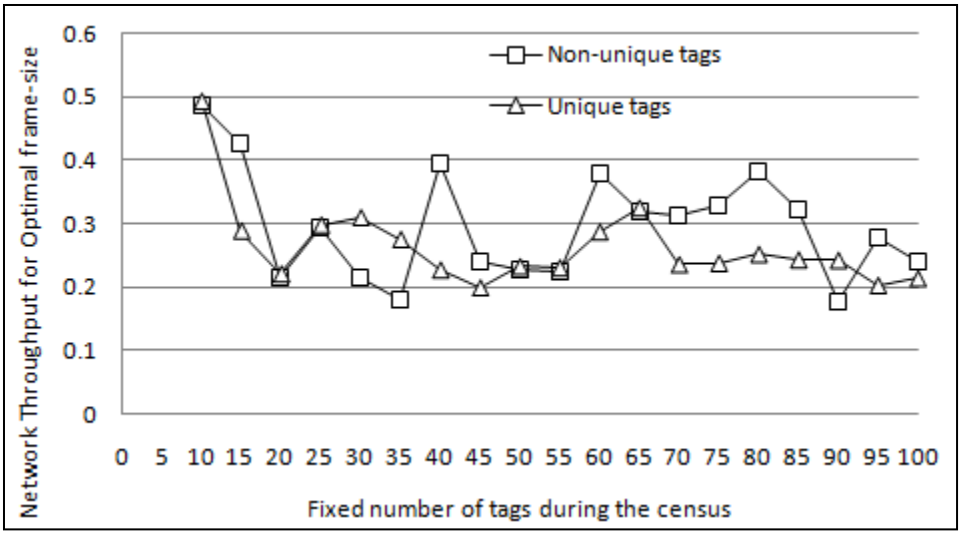

Figure 17: Network Throughput When Frame Size is Optimal for BFSA Muting

Table 5 shows the *F*-test for variance results for network throughput when frame size is optimal for unique and non-unique tags. Since *F*-statistic < *F*-critical (1.71574161 < 2.21719713) and $p > \alpha$ (0.13077254 > 0.05), we failed to reject the null hypothesis. With a *p*-value of 0.13, we cannot reject the assumption that the variance in throughput when frame size is optimal in a population of unique tags is equal to the variance in a population of non-unique tags.

	Non-Unique	*Unique*
Mean	0.29680101	0.263573
Variance	0.00757878	0.004417
Observations	19	19
df	18	18
F	1.71574161	
P(F<=f) one-tail	0.13077254	
F Critical one-tail	2.21719713	

Table 5: F-Test for Network Throughput for Non-Unique and Unique Tags

Network throughput depends on the total number of read cycles and the number of tags transmitted. When the frame size varies for a fixed number of tags, the number of read cycles required to complete the census varies, which results in fluctuation of network throughput. Hence, there is a need to evaluate the minimum throughput, the mean throughput, and the maximum throughput. Figures 18-20 show minimum throughput, mean throughput, and maximum throughput for BFSA muting with non-unique tags and BFSA muting with unique tags with frame size varying from 10 to 120. Figures 18-20 show throughput gradually decreases with an increase in the number of tags for BFSA muting with non-unique tags, while in the case of unique tags there is a rapid decrease in throughput with an increase in the number of tags.

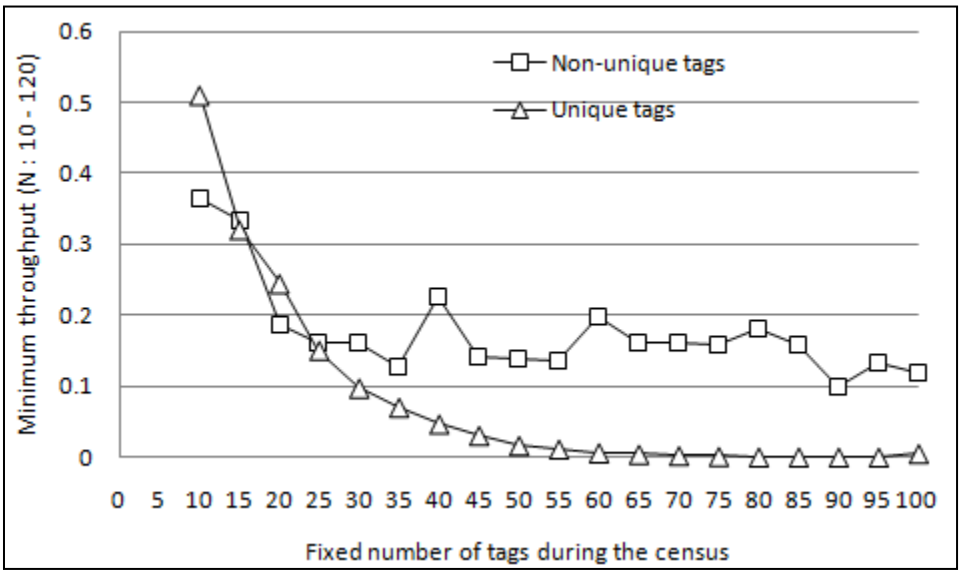

Figure 18: Minimum Network Throughput for BFSA Muting

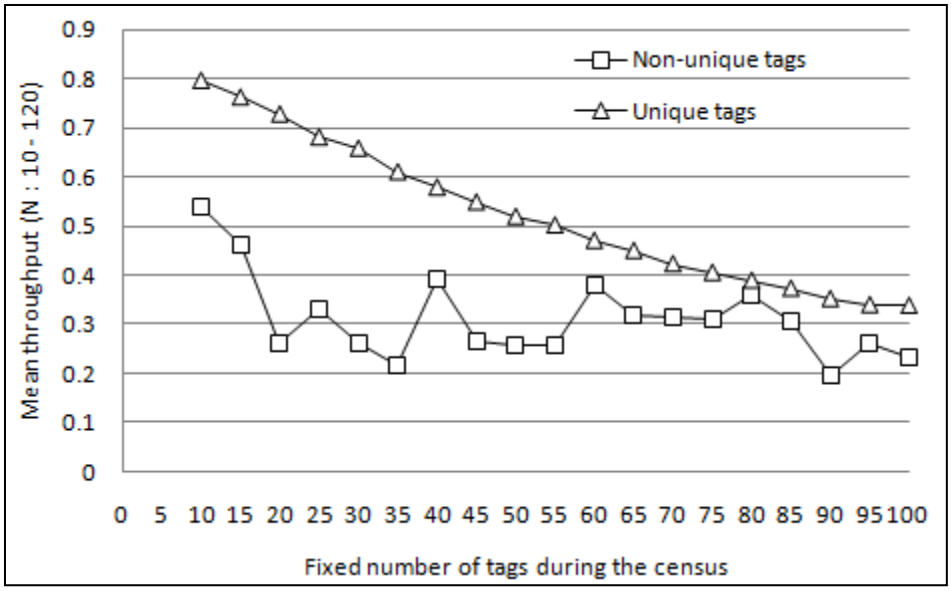

Figure 19: Mean Network Throughput for BFSA Muting

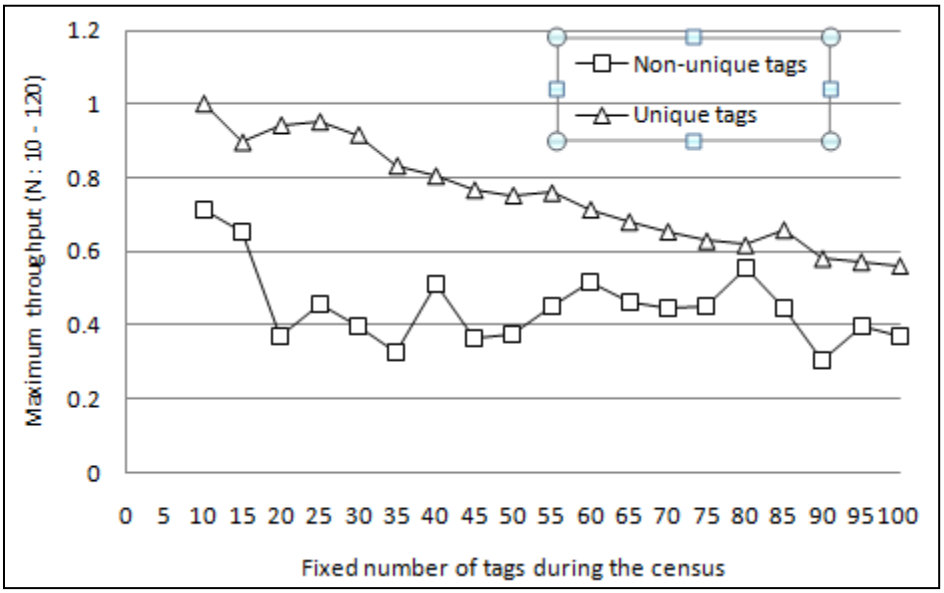

Figure 20: Maximum Network Throughput for BFSA Muting

Table 6 shows the F-test for variance results for minimum network throughput for non-unique tags and unique tags. Since F-statistic < F-critical (0.24034048 < 0.45101989) and $p < \alpha$ (0.00204818 < 0.05) the variance of non-unique tags differs from unique tags for minimum network throughput, we can reject the null hypothesis. The two

populations show a statistically significant difference in the variance in minimum network throughput.

	Non-unique	*Unique*
Mean	0.17569229	0.080122
Variance	0.00457054	0.019017
Observations	19	19
Df	18	18
F	0.24034048	
P(F<=f) one-tail	0.00204818	
F Critical one-tail	0.45101989	

Table 6: F-Test for Minimum Network Throughput for Non-Unique and Unique Tags

Table 7 shows the *F*-test for variance results for mean network throughput for non-unique tags and unique tags. Since *F*-statistic < *F*-critical (0.33422389 < 0.45101989) and $p < \alpha$ (0.01255299 < 0.05) we can reject the null hypothesis. The two populations show a statistically significant difference in the variance in mean network throughput.

	Non-Unique	*Unique*
Mean	0.31175584	0.521476
Variance	0.00741211	0.022177
Observations	19	19
df	18	18
F	0.33422389	
P(F<=f) one-tail	0.01255299	
F Critical one-tail	0.45101989	

Table 7: F-Test for Mean Network Throughput for Non-Unique and Unique Tags

Table 8 shows the *F*-test for variance results for maximum network throughput for non-unique tags and unique tags. Since *F*-statistic > *F*-critical (0.5693935 > 0.45101989) and $p > \alpha$ (0.1208633 > 0.05) we failed to reject the null hypothesis. With a *p*-value of 0.12,

we cannot reject the assumption that the variance in maximum throughput in a population of unique tags is equal to the variance in a population of non-unique tags.

	Non-unique	*Unique*
Mean	0.45136788	0.752328
Variance	0.01102014	0.019354
Observations	19	19
df	18	18
F	0.5693935	
P(F<=f) one-tail	0.1208633	
F Critical one-tail	0.45101989	

Table 8: F-Test for Maximum Network Throughput for Non-Unique and Unique Tags

Chapter 7

CONCLUSION

A model created using OPNET Modeler 14.5 evaluated the performance of the BFSA protocol supporting non-unique tags. The simulation log created by the model verified the accuracy of the model. To study the effect of non-unique tags on RFID systems, we compared the simulation results of the BFSA muting protocol supporting non-unique tags with BFSA muting protocol supporting unique tags.

Comparing minimum network throughput, mean network throughput, and maximum network throughput for unique and non-unique tags for variable frame sizes showed a decrease in network throughput with an increase in the number of tags. The statistical analysis of the results showed a significant difference between the populations of non-unique tags and unique tags in the results obtained for minimum network throughput and mean network throughput.

The minimum network throughput is worse for unique tags than non-unique tags, because the algorithm used to calculate the time slot calculation does not produce optimal results. The mean network throughput and maximum network throughput is worse for non-unique tags than unique tags because the algorithm used to calculate time slot with pseudo random number in case of non-unique tags takes more time than the algorithm for unique tags. The industries that like to use RFID technology supporting non-unique tags would have to sacrifice network throughput for the additional feature of identifying identical tags. Future

work needs to analyze the difference in the results further for total census delay, optimal frame size, and network throughput with optimal frame size, as *F*-test results were undeterminable.

REFERENCES

Print Publications:
[Finkenzeller03]
Finkenzeller, K., "RFID Handbook," 2nd Edition, John Wiley & Sons, 2003.

[ISO15693-3:2000]
Identification Cards - Contactless Integrated Circuit Cards - Vicinity Cards – Part 3: Anticollision and Transmission Protocol, ISO/IEC 15693-3, 2000-03-10.

[ISO 18000-6:2004]
Information technology – Radio Frequency Identification for Item Management – Part 6: Parameters for Air Interface Communications at 860 MHz to 960 MHz, ISO/IEC 18000-6, 2004-08-15.

[Kang08]
Kang, S., "Framed Slotted ALOHA: Multi-Tag Identifying Anti-Collision RFID Protocol Simulation Using OPNET Modeler 14," Thesis (M.S. in Computer and Information Sciences), University of North Florida, 2008.

[Zürich04]
Zürich, E., Burdet, L. A., "RFID Multiple Access Methods," Seminar "Smart Environments," (August 15, 2004).

Electronic Sources:
[AZAMBUJA08]
Azambuja, M. C., Marcon, C., and Hessel, F. P. "Survey of Standardized ISO 18000-6 RFID Anti-Collision Protocols," The Second International Conference on Sensor Technologies and Applications, 2008 IEEE Conference Publishing Services, 2008, ieeexplore.ieee.org.dax.lib.unf.edu/stamp/stamp.jsp?tp=&arnumber=4622705&isnumber=4622621.

[Bin05]
Bin, Z., Mamoru, K., and Masashi, S., "Framed ALOHA for Multiple RFID Objects Identification," IEICE Trans. Comm., Vol.E88–B, No.3 (March 15, 2005.), citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.2078&rep=rep1&type=pdf.

[Cappelletti06]
Cappelletti, F., Ferrari, G., and Raheli, R. "A Simple Performance Analysis of Multiple Access RFID Networks Based on the Binary Tree Protocol," ISCCSP, (March, 2006), www.eurasip.org/proceedings/ext/isccsp2006/defevent/papers/cr1211.pdf.

[Gerwin]
Gerwin, R., "A Painless Guide to Statistics,"
abacus.bates.edu/~ganderso/biology/resources/statistics.html, (2008-09-26)

[Hassan06]
Hassan, T. and Chatterjee, S. "A Taxonomy for RFID," Proceedings of the 39th Hawaii International Conference on System Sciences – IEEE 2006,
ieeexplore.ieee.org.dax.lib.unf.edu/stamp/stamp.jsp?tp=&arnumber=1579666&isnumber=33368.

[I-CODE1]
Philips Semiconductors – I-Code1 System Design Guide,
www.nxp.com/acrobat_download2/other/identification/SL048611.pdf (2002-05-17).

[Klair07]
Klair, D. K., Chin, K. W. and Raad, R., "On the Suitability of Framed Slotted Aloha based RFID Anti-collision Protocols for Use in RFID-Enhanced WSNs," Computer Communications and Networks, Proceedings of 16th International Conference (August, 2007), pp. 583-590.
ieeexplore.ieee.org.dax.lib.unf.edu/stamp/stamp.jsp?tp=&arnumber=4317881&isnumber=4317770 .

[Mullen05]
Mullen, D. , "The Application of RFID Technology in a Port," AIM Global,
www.aimglobal.org. (2005).

[Natrella]
Natrella, M., "The NIST/SEMATECH e-Handbook of Statistical Methods,"
www.itl.nist.gov/div898/handbook/

[OPNET]
OPNET Technologies, http://www.opnet.com.

[Prodanoff10]
Prodanoff, Z. G. "Optimal Frame Size Analysis for Framed Slotted ALOHA Based RFID networks," Computer Communication, (March. 2010).
portal.acm.org/citation.cfm?id=1716118

[Roberts06]
Roberts, C. M. "Radio Frequency Identification (RFID)", Computers
& Security, Vol. 25, 2006. www.sciencedirect.com/science/article/pii/S016740480500204X

[Vogt02]
Vogt, H., "Multiple Object Identification with Passive RFID Tags," Proceeding of the IEEE International Conference on Systems, Man, and Cybernetics, vol. 3, 6-9 (October 2002).
ieeexplore.ieee.org.dax.lib.unf.edu/stamp/stamp.jsp?tp=&arnumber=1176119&isnumber=26388

[Want06]
Want, R., “An Introduction to RFID Technology,” Published by the IEEE CS and IEEE ComSoc, Pages 25-33, (March 2006), ieeexplore.ieee.org.dax.lib.unf.edu/stamp/stamp.jsp?tp=&arnumber=1593568&isnumber=33539

APPENDIX A
BFSA Muting with Non-Unique Eight Tags Simulation Log

```
|-----------------------------------------------------------------------------------------------|
|  ____  _____  _  _ _______ _______                                                            |
| / __ \| __ \| \ | |__   __|Simulation and Model Library                                        |
|| |  | | |__) |  \| |  | |   Copyright 1986-2008 by                                           |
|| |  | |  ___/| . ` |  | |   OPNET Technologies, Inc.                                         |
|| |__| | |    | |\  |  | |   as a part of OPNET Release 14.5                                  |
|\____/|_|    |_| \_|  |_|                                                                     |
|-----------------------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
|                Making  Networks  and  Applications  Perform                   |
|-------------------------------------------------------------------------------|
|   OPNET Technologies, Inc. / 7255 Woodmont Av. / Bethesda, MD 20814, USA      |
|   WEB: http://www.opnet.com / TEL: +1.240.497.3000 / FAX: +1.240.497.3001
|
|-------------------------------------------------------------------------------|
| Protected by U.S. Patent 6,820,042.
|
|-------------------------------------------------------------------------------|
| Network Simulation of: BFSA_Muting-eight_tags                        |
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Simulation process ID is 1292
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Kernel: optimized, sequential, 32-bit address space                          |
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Opening animation history file.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Reading network model (BFSA_Muting-eight_tags)                     |
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Reading result collection file: (BFSA_Muting-eight_ta                |
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Verifying model consistency.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Rebuilding scenario model library.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Loading scenario model library.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Counting objects.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
```

```
| Allocating objects.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Allocating objects.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Allocating attributes.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Installing subnetworks and nodes.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Installing links.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Constructing model hierarchy.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Creating node contents.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Attaching links to nodes.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Preparing for results collection.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Releasing Model Memory.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Opening results file.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Final initializations for all objects.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Initializing results.
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Progress: Time (0.00 sec.); Events (0)
|
| Speed: Average (0 events/sec.); Current (0 events/sec.)               |
| Time: Elapsed (0.00 sec.)
|
|-------------------------------------------------------------------------------|
|-------------------------------------------------------------------------------|
| Beginning Simulation.
|
|-------------------------------------------------------------------------------|

Data rate (Reader->Tag):  500000.000000 (bps)
Frame size:  8 (slots)
```

```
A slot duration:  0.000160 (sec)
A read cycle duration:  0.001280 (sec)
Assurance level:  0.990000

Tag's ID:  1
Calculated Unique Tag's ID:  2
Data rate (Tag->Reader):  500000.000000 (bps)
Response packet size:  80 (bits)

Tag's ID:  2
Calculated Unique Tag's ID:  3
Data rate (Tag->Reader):  500000.000000 (bps)
Response packet size:  80 (bits)

Tag's ID:  1
Calculated Unique Tag's ID:  4
Data rate (Tag->Reader):  500000.000000 (bps)
Response packet size:  80 (bits)

Tag's ID:  2
Calculated Unique Tag's ID:  5
Data rate (Tag->Reader):  500000.000000 (bps)
Response packet size:  80 (bits)

Tag's ID:  5
Calculated Unique Tag's ID:  10
Data rate (Tag->Reader):  500000.000000 (bps)
Response packet size:  80 (bits)

Tag's ID:  6
Calculated Unique Tag's ID:  13
Data rate (Tag->Reader):  500000.000000 (bps)
Response packet size:  80 (bits)

Tag's ID:  7
Calculated Unique Tag's ID:  17
Data rate (Tag->Reader):  500000.000000 (bps)
Response packet size:  80 (bits)

Tag's ID:  1
Calculated Unique Tag's ID:  6
Data rate (Tag->Reader):  500000.000000 (bps)
Response packet size:  80 (bits)

READER: (TX): Time:  0
2. Sends a command: READ
Number of identified tags:  0 (ea), TIME:  0
READER: (READ_CYCLE_TIME_OUT begin): TIME:  0

TAG (ID: 1 unique ID: 2 ): (READ): Time:  0.000176 , Received frame size:  8
(slots).
TAG (ID: 1 unique ID: 2 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 2 unique ID: 3 ): (READ): Time:  0.000176 , Received frame size:  8
(slots).
TAG (ID: 2 unique ID: 3 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 1 unique ID: 4 ): (READ): Time:  0.000176 , Received frame size:  8
(slots).
TAG (ID: 1 unique ID: 4 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 2 unique ID: 5 ): (READ): Time:  0.000176 , Received frame size:  8
(slots).
TAG (ID: 2 unique ID: 5 ): (READCYCLE_TIMEOUT begin).
```

```
TAG (ID: 5 unique ID: 10 ): (READ): Time:  0.000176 , Received frame size:  8
(slots).
TAG (ID: 5 unique ID: 10 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 6 unique ID: 13 ): (READ): Time:  0.000176 , Received frame size:  8
(slots).
TAG (ID: 6 unique ID: 13 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 7 unique ID: 17 ): (READ): Time:  0.000176 , Received frame size:  8
(slots).
TAG (ID: 7 unique ID: 17 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 1 unique ID: 6 ): (READ): Time:  0.000176 , Received frame size:  8
(slots).
TAG (ID: 1 unique ID: 6 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 1 unique ID: 2 ): (TX): Time:  0.000336
1. Tag's ID ( 1 unique ID: 2 ) is transmitted.

TAG (ID: 2 unique ID: 3 ): (TX): Time:  0.000336
1. Tag's ID ( 2 unique ID: 3 ) is transmitted.

TAG (ID: 6 unique ID: 13 ): (TX): Time:  0.000336
1. Tag's ID ( 6 unique ID: 13 ) is transmitted.

TAG (ID: 1 unique ID: 4 ): (TX): Time:  0.000496
1. Tag's ID ( 1 unique ID: 4 ) is transmitted.

TAG (ID: 7 unique ID: 17 ): (TX): Time:  0.000496
1. Tag's ID ( 7 unique ID: 17 ) is transmitted.

READER: (LISTEN): Time:  0.000496
1. Collision occurs. (Current:  3 (tags), Total:  1 (times))

TAG (ID: 1 unique ID: 6 ): (TX): Time:  0.000656
1. Tag's ID ( 1 unique ID: 6 ) is transmitted.

READER: (LISTEN): Time:  0.000656
1. Collision occurs. (Current:  2 (tags), Total:  2 (times))

READER: (LISTEN): Time:  0.000816
1. A TAG (unique ID: 6 ) is identified.

TAG (ID: 2 unique ID: 5 ): (TX): Time:  0.000976
1. Tag's ID ( 2 unique ID: 5 ) is transmitted.

TAG (ID: 5 unique ID: 10 ): (TX): Time:  0.001136
1. Tag's ID ( 5 unique ID: 10 ) is transmitted.

READER: (LISTEN): Time:  0.001136
1. A TAG (unique ID: 5 ) is identified.

READER: (LISTEN): Time:  0.001296
1. A TAG (unique ID: 10 ) is identified.

READER: (TX): Time:  0.001456
1. Sends a command: Select a tag (ID:  6 )
1. Sends a command: Select a tag (ID:  5 )
1. Sends a command: Select a tag (ID:  10 )
2. Sends a command: READ
Number of identified tags:  3 (ea), TIME:  0.001456
READER: (READ_CYCLE_TIME_OUT begin): TIME:  0.001456

TAG (ID: 1 unique ID: 6 ): (READ): Time:  0.0016 , tag is selected and being
muted.
```

```
TAG (ID: 2 unique ID: 5 ): (READ): Time:  0.001744 , tag is selected and being
muted.

TAG (ID: 5 unique ID: 10 ): (READ): Time:  0.001888 , tag is selected and being
muted.

TAG (ID: 1 unique ID: 2 ): (READ): Time:  0.002064 , Received frame size:  8
(slots).
TAG (ID: 1 unique ID: 2 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 2 unique ID: 3 ): (READ): Time:  0.002064 , Received frame size:  8
(slots).
TAG (ID: 2 unique ID: 3 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 1 unique ID: 4 ): (READ): Time:  0.002064 , Received frame size:  8
(slots).
TAG (ID: 1 unique ID: 4 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 6 unique ID: 13 ): (READ): Time:  0.002064 , Received frame size:  8
(slots).
TAG (ID: 6 unique ID: 13 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 7 unique ID: 17 ): (READ): Time:  0.002064 , Received frame size:  8
(slots).
TAG (ID: 7 unique ID: 17 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 2 unique ID: 3 ): (TX): Time:  0.002064
1. Tag's ID ( 2 unique ID: 3 ) is transmitted.

TAG (ID: 7 unique ID: 17 ): (TX): Time:  0.002064
1. Tag's ID ( 7 unique ID: 17 ) is transmitted.

TAG (ID: 1 unique ID: 2 ): (TX): Time:  0.002224
1. Tag's ID ( 1 unique ID: 2 ) is transmitted.

TAG (ID: 6 unique ID: 13 ): (TX): Time:  0.002224
1. Tag's ID ( 6 unique ID: 13 ) is transmitted.

READER: (LISTEN): Time:  0.002224
1. Collision occurs. (Current:  2 (tags), Total:  3 (times))

TAG (ID: 1 unique ID: 4 ): (TX): Time:  0.002384
1. Tag's ID ( 1 unique ID: 4 ) is transmitted.

READER: (LISTEN): Time:  0.002384
1. Collision occurs. (Current:  2 (tags), Total:  4 (times))

READER: (LISTEN): Time:  0.002544
1. The non-unique tag (unique ID: 4 ) is identified.

READER: (TX): Time:  0.003344
1. Sends a command: Select a tag (ID:  4 )
2. Sends a command: READ
Number of identified tags:  1 (ea), TIME:  0.003344
READER: (READ_CYCLE_TIME_OUT begin): TIME:  0.003344

TAG (ID: 1 unique ID: 4 ): (READ): Time:  0.003488 , tag is selected and being
muted.

TAG (ID: 1 unique ID: 2 ): (READ): Time:  0.003664 , Received frame size:  8
(slots).
TAG (ID: 1 unique ID: 2 ): (READCYCLE_TIMEOUT begin).
```

```
TAG (ID: 2 unique ID: 3 ): (READ): Time:  0.003664 , Received frame size:  8
(slots).
TAG (ID: 2 unique ID: 3 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 6 unique ID: 13 ): (READ): Time:  0.003664 , Received frame size:  8
(slots).
TAG (ID: 6 unique ID: 13 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 7 unique ID: 17 ): (READ): Time:  0.003664 , Received frame size:  8
(slots).
TAG (ID: 7 unique ID: 17 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 7 unique ID: 17 ): (TX): Time:  0.003664
1. Tag's ID ( 7 unique ID: 17 ) is transmitted.

TAG (ID: 2 unique ID: 3 ): (TX): Time:  0.003824
1. Tag's ID ( 2 unique ID: 3 ) is transmitted.

TAG (ID: 6 unique ID: 13 ): (TX): Time:  0.003824
1. Tag's ID ( 6 unique ID: 13 ) is transmitted.

READER: (LISTEN): Time:  0.003824
1. A TAG (unique ID: 17 ) is identified.

TAG (ID: 1 unique ID: 2 ): (TX): Time:  0.003984
1. Tag's ID ( 1 unique ID: 2 ) is transmitted.

READER: (LISTEN): Time:  0.003984
1. Collision occurs. (Current:  2 (tags), Total:  5 (times))

READER: (LISTEN): Time:  0.004144
1. The non-unique tag (unique ID: 2 ) is identified.

READER: (TX): Time:  0.004944
1. Sends a command: Select a tag (ID:  17 )
1. Sends a command: Select a tag (ID:  2 )
2. Sends a command: READ
Number of identified tags:  2 (ea), TIME:  0.004944
READER: (READ_CYCLE_TIME_OUT begin): TIME:  0.004944

TAG (ID: 7 unique ID: 17 ): (READ): Time:  0.005088 , tag is selected and being
muted.

TAG (ID: 1 unique ID: 2 ): (READ): Time:  0.005232 , tag is selected and being
muted.

TAG (ID: 2 unique ID: 3 ): (READ): Time:  0.005408 , Received frame size:  8
(slots).
TAG (ID: 2 unique ID: 3 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 6 unique ID: 13 ): (READ): Time:  0.005408 , Received frame size:  8
(slots).
TAG (ID: 6 unique ID: 13 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 2 unique ID: 3 ): (TX): Time:  0.006208
1. Tag's ID ( 2 unique ID: 3 ) is transmitted.

TAG (ID: 6 unique ID: 13 ): (TX): Time:  0.006208
1. Tag's ID ( 6 unique ID: 13 ) is transmitted.

READER: (LISTEN): Time:  0.006368
1. Collision occurs. (Current:  2 (tags), Total:  6 (times))

READER: (TX): Time:  0.006688
2. Sends a command: READ
```

```
Number of identified tags:  0 (ea), TIME:  0.006688
READER: (READ_CYCLE_TIME_OUT begin): TIME:  0.006688

TAG (ID: 2 unique ID: 3 ): (READ): Time:  0.006864 , Received frame size:  8
(slots).
TAG (ID: 2 unique ID: 3 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 6 unique ID: 13 ): (READ): Time:  0.006864 , Received frame size:  8
(slots).
TAG (ID: 6 unique ID: 13 ): (READCYCLE_TIMEOUT begin).

TAG (ID: 6 unique ID: 13 ): (TX): Time:  0.007184
1. Tag's ID ( 6 unique ID: 13 ) is transmitted.

TAG (ID: 2 unique ID: 3 ): (TX): Time:  0.007344
1. Tag's ID ( 2 unique ID: 3 ) is transmitted.

READER: (LISTEN): Time:  0.007344
1. A TAG (unique ID: 13 ) is identified.

READER: (LISTEN): Time:  0.007504
1. The non-unique tag (unique ID: 3 ) is identified.

READER: (TX): Time:  0.008144
1. Sends a command: Select a tag (ID:  13 )
1. Sends a command: Select a tag (ID:  3 )
2. Sends a command: READ
Number of identified tags:  2 (ea), TIME:  0.008144
READER: (READ_CYCLE_TIME_OUT begin): TIME:  0.008144
|-------------------------------------------------------------------------|
| Simulation terminated by process (BFSA_Muting_reader) at module (top.Logical
Network.READER.READER_PROC), T (0.008144), EV (799)   |
| Reader finished the census
|
|
|
|
|
|-------------------------------------------------------------------------|
|-------------------------------------------------------------------------|
| Simulation Completed - Collating Results.                               |
| Events: Total (800); Average Speed (17,020 events/sec.)           |
| Time  : Elapsed (0.05 sec.); Simulated (0.01 sec.)                      |
| DES Log: 2 entries
|
|-------------------------------------------------------------------------|
```

APPENDIX B
The Source Code for BFSA Muting Tag with Non-Unique Tags

```
/* Process model C form file: BFSA_Muting_tag.pr.c */

/* This variable carries the header into the object file */
const char BFSA_Muting_tag_pr_c [] = "MIL_3_Tfile_Hdr_ 145A 30A
op_runsim 7 4D72DEAA 4D72DEAA 1 SOCOMTLO2 n00641712 0 0 none none 0 0
none 0 0 0 0 0 0 0 0 1e80 8
";

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


/* Header Block */

/* IN_STRM/OUT_STRM represents the Index of stream at the process
"TAG_PROC" */
/* Look at the connectivity of this process */
/* The number given below represents the number of stream */
#define IN_STRM 0
#define OUT_STRM 0

/* BEGIN: User defined Static variables */

/* 1. Commands used by both reader and tag */
#define REQUEST 1
#define SELECT 2
#define TRANSMISSON_FAIL 3
#define TRANSMISSON_SUCCESS 4

/* 2. The size of response packet (bits), (Tag -> Reader) */
/*    & FSA_request packet (bits), (Reader -> Tag) */
#define RESPONSE_PKT_SIZE 80
#define BFSA_READ_PKT_SIZE 88

/* 3.interrupt code for time out */
#define time_out 0
#define frame_time_out 1
#define readcycle_time_out 2

/* 4. Types of Flag used in the process() */
#define OUT_OF_POWER 1
#define ENOUGH_POWER 2

/* 5. TRUE/FALSE */
#define TRUE 1
#define FALSE 0

/* 6. Tag's Status */
#define TX 0
```

```
#define READ 1

/* END:  */

#define PKT_RCVD_FROM_READER ( op_intrpt_type() == OPC_INTRPT_STRM && \
                                op_intrpt_strm() == IN_STRM )
#define TIME_OUT ( op_intrpt_type() == OPC_INTRPT_SELF && \
                   op_intrpt_code() == time_out )
#define READ_CYCLE_TIME_OUT ( op_intrpt_type() == OPC_INTRPT_SELF && \
                                          op_intrpt_code() ==
readcycle_time_out )

/* End of Header Block */

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



/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      int   selected ;     /* Indicates the status of tag which is
whether selcted or not */
      Stathandle  number_successful_command ;
      Stathandle  number_collision ;      /*  */
      Stathandle  number_selected_tag ;
      Stathandle  received_power     ;
      Evhandle    timeout    ;
      int               command     ;
      int               count_success_command;   /* Counter for the
number of success */
      Stathandle  number_received_command      ;
      int               count_received_command        ;
      int               count_selected_tag      ;
      int               serial_number           ;
      double                  A_SLOT_DURATION     ;
      int               Frame_size        ;
      int               RANGE   ;
      int               frame_flag     ;
      Evhandle    frame_timeout         ;
      int               first_time             ;
      int               readcycle_flag         ;
      Evhandle    readcycle_timeout     ;
      int               readcycle_flag2     ;
      int               tag_status    ;
      int               uniqueID    ;
```

```
      } BFSA_Muting_tag_state;

#define selected                                op_sv_ptr->selected
#define number_successful_command   op_sv_ptr-
>number_successful_command
#define number_collision                  op_sv_ptr->number_collision
#define number_selected_tag               op_sv_ptr-
>number_selected_tag
#define received_power                    op_sv_ptr->received_power
#define timeout                                 op_sv_ptr->timeout
#define command                                 op_sv_ptr->command
#define count_success_command             op_sv_ptr-
>count_success_command
#define number_received_command           op_sv_ptr-
>number_received_command
#define count_received_command            op_sv_ptr-
>count_received_command
#define count_selected_tag                op_sv_ptr->count_selected_tag
#define serial_number                           op_sv_ptr-
>serial_number
#define A_SLOT_DURATION             op_sv_ptr->A_SLOT_DURATION
#define Frame_size                        op_sv_ptr->Frame_size
#define RANGE                             op_sv_ptr->RANGE
#define frame_flag                        op_sv_ptr->frame_flag
#define frame_timeout                     op_sv_ptr->frame_timeout
#define first_time                        op_sv_ptr->first_time
#define readcycle_flag                    op_sv_ptr->readcycle_flag
#define readcycle_timeout           op_sv_ptr->readcycle_timeout
#define readcycle_flag2                   op_sv_ptr->readcycle_flag2
#define tag_status                        op_sv_ptr->tag_status
#define uniqueID                          op_sv_ptr->uniqueID

/* 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      BFSA_Muting_tag_state *op_sv_ptr;
#define FIN_PREAMBLE_CODE     \
           op_sv_ptr = ((BFSA_Muting_tag_state *)(OP_SIM_CONTEXT_PTR-
>_op_mod_state_ptr));


/* Function Block */

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

void Timeout(void) {

      FIN(Timeout());

      tag_status = TX;

      FOUT;
}
```

```
void Readcycle_timeout(void) {

      FIN(Readcycle_timeout());
//    printf("%s %i %s %g %s", "\nTAG (ID:",serial_number,"):
(READ_CYCLE_TIME_OUT end): Time: ", op_sim_time()," \n");

      readcycle_flag = TRUE;
      command = -1;  /* Resets command to check the arrival of new
REQUEST command  */

      FOUT;

}

/* End of Function Block */

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

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

#if defined (__cplusplus)
extern "C" {
#endif
      void BFSA_Muting_tag (OP_SIM_CONTEXT_ARG_OPT);
      VosT_Obtype _op_BFSA_Muting_tag_init (int * init_block_ptr);
      void _op_BFSA_Muting_tag_diag (OP_SIM_CONTEXT_ARG_OPT);
      void _op_BFSA_Muting_tag_terminate (OP_SIM_CONTEXT_ARG_OPT);
      VosT_Address _op_BFSA_Muting_tag_alloc (VosT_Obtype, int);
      void _op_BFSA_Muting_tag_svar (void *, const char *, void **);


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




/* Process model interrupt handling procedure */


void
BFSA_Muting_tag (OP_SIM_CONTEXT_ARG_OPT)
      {
#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      FIN_MT (BFSA_Muting_tag ());

            {
            /* Temporary Variables */
            Objid tag_id, tx_id, ctag_id;
            Objid comp_attr_objid, comp_attr_row_objid;
```

```
            Packet *pkout, *pkin;
            OpT_Packet_Size pkt_size;

            double data_rate = 0.0;
            double random;
            double rx_power_tag;

            int ID;
            /* End of Temporary Variables */


            FSM_ENTER ("BFSA_Muting_tag")

            FSM_BLOCK_SWITCH
                  {
                  /*-------------------------------------------------
------*/
                  /** state (INIT) enter executives **/
                  FSM_STATE_ENTER_FORCED_NOLABEL (0, "INIT",
"BFSA_Muting_tag [INIT enter execs]")
                        FSM_PROFILE_SECTION_IN ("BFSA_Muting_tag [INIT
enter execs]", state0_enter_exec)
                        {
                        /* Retrieves the user TagID of each tag defined
in the attribute */
                              tag_id=op_topo_parent(op_id_self());
                              op_ima_obj_attr_get(tag_id, "user id",
&uniqueID);

                        /* Retrieves the Calculated TagID of each tag
defined in the attribute */
                              ctag_id= tag_id;
                              op_ima_obj_attr_get(ctag_id, "Calculated
TagID", &serial_number);

                        /* Retrieves the data rate of each tag defined
in the attribute of TX processor */
                              tx_id = op_id_from_name(tag_id,
OPC_OBJTYPE_RATX, "TX");
                              op_ima_obj_attr_get(tx_id, "channel",
&comp_attr_objid);
                              comp_attr_row_objid =
op_topo_child(comp_attr_objid, OPC_OBJTYPE_RATXCH, 0);
                              op_ima_obj_attr_get(comp_attr_row_objid,
"data rate", &data_rate);


                        /* Calculates the duration of a single slot */
                              A_SLOT_DURATION = (RESPONSE_PKT_SIZE /
data_rate)+0.0000000000001;

                              printf("%s %d %s", "\nTag's ID: ",
uniqueID, " \n");
                           printf("%s %d %s", "Calculated Unique Tag's
ID: ", serial_number, " \n");
                              printf("%s %f %s", "Data rate (Tag-
>Reader): ", data_rate, "(bps)\n");
                              printf("%s %i %s", "Response packet size:
", RESPONSE_PKT_SIZE, "(bits) \n");
```

```
                        /* Initializes the variables */
                              selected = FALSE;
                              command = -1;
                              count_received_command = 0;
                              count_success_command = 0;
                              first_time = TRUE;
                              frame_flag = TRUE;
                              tag_status = READ;


                        /* Declares the statistical variables */
                              number_received_command =
op_stat_reg("Number of received
command",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              number_successful_command =
op_stat_reg("Number of successful
command",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              received_power = op_stat_reg("Received
power",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                        }
                        FSM_PROFILE_SECTION_OUT (state0_enter_exec)

                  /** state (INIT) exit executives **/
                  FSM_STATE_EXIT_FORCED (0, "INIT", "BFSA_Muting_tag
[INIT exit execs]")


                  /** state (INIT) transition processing **/
                  FSM_TRANSIT_FORCE (1, state1_enter_exec, ;,
"default", "", "INIT", "START", "tr_33", "BFSA_Muting_tag [INIT ->
START : default / ]")
                        /*---------------------------------------------
------------*/



                  /** state (START) enter executives **/
                  FSM_STATE_ENTER_UNFORCED (1, "START",
state1_enter_exec, "BFSA_Muting_tag [START enter execs]")
                        FSM_PROFILE_SECTION_IN ("BFSA_Muting_tag [START
enter execs]", state1_enter_exec)
                        {
                        if (command == REQUEST) {

                              if ( readcycle_flag == TRUE ) {
                                    readcycle_flag = FALSE;

                                    random = (int) (op_dist_uniform(1)
* (Frame_size-1));

      timeout=op_intrpt_schedule_self(op_sim_time() + \

      random*A_SLOT_DURATION, time_out);

                                    if ( readcycle_flag2 == TRUE ) {
                                          readcycle_flag2 = FALSE;
```

```
                                        /* The timer to inform
beginning of a new read cycle */

      readcycle_timeout=op_intrpt_schedule_self(op_sim_time() + \

Frame_size*A_SLOT_DURATION, \

readcycle_time_out);

                                          printf ("%s %i %s %i %s",
"TAG (ID:", uniqueID ,"unique ID:", serial_number,"):
(READCYCLE_TIMEOUT begin). \n");
                                    }

                               }

                         } else {
                               /* Keeps silent until REQUEST command is
arrived. */
                         }
                         }
                         FSM_PROFILE_SECTION_OUT (state1_enter_exec)

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


                  /** state (START) exit executives **/
                  FSM_STATE_EXIT_UNFORCED (1, "START", "BFSA_Muting_tag
[START exit execs]")


                  /** state (START) transition processing **/
                  FSM_PROFILE_SECTION_IN ("BFSA_Muting_tag [START trans
conditions]", state1_trans_conds)
                  FSM_INIT_COND (PKT_RCVD_FROM_READER)
                  FSM_TEST_COND (TIME_OUT)
                  FSM_TEST_COND (READ_CYCLE_TIME_OUT)
                  FSM_TEST_LOGIC ("START")
                  FSM_PROFILE_SECTION_OUT (state1_trans_conds)

                  FSM_TRANSIT_SWITCH
                         {
                         FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;,
"PKT_RCVD_FROM_READER", "", "START", "READ", "tr_41", "BFSA_Muting_tag
[START -> READ : PKT_RCVD_FROM_READER / ]")
                         FSM_CASE_TRANSIT (1, 3, state3_enter_exec,
Timeout();, "TIME_OUT", "Timeout()", "START", "TX", "tr_47",
"BFSA_Muting_tag [START -> TX : TIME_OUT / Timeout()]")
                         FSM_CASE_TRANSIT (2, 1, state1_enter_exec,
Readcycle_timeout();, "READ_CYCLE_TIME_OUT", "Readcycle_timeout()",
"START", "START", "tr_50", "BFSA_Muting_tag [START -> START :
READ_CYCLE_TIME_OUT / Readcycle_timeout()]")
                         }
                         /*-----------------------------------------
------------*/
```

```
                  /** state (READ) enter executives **/
                  FSM_STATE_ENTER_FORCED (2, "READ", state2_enter_exec,
"BFSA_Muting_tag [READ enter execs]")
                        FSM_PROFILE_SECTION_IN ("BFSA_Muting_tag [READ
enter execs]", state2_enter_exec)
                        {
                        pkin = op_pk_get(op_intrpt_strm());

                        /* If a packet is comes from itself then
destroy. */
                        /* This model uses the same Frequency for TX
and RX */
                        pkt_size = op_pk_total_size_get (pkin);
                        if ( pkt_size != RESPONSE_PKT_SIZE ) {

                        if ( tag_status == READ ) {

                              /* Measures the number of command
received */
                              count_received_command++;
                              op_stat_write(number_received_command,
count_received_command);

                              /* Measures the received power */
                              rx_power_tag =
op_td_get_dbl(pkin,OPC_TDA_RA_RCVD_POWER);
                              op_stat_write(received_power,
rx_power_tag);


                              if (selected == TRUE) {
                                    /* TAG is SELECTED. So, Keeps
muting. */

                              } else {

                                    op_pk_fd_get(pkin,0,&command);

                                    /* Measures the number of
successful command received */
                                    count_success_command++;

      op_stat_write(number_successful_command, count_success_command);

                                    if (command == SELECT) {
                                          op_pk_fd_get(pkin,1,&ID);

                                          if (ID == serial_number) {
                                                selected = TRUE;
                                                printf("%s %i %s %i %s
%g %s", "\nTAG (ID:",uniqueID,"unique ID:", serial_number ,"): (READ):
Time: ", op_sim_time(), ", tag is selected and being muted.\n");

                                          }

                                    } else if (command == REQUEST) {

      op_pk_fd_get(pkin,2,&Frame_size);
```

```
                                        printf("%s %i %s %i %s %g %s
%i %s", "\nTAG (ID:",uniqueID,"unique ID:", serial_number ,"): (READ):
Time: ", op_sim_time(), ", Received frame size: ", Frame_size,
"(slots). \n");

                                        readcycle_flag = TRUE;
                                        readcycle_flag2 = TRUE;
                                  }
                            }

                      }

                      }

                      op_pk_destroy (pkin);
                      }
                      FSM_PROFILE_SECTION_OUT (state2_enter_exec)

                /** state (READ) exit executives **/
                FSM_STATE_EXIT_FORCED (2, "READ", "BFSA_Muting_tag
[READ exit execs]")


                /** state (READ) transition processing **/
                FSM_TRANSIT_FORCE (1, state1_enter_exec, ;,
"default", "", "READ", "START", "tr_42", "BFSA_Muting_tag [READ ->
START : default / ]")
                      /*---------------------------------------------
------------*/



                /** state (TX) enter executives **/
                FSM_STATE_ENTER_FORCED (3, "TX", state3_enter_exec,
"BFSA_Muting_tag [TX enter execs]")
                      FSM_PROFILE_SECTION_IN ("BFSA_Muting_tag [TX
enter execs]", state3_enter_exec)
                      {
                      printf("%s %i %s %i %s %g %s", "\nTAG (ID:",
uniqueID, "unique ID:", serial_number ,"): (TX): Time: ",
op_sim_time(), "\n");

                      if ( tag_status == TX ) {

                            if (selected == TRUE) {  /* If a tag is
selected then must keep slient  */
                                  printf("%s %i %s %i %s", "1. Tag
(ID:", uniqueID,"unique ID:", serial_number ,") has been SELECTED. Keep
muting. \n");
                            } else {

                                  if(command == REQUEST) {
                                        pkout =
op_pk_create_fmt("response_pkt");

      op_pk_nfd_set_int32(pkout,"ID",serial_number);
                                        op_pk_send(pkout,OUT_STRM);
```

```
                                        printf("%s %d %s %d %s", "1.
Tag's ID (", uniqueID ,"unique ID:", serial_number ,") is transmitted.
\n");

                                    } else {
                                        printf("%s %i %s ", "1.
Current command (", command, ") is not READ command. \n");
                                    }
                                }

                            }
                            }
                            FSM_PROFILE_SECTION_OUT (state3_enter_exec)

                        /** state (TX) exit executives **/
                        FSM_STATE_EXIT_FORCED (3, "TX", "BFSA_Muting_tag [TX
exit execs]")
                            FSM_PROFILE_SECTION_IN ("BFSA_Muting_tag [TX
exit execs]", state3_exit_exec)
                            {
                            tag_status = READ;
                            }
                            FSM_PROFILE_SECTION_OUT (state3_exit_exec)


                        /** state (TX) transition processing **/
                        FSM_TRANSIT_FORCE (1, state1_enter_exec, ;,
"default", "", "TX", "START", "tr_48", "BFSA_Muting_tag [TX -> START :
default / ]")
                            /*---------------------------------------------
------------*/



                        }


                FSM_EXIT (0,"BFSA_Muting_tag")
                }
        }




void
_op_BFSA_Muting_tag_diag (OP_SIM_CONTEXT_ARG_OPT)
        {
        /* No Diagnostic Block */
        }




void
_op_BFSA_Muting_tag_terminate (OP_SIM_CONTEXT_ARG_OPT)
        {

        FIN_MT (_op_BFSA_Muting_tag_terminate ())
```

```
	/* No Termination Block */

	Vos_Poolmem_Dealloc (op_sv_ptr);

	FOUT
	}


/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in _op_BFSA_Muting_tag_svar function. */
#undef selected
#undef number_successful_command
#undef number_collision
#undef number_selected_tag
#undef received_power
#undef timeout
#undef command
#undef count_success_command
#undef number_received_command
#undef count_received_command
#undef count_selected_tag
#undef serial_number
#undef A_SLOT_DURATION
#undef Frame_size
#undef RANGE
#undef frame_flag
#undef frame_timeout
#undef first_time
#undef readcycle_flag
#undef readcycle_timeout
#undef readcycle_flag2
#undef tag_status
#undef uniqueID

#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

VosT_Obtype
_op_BFSA_Muting_tag_init (int * init_block_ptr)
	{
	VosT_Obtype obtype = OPC_NIL;
	FIN_MT (_op_BFSA_Muting_tag_init (init_block_ptr))

	obtype = Vos_Define_Object_Prstate ("proc state vars
(BFSA_Muting_tag)",
		    sizeof (BFSA_Muting_tag_state));
	*init_block_ptr = 0;

	FRET (obtype)
	}

VosT_Address
_op_BFSA_Muting_tag_alloc (VosT_Obtype obtype, int init_block)
	{
```

```
#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      BFSA_Muting_tag_state * ptr;
      FIN_MT (_op_BFSA_Muting_tag_alloc (obtype))

      ptr = (BFSA_Muting_tag_state *)Vos_Alloc_Object (obtype);
      if (ptr != OPC_NIL)
            {
            ptr->_op_current_block = init_block;
#if defined (OPD_ALLOW_ODB)
            ptr->_op_current_state = "BFSA_Muting_tag [INIT enter
execs]";
#endif
            }
      FRET ((VosT_Address)ptr)
      }



void
_op_BFSA_Muting_tag_svar (void * gen_ptr, const char * var_name, void
** var_p_ptr)
      {
      BFSA_Muting_tag_state         *prs_ptr;

      FIN_MT (_op_BFSA_Muting_tag_svar (gen_ptr, var_name, var_p_ptr))

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

      if (strcmp ("selected" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->selected);
            FOUT
            }
      if (strcmp ("number_successful_command" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>number_successful_command);
            FOUT
            }
      if (strcmp ("number_collision" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->number_collision);
            FOUT
            }
      if (strcmp ("number_selected_tag" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->number_selected_tag);
            FOUT
            }
      if (strcmp ("received_power" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->received_power);
```

```
      FOUT
      }
if (strcmp ("timeout" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->timeout);
      FOUT
      }
if (strcmp ("command" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->command);
      FOUT
      }
if (strcmp ("count_success_command" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->count_success_command);
      FOUT
      }
if (strcmp ("number_received_command" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->number_received_command);
      FOUT
      }
if (strcmp ("count_received_command" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->count_received_command);
      FOUT
      }
if (strcmp ("count_selected_tag" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->count_selected_tag);
      FOUT
      }
if (strcmp ("serial_number" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->serial_number);
      FOUT
      }
if (strcmp ("A_SLOT_DURATION" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->A_SLOT_DURATION);
      FOUT
      }
if (strcmp ("Frame_size" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->Frame_size);
      FOUT
      }
if (strcmp ("RANGE" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->RANGE);
      FOUT
      }
if (strcmp ("frame_flag" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->frame_flag);
      FOUT
      }
if (strcmp ("frame_timeout" , var_name) == 0)
      {
```

```
      *var_p_ptr = (void *) (&prs_ptr->frame_timeout);
      FOUT
      }
if (strcmp ("first_time" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->first_time);
      FOUT
      }
if (strcmp ("readcycle_flag" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->readcycle_flag);
      FOUT
      }
if (strcmp ("readcycle_timeout" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->readcycle_timeout);
      FOUT
      }
if (strcmp ("readcycle_flag2" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->readcycle_flag2);
      FOUT
      }
if (strcmp ("tag_status" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->tag_status);
      FOUT
      }
if (strcmp ("uniqueID" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->uniqueID);
      FOUT
      }
*var_p_ptr = (void *)OPC_NIL;

FOUT
}
```

APPENDIX C

The Source Code for BFSA Muting Reader to Identify Non-Unique Tags

```
/* Process model C form file: BFSA_Muting_reader.pr.c */


/* This variable carries the header into the object file */
const char BFSA_Muting_reader_pr_c [] = "MIL_3_Tfile_Hdr_ 145A 30A
modeler 7 4D7595BB 4D7595BB 1 SOCOMTLO2 n00641712 0 0 none none 0 0
none 0 0 0 0 0 0 0 0 1e80 8
";

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


/* Header Block */

/* BEGIN: The index of an Input/Output stream */

/* Input Stream from tags */
#define IN_STRM 0

/* Output stream from this processor to tags */
#define OUT_STRM 0

/* END */


/* BEGIN: User defined Static variables */

/* 1.Commands used by both reader and tag */
#define REQUEST 1
#define SELECT 2
#define TRANSMISSON_FAIL 3
#define TRANSMISSON_SUCCESS 4

/* 2.interrupt code for time out */
#define time_out 0
#define no_collision 1
#define assurance_level 2

/* 3.Number of frames in a read cycle */
/*   & Number of slots per a frame */
/*   & The size of response packet */
#define RESPONSE_PKT_SIZE 80
#define BFSA_READ_PKT_SIZE 88
#define SELECT_PKT_SIZE 72
#define ASSURANCE_LEVEL 0.99

/* 4.Flag for read cycle */
#define LISTEN 0
```

```
#define TX 1

/* 5. TRUE/FALSE */
#define TRUE 1
#define FALSE 0

/* END */


/* Conditional macros */
#define TIME_OUT (op_intrpt_type() == OPC_INTRPT_SELF && \
                  op_intrpt_code() == time_out)
#define NO_COLLISION (op_intrpt_type() == OPC_INTRPT_SELF && \
                      op_intrpt_code() == no_collision)
#define REACH_ASSURANCE_LEVEL (op_intrpt_type() == OPC_INTRPT_SELF && \
                                                op_intrpt_code() ==
assurance_level)
#define PKT_RCVD_FROM_TAG (op_intrpt_type() == OPC_INTRPT_STRM && \
                           op_intrpt_strm() == IN_STRM)

/* End of Header Block */

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



/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      int     reader_status; /* The flag to determine the read cycle is
performing or not */
      Evhandle       timeout    ;
      Stathandle     collision_handle     ;
      Stathandle     SH_collisions_in_a_time;   /* Status handler for
the Number of collisions occurs in a time */
      int                   collision_counter    ;    /* Count the
number of collision occured */
      Stathandle      SH_total_num_collisions   ;     /* Statistics
Handler for the total number of collisions during census */
      Stathandle       SH_selected_tag_ID   ;   /* The Statistic
handler for selected tag's ID */
      Stathandle       SH_total_num_selected_tag   ;  /* The statistic
handle for the total number of selected tags */
      Evhandle         nocollision;
      int                   tag_ID[100];
      int                   started    ;
```

```
      int                   first_time ;
      int                   new_variable         ;
      int                   tag_counter;
      int                   count_selected_tag   ;
      int                   selected_ID[100]     ;
      int                   cnt        ;
      double                      A_READ_CYCLE_DURATION;
      double                      A_SLOT_DURATION      ;
      int                   readcycle_flag       ;
      double                      BFSA_READ_PKT_TX_TIME;
      double                      SELECT_PKT_TX_TIME   ;
      int                   unread_tags;
      int                   number_of_identified_tags_per_frame
;
      int                   number_of_collided_slots_per_frame
;
      int                   number_of_idle_slots_per_frame
;
      int                   current_frame_size   ;
      Stathandle       SH_number_of_identified_tags_per_frame
;
      Stathandle       SH_number_of_collided_slots_per_frame
;
      Stathandle       SH_number_of_idle_slots_per_frame
;
      Stathandle       SH_current_frame_size;
      int                   sum_of_frame_size    ;
      int                   sum_of_collided_slots_per_frame
;
      int                   sum_of_idle_slots_per_frame ;
      Stathandle       SH_number_of_unread_tags_per_frame
;
      Stathandle       SH_total_delay       ;
      Stathandle       SH_collision_delay   ;
      Stathandle       SH_idle_delay        ;
      double                      sum_of_identified_tags_per_frame
;
      Stathandle       SH_success_delay     ;
      Evhandle         EV_assurance_level   ;
      Stathandle       SH_assurance_level   ;
      double                      assuranceLevel       ;
      int                   actual_number_of_tags;
      int                   frame_size ;
      Stathandle       SH_num_required_read_cycle  ;
      Stathandle       SH_num_required_slots;
      int                   count_read_cycle     ;
      int                   total_slots;
      int                   non-unique_tags       ;
      Stathandle       SH_non-unique_tags_count     ;
      int                   count_tag_id[100]    ;
      } BFSA_Muting_reader_state;

#define reader_status                     op_sv_ptr->reader_status
#define timeout                           op_sv_ptr->timeout
#define collision_handle                  op_sv_ptr->collision_handle
#define SH_collisions_in_a_time           op_sv_ptr-
>SH_collisions_in_a_time
#define collision_counter                 op_sv_ptr->collision_counter
```

```
#define SH_total_num_collisions           op_sv_ptr-
>SH_total_num_collisions
#define SH_selected_tag_ID                op_sv_ptr->SH_selected_tag_ID
#define SH_total_num_selected_tag         op_sv_ptr-
>SH_total_num_selected_tag
#define nocollision                       op_sv_ptr->nocollision
#define tag_ID                            op_sv_ptr->tag_ID
#define started                           op_sv_ptr->started
#define first_time                        op_sv_ptr->first_time
#define new_variable                      op_sv_ptr->new_variable
#define tag_counter                       op_sv_ptr->tag_counter
#define count_selected_tag                op_sv_ptr->count_selected_tag
#define selected_ID                       op_sv_ptr->selected_ID
#define cnt                               op_sv_ptr->cnt
#define A_READ_CYCLE_DURATION   op_sv_ptr->A_READ_CYCLE_DURATION
#define A_SLOT_DURATION                   op_sv_ptr->A_SLOT_DURATION
#define readcycle_flag                    op_sv_ptr->readcycle_flag
#define BFSA_READ_PKT_TX_TIME       op_sv_ptr->BFSA_READ_PKT_TX_TIME
#define SELECT_PKT_TX_TIME          op_sv_ptr->SELECT_PKT_TX_TIME
#define unread_tags                       op_sv_ptr->unread_tags
#define number_of_identified_tags_per_frame
                        op_sv_ptr->number_of_identified_tags_per_frame
#define number_of_collided_slots_per_frame
                        op_sv_ptr->number_of_collided_slots_per_frame
#define number_of_idle_slots_per_frame
                        op_sv_ptr->number_of_idle_slots_per_frame
#define current_frame_size                op_sv_ptr->current_frame_size
#define SH_number_of_identified_tags_per_frame
                        op_sv_ptr-
                   >SH_number_of_identified_tags_per_frame
#define SH_number_of_collided_slots_per_frame
                        op_sv_ptr-
                   >SH_number_of_collided_slots_per_frame
#define SH_number_of_idle_slots_per_frame
                        op_sv_ptr->SH_number_of_idle_slots_per_frame
#define SH_current_frame_size             op_sv_ptr-
>SH_current_frame_size
#define sum_of_frame_size                 op_sv_ptr->sum_of_frame_size
#define sum_of_collided_slots_per_frame
                        op_sv_ptr->sum_of_collided_slots_per_frame
#define sum_of_idle_slots_per_frame
                        op_sv_ptr->sum_of_idle_slots_per_frame
#define SH_number_of_unread_tags_per_frame
                        op_sv_ptr->SH_number_of_unread_tags_per_frame
#define SH_total_delay                    op_sv_ptr->SH_total_delay
#define SH_collision_delay          op_sv_ptr->SH_collision_delay
#define SH_idle_delay                     op_sv_ptr->SH_idle_delay
#define sum_of_identified_tags_per_frame
                              op_sv_ptr-
                        >sum_of_identified_tags_per_frame
#define SH_success_delay                  op_sv_ptr->SH_success_delay
#define EV_assurance_level                op_sv_ptr->EV_assurance_level
#define SH_assurance_level                op_sv_ptr->SH_assurance_level
#define assuranceLevel                          op_sv_ptr-
>assuranceLevel
#define actual_number_of_tags             op_sv_ptr-
>actual_number_of_tags
#define frame_size                              op_sv_ptr->frame_size
#define SH_num_required_read_cycle
```

```
                              op_sv_ptr->SH_num_required_read_cycle
#define SH_num_required_slots       op_sv_ptr->SH_num_required_slots
#define count_read_cycle            op_sv_ptr->count_read_cycle
#define total_slots                       op_sv_ptr->total_slots
#define non-unique_tags             op_sv_ptr->non-unique_tags
#define SH_non-unique_tags_count    op_sv_ptr->SH_non-unique_tags_count
#define count_tag_id                      op_sv_ptr->count_tag_id

/* 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      BFSA_Muting_reader_state *op_sv_ptr;
#define FIN_PREAMBLE_CODE     \
           op_sv_ptr = ((BFSA_Muting_reader_state
*)(OP_SIM_CONTEXT_PTR->_op_mod_state_ptr));


/* Function Block */

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

void Time_out (void) {

      FIN(Time_out());

      /* Switch the flag from LISTEN to TX */
      /* This flag disables the LISTEN state of reader */
      reader_status = TX;
      readcycle_flag = TRUE;
      count_read_cycle++;

      number_of_idle_slots_per_frame = current_frame_size - \

(number_of_identified_tags_per_frame + \

number_of_collided_slots_per_frame);

      unread_tags -= number_of_identified_tags_per_frame;

      op_stat_write(SH_number_of_identified_tags_per_frame,
number_of_identified_tags_per_frame);
      op_stat_write(SH_number_of_collided_slots_per_frame,number_of_col
lided_slots_per_frame);
      op_stat_write(SH_number_of_idle_slots_per_frame,number_of_idle_sl
ots_per_frame);
      op_stat_write(SH_current_frame_size,current_frame_size);
      op_stat_write(SH_number_of_unread_tags_per_frame,unread_tags);
      op_stat_write(SH_total_num_selected_tag,count_selected_tag);


      sum_of_frame_size += current_frame_size;
      sum_of_collided_slots_per_frame +=
number_of_collided_slots_per_frame;
      sum_of_idle_slots_per_frame += number_of_idle_slots_per_frame;
```

```
      sum_of_identified_tags_per_frame +=
number_of_identified_tags_per_frame;

      if ( number_of_collided_slots_per_frame == 0 && first_time ==
FALSE ) {
            nocollision =
op_intrpt_schedule_self(op_sim_time(),no_collision);
      }
      first_time = FALSE;

      number_of_idle_slots_per_frame = 0;
      number_of_identified_tags_per_frame = 0;
      number_of_collided_slots_per_frame = 0;

      FOUT;
}

void finish_census (void) {

      FIN(finish_census());

//    printf("%s %g %s", "\nREADER: (NO_COLLISION): TIME: ",
op_sim_time(), " \n");
      op_stat_write(SH_total_delay,A_SLOT_DURATION*sum_of_frame_size);
      op_stat_write(SH_collision_delay,A_SLOT_DURATION*sum_of_collided_
slots_per_frame);
      op_stat_write(SH_idle_delay,A_SLOT_DURATION*sum_of_idle_slots_per
_frame);
      op_stat_write(SH_success_delay,A_SLOT_DURATION*sum_of_identified_
tags_per_frame);
      op_stat_write(SH_num_required_read_cycle,count_read_cycle-1);
      op_stat_write(SH_num_required_slots,(count_read_cycle-
1)*frame_size);
    op_stat_write(SH_non-unique_tags_count,non-unique_tags);
      //op_stat_write(SH_Non-unique_Tags_Count,non-uniqueTags);
      op_sim_end("Reader finished the census","","","");

      FOUT;
}

/* End of Function Block */

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

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

#if defined (__cplusplus)
extern "C" {
#endif
      void BFSA_Muting_reader (OP_SIM_CONTEXT_ARG_OPT);
      VosT_Obtype _op_BFSA_Muting_reader_init (int * init_block_ptr);
      void _op_BFSA_Muting_reader_diag (OP_SIM_CONTEXT_ARG_OPT);
      void _op_BFSA_Muting_reader_terminate (OP_SIM_CONTEXT_ARG_OPT);
      VosT_Address _op_BFSA_Muting_reader_alloc (VosT_Obtype, int);
```

```
      void _op_BFSA_Muting_reader_svar (void *, const char *, void **);


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



/* Process model interrupt handling procedure */


void
BFSA_Muting_reader (OP_SIM_CONTEXT_ARG_OPT)
      {
#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      FIN_MT (BFSA_Muting_reader ());

            {
            /* Temporary Variables */
            Objid objid, rx_id;
            Objid comp_attr_objid, comp_attr_row_objid;
            Packet *pkout, *pkin;
            OpT_Packet_Size pkt_size;

            double data_rate = 0.0;

            int number_collision;
            int i, ID;
            int duplicated_tag;
            /* End of Temporary Variables */


            FSM_ENTER ("BFSA_Muting_reader")

            FSM_BLOCK_SWITCH
                  {
                  /*-------------------------------------------------
------*/
                  /** state (INIT) enter executives **/
                  FSM_STATE_ENTER_FORCED_NOLABEL (0, "INIT",
"BFSA_Muting_reader [INIT enter execs]")
                        FSM_PROFILE_SECTION_IN ("BFSA_Muting_reader
[INIT enter execs]", state0_enter_exec)
                        {
                        /* Retrieves the data rate of RX of Reader
defined in the attribute of RX processor */
                              objid=op_topo_parent(op_id_self());
                              rx_id = op_id_from_name(objid,
OPC_OBJTYPE_RARX, "RX");
                              op_ima_obj_attr_get(rx_id, "channel",
&comp_attr_objid);
                              comp_attr_row_objid =
op_topo_child(comp_attr_objid, OPC_OBJTYPE_RARXCH, 0);
                              op_ima_obj_attr_get(comp_attr_row_objid,
"data rate", &data_rate);
```

```
                        /* Calculates the duration of a single slot and
a read cycle */
                              A_SLOT_DURATION = RESPONSE_PKT_SIZE /
data_rate;
                              op_ima_sim_attr_get_int32("Frame
size",&frame_size);
                              A_READ_CYCLE_DURATION =A_SLOT_DURATION *
frame_size;
                              BFSA_READ_PKT_TX_TIME =
BFSA_READ_PKT_SIZE / data_rate;
                             SELECT_PKT_TX_TIME = SELECT_PKT_SIZE /
data_rate;

                              printf("%s %f %s", "\nData rate (Reader-
>Tag): ", data_rate, "(bps) \n");
                              printf("%s %i %s", "Frame size:
",frame_size,"(slots) \n");
                             printf("%s %f %s", "A slot duration: ",
A_SLOT_DURATION, "(sec) \n");
                             printf("%s %f %s", "A read cycle duration:
", A_READ_CYCLE_DURATION, "(sec) \n");
                             printf("%s %f %s", "Assurance level:
",ASSURANCE_LEVEL," \n");


                        /* Initializes the variables */
                              first_time = TRUE;
                              reader_status = TX;
                              tag_counter = 0;
                              count_selected_tag = 0;
                              collision_counter = 0;
                              cnt = 0;
                             non-unique_tags = 0;
                              op_ima_sim_attr_get_int32("Number of
actual tags",&actual_number_of_tags);
                              unread_tags = actual_number_of_tags;
                              number_of_identified_tags_per_frame = 0;
                              number_of_collided_slots_per_frame = 0;
                              number_of_idle_slots_per_frame = 0;
                              current_frame_size = frame_size;
                              sum_of_frame_size = 0;
                              sum_of_collided_slots_per_frame = 0;
                              sum_of_idle_slots_per_frame = 0;
                              sum_of_identified_tags_per_frame = 0;
                              assuranceLevel = 0;
                              count_read_cycle = 0;
                              total_slots = 0;


                        /* Declares the statistical variables */
                              SH_total_delay = op_stat_reg("Total
delay",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_collision_delay =
op_stat_reg("Collision delay",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_idle_delay = op_stat_reg("Idle
delay",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
```

```
                              SH_success_delay = op_stat_reg("Success
delay",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_number_of_identified_tags_per_frame =
op_stat_reg("Number of identified
tags",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_number_of_collided_slots_per_frame =
op_stat_reg("Number of collided
slots",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_number_of_idle_slots_per_frame =
op_stat_reg("Number of idle slots",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_total_num_collisions =
op_stat_reg("Total number of
collisions",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_total_num_selected_tag =
op_stat_reg("Total number of identified
tags",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_number_of_unread_tags_per_frame =
op_stat_reg("Number of unread
tags",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_current_frame_size =
op_stat_reg("Current frame size",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_selected_tag_ID =
op_stat_reg("selected tag ID",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                             SH_non-unique_tags_count =
op_stat_reg("Total number of non-unique
tags",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_collisions_in_a_time =
op_stat_reg("Number of collisions",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                             SH_num_required_read_cycle =
op_stat_reg("Total number of required read
cycle",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);
                              SH_num_required_slots = op_stat_reg("Total
number of required slots",OPC_STAT_INDEX_NONE,OPC_STAT_LOCAL);


                         }
                         FSM_PROFILE_SECTION_OUT (state0_enter_exec)

                    /** state (INIT) exit executives **/
                    FSM_STATE_EXIT_FORCED (0, "INIT", "BFSA_Muting_reader
[INIT exit execs]")


                    /** state (INIT) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;,
"default", "", "INIT", "START", "tr_0", "BFSA_Muting_reader [INIT ->
START : default / ]")
                         /*---------------------------------------------
------------*/



                    /** state (START) enter executives **/
                    FSM_STATE_ENTER_UNFORCED (1, "START",
state1_enter_exec, "BFSA_Muting_reader [START enter execs]")
                         FSM_PROFILE_SECTION_IN ("BFSA_Muting_reader
[START enter execs]", state1_enter_exec)
                         {
                         if ( first_time == TRUE ) {
```

```
                              timeout =
op_intrpt_schedule_self(op_sim_time(),time_out);

                        } else if (readcycle_flag == TRUE) {
                              readcycle_flag = FALSE;
                              timeout =
op_intrpt_schedule_self(op_sim_time() + \

A_READ_CYCLE_DURATION + \

BFSA_READ_PKT_TX_TIME + \

        ((int)tag_counter * SELECT_PKT_TX_TIME)+0.0000000001,time_out);
                              printf("%s %i %s %g %s", "Number of
identified tags: ", (int)tag_counter, "(ea), TIME: ", op_sim_time(),
"\n");
                              printf("%s %g %s", "READER:
(READ_CYCLE_TIME_OUT begin): TIME: ", op_sim_time(), "\n");
                              tag_counter = 0;
                        }
                        }
                        FSM_PROFILE_SECTION_OUT (state1_enter_exec)

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


                  /** state (START) exit executives **/
                  FSM_STATE_EXIT_UNFORCED (1, "START",
"BFSA_Muting_reader [START exit execs]")


                  /** state (START) transition processing **/
                  FSM_PROFILE_SECTION_IN ("BFSA_Muting_reader [START
trans conditions]", state1_trans_conds)
                  FSM_INIT_COND (TIME_OUT)
                  FSM_TEST_COND (PKT_RCVD_FROM_TAG)
                  FSM_TEST_COND (NO_COLLISION)
                  FSM_TEST_LOGIC ("START")
                  FSM_PROFILE_SECTION_OUT (state1_trans_conds)

                  FSM_TRANSIT_SWITCH
                        {
                        FSM_CASE_TRANSIT (0, 2, state2_enter_exec,
Time_out();, "TIME_OUT", "Time_out()", "START", "TX", "tr_21",
"BFSA_Muting_reader [START -> TX : TIME_OUT / Time_out()]")
                        FSM_CASE_TRANSIT (1, 3, state3_enter_exec, ;,
"PKT_RCVD_FROM_TAG", "", "START", "LISTEN", "tr_26",
"BFSA_Muting_reader [START -> LISTEN : PKT_RCVD_FROM_TAG / ]")
                        FSM_CASE_TRANSIT (2, 1, state1_enter_exec,
finish_census();, "NO_COLLISION", "finish_census()", "START", "START",
"tr_28", "BFSA_Muting_reader [START -> START : NO_COLLISION /
finish_census()]")
                        }
                        /*-----------------------------------------
------------*/
```

```
                  /** state (TX) enter executives **/
                  FSM_STATE_ENTER_FORCED (2, "TX", state2_enter_exec,
"BFSA_Muting_reader [TX enter execs]")
                        FSM_PROFILE_SECTION_IN ("BFSA_Muting_reader [TX
enter execs]", state2_enter_exec)
                        {
                        printf("%s %g %s", "\nREADER: (TX): Time: ",
op_sim_time(), "\n");

                        if ( reader_status == TX ) {

                              if ( tag_counter != 0 ) {
                                    /* Transmits the ACK */
                                    for ( i=0; i<tag_counter; i++ ) {

      pkout=op_pk_create_fmt("select_pkt");

      op_pk_nfd_set(pkout,"ID",tag_ID[i]);

      op_pk_nfd_set(pkout,"Command",SELECT);
                                          op_pk_send(pkout,OUT_STRM);
                                          printf("%s %i %s", "1. Sends
a command: Select a tag (Unique ID: ",tag_ID[i], ")\n");
                                    }
                              }
                                    /* Transmits the READ command */

      pkout=op_pk_create_fmt("BFSA_read_pkt");

      op_pk_nfd_set(pkout,"Command",REQUEST);
                                    op_pk_nfd_set(pkout,"Frame
size",current_frame_size);
                                    printf("%s", "2. Sends a command:
READ \n");
                                    op_pk_send(pkout,OUT_STRM);
                        }
                        }
                        FSM_PROFILE_SECTION_OUT (state2_enter_exec)

                  /** state (TX) exit executives **/
                  FSM_STATE_EXIT_FORCED (2, "TX", "BFSA_Muting_reader
[TX exit execs]")
                        FSM_PROFILE_SECTION_IN ("BFSA_Muting_reader [TX
exit execs]", state2_exit_exec)
                        {
                              /* Reset the counter for number of tags
without collision to Zero */
                              /* Because, a read cycle is done and
required to count tags again */
                              for ( i=0; i < tag_counter; i++)
                                    tag_ID[i] = -1;

                              reader_status = LISTEN;
                        }
                        FSM_PROFILE_SECTION_OUT (state2_exit_exec)


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

```
                  FSM_TRANSIT_FORCE (1, state1_enter_exec, ;,
"default", "", "TX", "START", "tr_20", "BFSA_Muting_reader [TX -> START
: default / ]")
                        /*---------------------------------------------
------------*/



                  /** state (LISTEN) enter executives **/
                  FSM_STATE_ENTER_FORCED (3, "LISTEN",
state3_enter_exec, "BFSA_Muting_reader [LISTEN enter execs]")
                        FSM_PROFILE_SECTION_IN ("BFSA_Muting_reader
[LISTEN enter execs]", state3_enter_exec)
                        {
                        printf("%s %g %s", "\nREADER: (LISTEN): Time:
", op_sim_time(), "\n");

                                                /* Read a packet from
the tags */

      pkin=op_pk_get(IN_STRM);

                                                /* If a packet is comes
from itself then destroy. */
                                                /* This model uses the
same Frequency for TX and RX */
                                                pkt_size =
op_pk_total_size_get (pkin);
                                                if ( pkt_size ==
RESPONSE_PKT_SIZE ) {

                                                if ( reader_status ==
LISTEN ) {

                                                      /* Check whether
a collision occurs or not */
                                                      number_collision
= 0;
                                                      number_collision
= op_td_get_int(pkin,OPC_TDA_RA_NUM_COLLS);
                                                      if (
number_collision != 0 )

      number_collision++;

      op_stat_write(SH_collisions_in_a_time,(double) number_collision);

                                                      if (
number_collision != 0 ) {     /* If a collision occurs  */

      number_of_collided_slots_per_frame++;

      collision_counter++;

      op_stat_write(SH_total_num_collisions,(double)
collision_counter);
                                                            printf("%s
%i %s %i %s", "1. Collision occurs. (Current:
",number_collision,"(tags), Total: ", collision_counter,"(times)) \n");
```

```
                                                        } else {
/* If a transmission is successful  */
      op_pk_nfd_get(pkin,"ID",&tag_ID[tag_counter]);
                                                              ID =
tag_ID[tag_counter];
      selected_ID[cnt] = ID;


      duplicated_tag = FALSE;
                                                              for ( i=0;
i <= cnt; i++ ) {
                                                                    if (
selected_ID[i] == ID-2 ) {
      duplicated_tag = TRUE;
      break;
                                                                    }
                                                                    if (
selected_ID[i] == ID+2 ) {
      duplicated_tag = TRUE;
      break;
                                                                    }
                                                              }

                                                              if (
duplicated_tag == FALSE ) {
                                                              printf("%s
%i %s", "1. A TAG (unique ID:",ID,") is identified.\n");


      op_stat_write_t(SH_selected_tag_ID,(double) ID,op_sim_time());
      number_of_identified_tags_per_frame++;
      count_selected_tag++;
      tag_counter++;
      cnt++;
                                                                    for(
i=1; i <= actual_number_of_tags; i++ ) {
      count_tag_id[i] =0;
                                                                    }
                                                                    for(
i=0; i < cnt; i++ ) {
      count_tag_id[selected_ID[i]]=count_tag_id[selected_ID[i]]+1;
                                                                    }
                                                              } else {
```

```
      printf("%s %i %s", "1. The non-unique tag (unique ID:",ID,") is
identified. \n");

      op_stat_write_t(SH_selected_tag_ID,(double) ID,op_sim_time());

      number_of_identified_tags_per_frame++;

      count_selected_tag++;

      tag_counter++;

      cnt++;
                                                                    non-
unique_tags++;

                                                                    for(
i=1; i <= actual_number_of_tags; i++ ) {

      count_tag_id[i] =0;
                                                                    }

                                                                    for(
i=0; i < cnt; i++ ) {

      count_tag_id[selected_ID[i]]=count_tag_id[selected_ID[i]]+1;
                                                                    }
                                                                 }
                                                              }

                                                           }
                                                           }
                                                           op_pk_destroy(pkin);
                           }
                           FSM_PROFILE_SECTION_OUT (state3_enter_exec)

                     /** state (LISTEN) exit executives **/
                     FSM_STATE_EXIT_FORCED (3, "LISTEN",
"BFSA_Muting_reader [LISTEN exit execs]")


                     /** state (LISTEN) transition processing **/
                     FSM_TRANSIT_FORCE (1, state1_enter_exec, ;,
"default", "", "LISTEN", "START", "tr_27", "BFSA_Muting_reader [LISTEN
-> START : default / ]")
                           /*---------------------------------------------
------------*/



                     }


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

```
void
_op_BFSA_Muting_reader_diag (OP_SIM_CONTEXT_ARG_OPT)
	{
	/* No Diagnostic Block */
	}



void
_op_BFSA_Muting_reader_terminate (OP_SIM_CONTEXT_ARG_OPT)
	{

	FIN_MT (_op_BFSA_Muting_reader_terminate ())


	/* No Termination Block */

	Vos_Poolmem_Dealloc (op_sv_ptr);

	FOUT
	}


/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in _op_BFSA_Muting_reader_svar function. */
#undef reader_status
#undef timeout
#undef collision_handle
#undef SH_collisions_in_a_time
#undef collision_counter
#undef SH_total_num_collisions
#undef SH_selected_tag_ID
#undef SH_total_num_selected_tag
#undef nocollision
#undef tag_ID
#undef started
#undef first_time
#undef new_variable
#undef tag_counter
#undef count_selected_tag
#undef selected_ID
#undef cnt
#undef A_READ_CYCLE_DURATION
#undef A_SLOT_DURATION
#undef readcycle_flag
#undef BFSA_READ_PKT_TX_TIME
#undef SELECT_PKT_TX_TIME
#undef unread_tags
#undef number_of_identified_tags_per_frame
#undef number_of_collided_slots_per_frame
#undef number_of_idle_slots_per_frame
#undef current_frame_size
#undef SH_number_of_identified_tags_per_frame
#undef SH_number_of_collided_slots_per_frame
#undef SH_number_of_idle_slots_per_frame
#undef SH_current_frame_size
```

```
#undef sum_of_frame_size
#undef sum_of_collided_slots_per_frame
#undef sum_of_idle_slots_per_frame
#undef SH_number_of_unread_tags_per_frame
#undef SH_total_delay
#undef SH_collision_delay
#undef SH_idle_delay
#undef sum_of_identified_tags_per_frame
#undef SH_success_delay
#undef EV_assurance_level
#undef SH_assurance_level
#undef assuranceLevel
#undef actual_number_of_tags
#undef frame_size
#undef SH_num_required_read_cycle
#undef SH_num_required_slots
#undef count_read_cycle
#undef total_slots
#undef non-unique_tags
#undef SH_non-unique_tags_count
#undef count_tag_id

#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

VosT_Obtype
_op_BFSA_Muting_reader_init (int * init_block_ptr)
      {
      VosT_Obtype obtype = OPC_NIL;
      FIN_MT (_op_BFSA_Muting_reader_init (init_block_ptr))

      obtype = Vos_Define_Object_Prstate ("proc state vars
(BFSA_Muting_reader)",
            sizeof (BFSA_Muting_reader_state));
      *init_block_ptr = 0;

      FRET (obtype)
      }

VosT_Address
_op_BFSA_Muting_reader_alloc (VosT_Obtype obtype, int init_block)
      {
#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      BFSA_Muting_reader_state * ptr;
      FIN_MT (_op_BFSA_Muting_reader_alloc (obtype))

      ptr = (BFSA_Muting_reader_state *)Vos_Alloc_Object (obtype);
      if (ptr != OPC_NIL)
            {
            ptr->_op_current_block = init_block;
#if defined (OPD_ALLOW_ODB)
            ptr->_op_current_state = "BFSA_Muting_reader [INIT enter
execs]";
#endif
```

```
            }
      FRET ((VosT_Address)ptr)
      }



void
_op_BFSA_Muting_reader_svar (void * gen_ptr, const char * var_name,
void ** var_p_ptr)
      {
      BFSA_Muting_reader_state            *prs_ptr;

      FIN_MT (_op_BFSA_Muting_reader_svar (gen_ptr, var_name,
var_p_ptr))

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

      if (strcmp ("reader_status" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->reader_status);
            FOUT
            }
      if (strcmp ("timeout" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->timeout);
            FOUT
            }
      if (strcmp ("collision_handle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->collision_handle);
            FOUT
            }
      if (strcmp ("SH_collisions_in_a_time" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_collisions_in_a_time);
            FOUT
            }
      if (strcmp ("collision_counter" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->collision_counter);
            FOUT
            }
      if (strcmp ("SH_total_num_collisions" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_total_num_collisions);
            FOUT
            }
      if (strcmp ("SH_selected_tag_ID" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_selected_tag_ID);
            FOUT
            }
      if (strcmp ("SH_total_num_selected_tag" , var_name) == 0)
            {
```

```
            *var_p_ptr = (void *) (&prs_ptr-
>SH_total_num_selected_tag);
            FOUT
            }
      if (strcmp ("nocollision" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->nocollision);
            FOUT
            }
      if (strcmp ("tag_ID" , var_name) == 0)
            {
            *var_p_ptr = (void *) (prs_ptr->tag_ID);
            FOUT
            }
      if (strcmp ("started" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->started);
            FOUT
            }
      if (strcmp ("first_time" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->first_time);
            FOUT
            }
      if (strcmp ("new_variable" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->new_variable);
            FOUT
            }
      if (strcmp ("tag_counter" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->tag_counter);
            FOUT
            }
      if (strcmp ("count_selected_tag" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->count_selected_tag);
            FOUT
            }
      if (strcmp ("selected_ID" , var_name) == 0)
            {
            *var_p_ptr = (void *) (prs_ptr->selected_ID);
            FOUT
            }
      if (strcmp ("cnt" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->cnt);
            FOUT
            }
      if (strcmp ("A_READ_CYCLE_DURATION" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->A_READ_CYCLE_DURATION);
            FOUT
            }
      if (strcmp ("A_SLOT_DURATION" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->A_SLOT_DURATION);
            FOUT
            }
```

```
      if (strcmp ("readcycle_flag" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->readcycle_flag);
            FOUT
            }
      if (strcmp ("BFSA_READ_PKT_TX_TIME" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->BFSA_READ_PKT_TX_TIME);
            FOUT
            }
      if (strcmp ("SELECT_PKT_TX_TIME" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SELECT_PKT_TX_TIME);
            FOUT
            }
      if (strcmp ("unread_tags" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->unread_tags);
            FOUT
            }
      if (strcmp ("number_of_identified_tags_per_frame" , var_name) ==
0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>number_of_identified_tags_per_frame);
            FOUT
            }
      if (strcmp ("number_of_collided_slots_per_frame" , var_name) ==
0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>number_of_collided_slots_per_frame);
            FOUT
            }
      if (strcmp ("number_of_idle_slots_per_frame" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>number_of_idle_slots_per_frame);
            FOUT
            }
      if (strcmp ("current_frame_size" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->current_frame_size);
            FOUT
            }
      if (strcmp ("SH_number_of_identified_tags_per_frame" , var_name)
== 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>SH_number_of_identified_tags_per_frame);
            FOUT
            }
      if (strcmp ("SH_number_of_collided_slots_per_frame" , var_name)
== 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>SH_number_of_collided_slots_per_frame);
            FOUT
            }
```

```
      if (strcmp ("SH_number_of_idle_slots_per_frame" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>SH_number_of_idle_slots_per_frame);
            FOUT
            }
      if (strcmp ("SH_current_frame_size" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_current_frame_size);
            FOUT
            }
      if (strcmp ("sum_of_frame_size" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->sum_of_frame_size);
            FOUT
            }
      if (strcmp ("sum_of_collided_slots_per_frame" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>sum_of_collided_slots_per_frame);
            FOUT
            }
      if (strcmp ("sum_of_idle_slots_per_frame" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>sum_of_idle_slots_per_frame);
            FOUT
            }
      if (strcmp ("SH_number_of_unread_tags_per_frame" , var_name) ==
0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>SH_number_of_unread_tags_per_frame);
            FOUT
            }
      if (strcmp ("SH_total_delay" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_total_delay);
            FOUT
            }
      if (strcmp ("SH_collision_delay" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_collision_delay);
            FOUT
            }
      if (strcmp ("SH_idle_delay" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_idle_delay);
            FOUT
            }
      if (strcmp ("sum_of_identified_tags_per_frame" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>sum_of_identified_tags_per_frame);
            FOUT
            }
      if (strcmp ("SH_success_delay" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_success_delay);
```

```
            FOUT
            }
      if (strcmp ("EV_assurance_level" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->EV_assurance_level);
            FOUT
            }
      if (strcmp ("SH_assurance_level" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_assurance_level);
            FOUT
            }
      if (strcmp ("assuranceLevel" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->assuranceLevel);
            FOUT
            }
      if (strcmp ("actual_number_of_tags" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->actual_number_of_tags);
            FOUT
            }
      if (strcmp ("frame_size" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->frame_size);
            FOUT
            }
      if (strcmp ("SH_num_required_read_cycle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>SH_num_required_read_cycle);
            FOUT
            }
      if (strcmp ("SH_num_required_slots" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_num_required_slots);
            FOUT
            }
      if (strcmp ("count_read_cycle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->count_read_cycle);
            FOUT
            }
      if (strcmp ("total_slots" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->total_slots);
            FOUT
            }
      if (strcmp ("non-unique_tags" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->non-unique_tags);
            FOUT
            }
      if (strcmp ("SH_non-unique_tags_count" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->SH_non-unique_tags_count);
            FOUT
            }
      if (strcmp ("count_tag_id" , var_name) == 0)
```

```
        {
        *var_p_ptr = (void *) (prs_ptr->count_tag_id);
        FOUT
        }
*var_p_ptr = (void *)OPC_NIL;

FOUT
}
```

VITA

Kirti Chemburkar was born in India on She received a Bachelor of Engineering in Computer Science from the University of Mumbai in 2005. She is expected to receive a Masters in Science in Computer and Information Science from the University of North Florida in August 2011. Kirti is currently employed as an application developer at JP Morgan Chase, where she develops and maintains web applications. Prior to that, she worked with St. John's County Board of County Commissioners for a period of two years as a .NET developer.

Kirti has extensive experience utilizing .NET technology for windows and web applications. She also has experience with a profound understanding of multiple languages that include C#, Java, JavaScript, XML, SQL, Oracle, VXML, CSS, C, C++, Visual Basic, and HTML.