Message Ferrying: Implementation and Simulation in OPNET

ENSC 427

Group 9

Dan Hendry (danh@sfu.ca), 301133878

Yazan Shehadeh (ysa5@sfu.ca), 301028275

Timbo Yuen (tty2@sfu.ca), 301023080

Simon Fraser University

School of Engineering Science

April 19, 2010

Course Instructor: Professor Ljiljana Trajkovic

Abstract

Machine to machine communication has long been considered a feature of the next technological age. For many applications, networking options available today are either too expensive or cumbersome to justify the information they are able to provide (such as dedicated wired Ethernet or cellular data modems), or do not provide uniform connectivity (such as WiFi). An alternate solution is to use ad-hoc mesh networking. Such networks however, require every node to be connect to another and fail when the network is sparse or becomes partitioned. Message ferrying is a technique which uses physical mobile devices, known as message ferries, as data transport mechanisms between disconnected network nodes or partitioned subnetworks. This report describes a message ferrying algorithm and simulation model for task oriented ferries created in OPNET which is applicable to a specialized remote sensor network in which a central repository maintains current sensor state. Update success rate and delay results are presented for two simulations.

Contents

List of Figures

1

Introduction

1.1 Background

Message ferrying is a networking approach where data is physically carried between network nodes which cannot communicate directly. It is sometimes called "store-carry-forward" routing [1]. Message ferries can be of two types; message oriented ferries and task oriented ferries [2]. Message oriented ferries are ferries dedicated to the task of transporting data, known in this context as messages, and their position is controlled by a ferrying algorithm. Task oriented ferries transport data but do so while performing another task. Their movement is not controlled by the ferrying algorithm. Much past research has focused on message oriented ferries within partitioned, wireless ad-hoc networks [1] [3] . Very little research has been found on the topic of task oriented ferries, the focus of this project. Related networking concepts are presented in section 1.1.1. Potential examples of applications suitable for message ferrying are presented in section 1.2.

1.1.1 Ad Hoc Network Types

Mobile Ad Hoc Networks

A mobile ad hoc network (MANET), is a self-configuring mesh network of mobile devices connected by wireless links [4]. These mobile devices are free to move independently in any direction and act as a router, where it must forward traffic unrelated to its own. Much past research on message ferrying has focused on MANETs.

Partitioned Networks

Partitioned networks are networks with no single hop or multiple hop route between some or even all node pairs. [2] In a partitioned network, nodes may remain fully disconnected or they may *cluster*, forming subnetworks in which all nodes are connected. All current used routing algorithms used in MANETs fail in the presence of partitioning [1]. This project assumes disconnected nodes, but does not assume nodes cluster.

Delay Tolerant Networks

A delay tolerant network is one in which routing strategies and applications must tolerate significant delays delivering packets. This delay may range from a few minutes up to hours or even days [1]. The network presented in this report is inherently delay tolerant.

1.1.2 Message Ferrying & Store-Carry-Forward Routing

Message ferrying is a technique where mobile nodes in a MANET buffer data and physically carry it between nodes which are unable to communicate directly [3]. Store-carry-forward routing is a strategy which makes use of, typically, known or assigned trajectories of these mobile nodes, known as message ferries [1]. Some messages are dropped if no route to the destination can be found [5].

1.2 Motivations and Potential Applications

With the significant number of mobile devices in use today, such as smartphones, laptops, tablets, netbooks, and more, there are many devices which could be potentially used as message ferries [6]. This project proposes one way to make use of the technology we transport with us on a daily basis. A message ferrying network could transport small amounts of data over large distances essentially for free. Beyond the use of message ferrying in remote sensor networks, discussed throughout this report, other applications of this technique might include tracking road traffic conditions, in-house utility management, automation for home devices, industrial monitoring, robot to robot communication and more [4].

1.3 Project Goals

Message ferrying has typically been examined within the context of improving throughput, reducing delay and increasing reliability within an ad hoc network [7]. Due to the complexity of incorporating message ferrying into existing ad hoc and MANET routing algorithms, this project will focus on a network in which data is transported strictly using ferries. No clustering of network nodes and routing within subnetworks will be considered (as discussed in section 1.1.1). Surprisingly, very little research has been found for a network with these characteristics. The goals of this project may be listed as follows.

- Design and implement a message ferrying algorithm.
- Simulate this algorithm in a highly partitioned network without node clustering or subnetworks.
- Evaluate the network using delay and message loss metrics.
- Examine the impact of node density, ferry count, and memory limits.

2

Project Premise and OPNET Model Design

This chapter presents the premise and details of a specialized network design used to analyze message ferrying in section 2.1. A general overview of the OPNET model which was created is then presented in section 2.2. Please refer to the appendix for specific details. Finally, the results of an initial simulation used to validate the node models is presented in 2.3.

2.1 Premise

This section outlines the requirements for any application which uses message ferrying. Details of a specialized 'state monitoring' network are then discussed.

2.1.1 Application Characteristics and Requirements

Any application making use of message ferrying must have the following characteristics:

Delay Tolerance: Since data is transported by a physical device, significant delays of minutes to hours must be expected.

Loss Tolerance: Given that ferries have limited memory, loss of data must be expected.

Small and Independent Messages: Following from the limited memory capacity of ferries and the high probability of data loss, a reliable method for segmentation and reassembly of messages should not be expected. Applications should limit the size of messages such that the can be transmitted in their entirety using one protocol data unit.

Given these criteria, a message ferrying network is unsuitable for many typical networking applications including web browsing, real-time voice or text communication and file transfer. As such, a very specialized 'state monitoring' network designed for non-critical monitoring of remote sensors is considered.

2.1.2 State Monitoring Network

The general premise for this project consists of a network containing numerous, uniquely identifiable source nodes. Each source node has a limited number of properties, in the form of key/value pairs, specifying a property name (the key) and its current value. Properties may change overtime and each change defines a new state for the source node. A temperature sensor for example, might support a 'temperature' property, the value of which is the current temperature updated every hour. Properties do not have to contain a single value and each may be as large as the payload limit of network packets.

The network and message ferrying algorithm is designed to synchronize a central repository with the current state of every source node. Only the most recent state (or most recent value) for each property is important, not the history of how that property has changed. This limits the number of packets which can exist in the network as only the most recent update must be reported. The message ferries collect data from source nodes when they are in range and transport it to the central repository. The central repository is assumed to be a server connected to the Internet. Ferries pass updates they have collected from source nodes to special gateway nodes. These gateway nodes are then responsible for using a reliable delivery mechanism over a standard IP network to update the central repository. This last stage is not considered for the implementation presented here. Once messages have been delivered to gateway nodes, they are assumed to have been delivered.

2.2 OPNET Model Design

Due to the lack of support in OPNET for message ferrying, all node and process models were creates specifically for this project. An overview of the basic network elements, including node and packets types, is presented in 2.2.1. A description of the networking algorithm is then presented in section 2.2.2. Please refer to the appendix for specific implementation details.

2.2.1 Network Elements

Network Nodes

The network is comprised of three types of network nodes:

Source Node: Static (non-mobile) nodes in the network which have a set of properties (key/value pairs). After a property of a source node changes, known as a state change, it attempt to notify the central repository via gateway nodes by transferring update packets to any message ferries which are in range. It is important to note that source updates may be delivered to any gateway. A source node could be, for example, a remote temperature sensor.

Message Ferry: Mobile nodes which collects updates from source nodes when they are in range. Message ferries store updates from source nodes within a buffer. When in range, these update packets are forwarded to gateway nodes. A source node could be, for example, a specially equipt cell phone or a small computer attached to a vehicle.

Gateway: Gateway nodes download update packets from message ferries and mark them as received.

Source nodes have properties (prop1, prop2, etc) which generate update messages. These messages get carried to *any* gateway node by message ferries. The OPNET node model for a source node node may be seen in 2.1

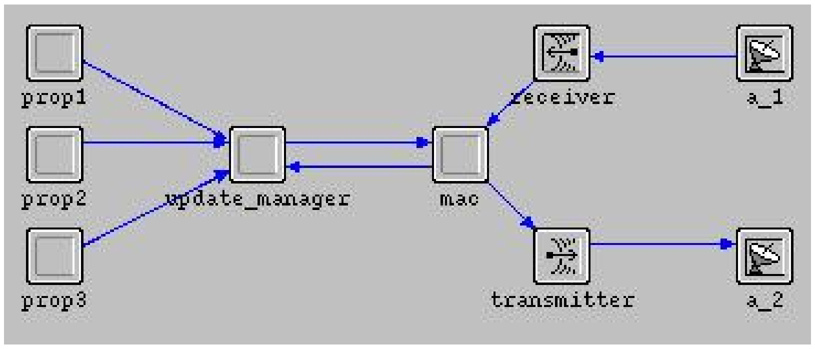


Figure 2.1: Source Node Model

The OPNET node model for a message ferry is shown in figure 2.2. The ferry is a mobile node which collects updates from source nodes when they are in range. These updates are then stored in memory (the storage process in figure 2.2). The storage process compares the updates, uniquely identified by source ID and property key, and keeps the most recent according to the key update number (see below).

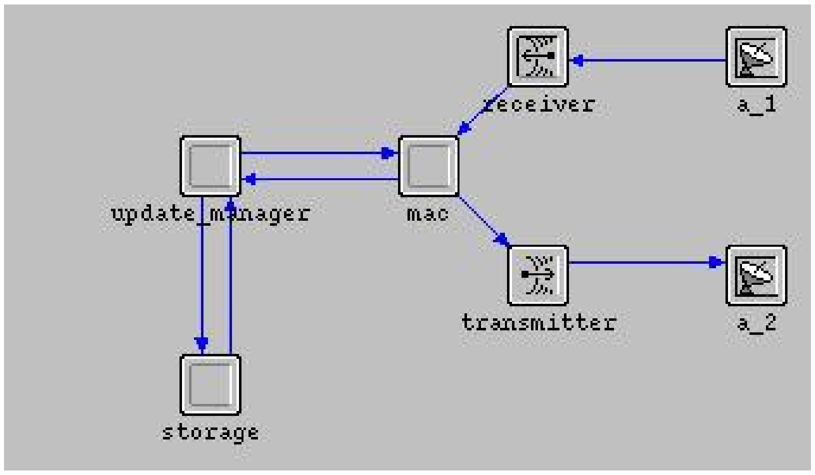


Figure 2.2: Ferry Node Model

The OPNET model for a gateway node is shown in figure 2.3. The gateway process model seen in the figure is responsible for tracking what updates have been received.

It makes use of global variables so updates may be received by any gateway.

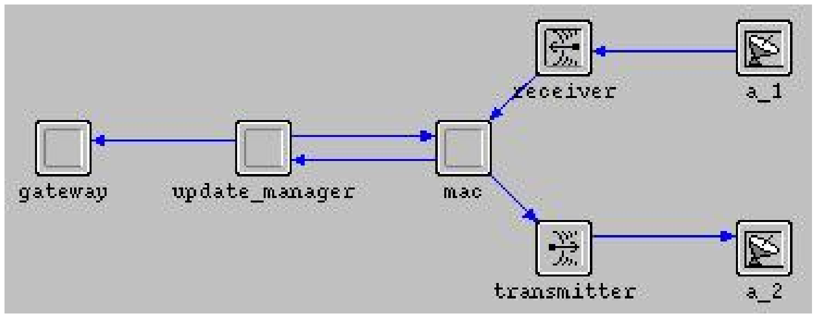


Figure 2.3: Gateway Node Model

Properties and Property Process

Each source node supports three properties as can be seen in figure 2.1. Furthermore, each property has three main pieces of data as described below. Updates generated by the property contain these three pieces of information. For the purposes of this project, the value of the property is inconsequential and not considered.

Source ID: A unique identifier of the source node a property is associated with

Key: Or property key is a unique for each property within a source node

Key Update Number: A counter which is incremented each time the property value changes. Only the most recent value of a property, as defined by its key update number, is of importance.

Packets

Two types of packets are used within the OPNET model.

Update Packet: Update packets are generated by the property processes of source nodes when their value changes. They are transmitted to message ferries which in turn transport them to the gateway.

Beacon Packet: Beacon packets are used to detect when nodes are in range and are able to communicate. Beacon packets are generated periodically be ferry and gateway nodes and trigger transmission of stored update messages by receiving nodes.

2.2.2 Algorithm and Behaviour

A brief overview of the algorithm implemented in each node is presented in this section. Pseudo code is presented here; refer to the appendix for the actual implementation.

Ferry Node Algorithm

Ferry nodes behave in the following way. Note that ferry nodes are mobile.

1. Periodically send beacons to notify other nodes that there is a ferry in range. These beacons trigger source nodes and other ferries to transmit stored updates.
2. When updates are received, store them. Updates are discard based on the following conditions.
   (a) If two updates (one received and one in memory) with the same source id and key are detected, discard the updated with the smallest key update number.
   (b) If the memory limit has been reached, discard the oldest update (regardless of source id and key).
3. When a beacon is received, transmit all updates in memory.
4. Repeat

Source Node Algorithm

Source nodes behave in the following way

1. Wait for a beacon.

2. Based on the current state, defined by the key update number for each property, transmit updates.
3. Repeat

Gateway Algorithm

Gateway nodes behave in the following way:

1. Periodically send beacons to notify other nodes that of the gateways existence.
2. Wait for updates to be received
3. If an update received for a given source id and key has a greater key update number than the last update received for that source id and key, record an update. If the key update number is equal to or less than the last update, discard the update.

2.2.3 Assumptions

In order to simplify the simulation and focus on the message ferrying algorithm, a number of assumptions were made regarding data transmission and wireless communication.

- Communication range of 60 meters. When nodes are closer than 60 meters they can communicate and when they are further apart than 60 meters, they cannot.
- No propagation and transmission delay. This assumption was made to ensure ferries receive all updates when moving past source nodes.
- No unintentional loss, the node link is assumed to be reliable. It is assumed that there is no loss caused by radio interference. This assumption was made to eliminate the need for an acknowledgment and retransmission mechanism.

These assumptions are considered valid as there are are number of technologies, implemented at a lower network layer, which provides reliable data transfer services.

2.3 Validation

This section presents, an initial simulation which was used to validate the OPNET node and process models.

2.3.1 Scenario Topology and Details

The network model is shown in figure 2.4. It is used to ensure the ferry receives updates from source nodes as it passes by them and transmits them to the gateway. There is one gateway node, one ferry node, and seven source nodes. The size of the map is 0.75 km x 0.75 km with source nodes placed evenly apart by 0.375 km. The gateway node is at the top left corner, and the ferry is in motion indicated next to the red arrow. The speed of the ferry is constant at 60 kmph, as it moves clockwise two times along the rectangular path that is highlighted in white. The simulated time was six minutes.

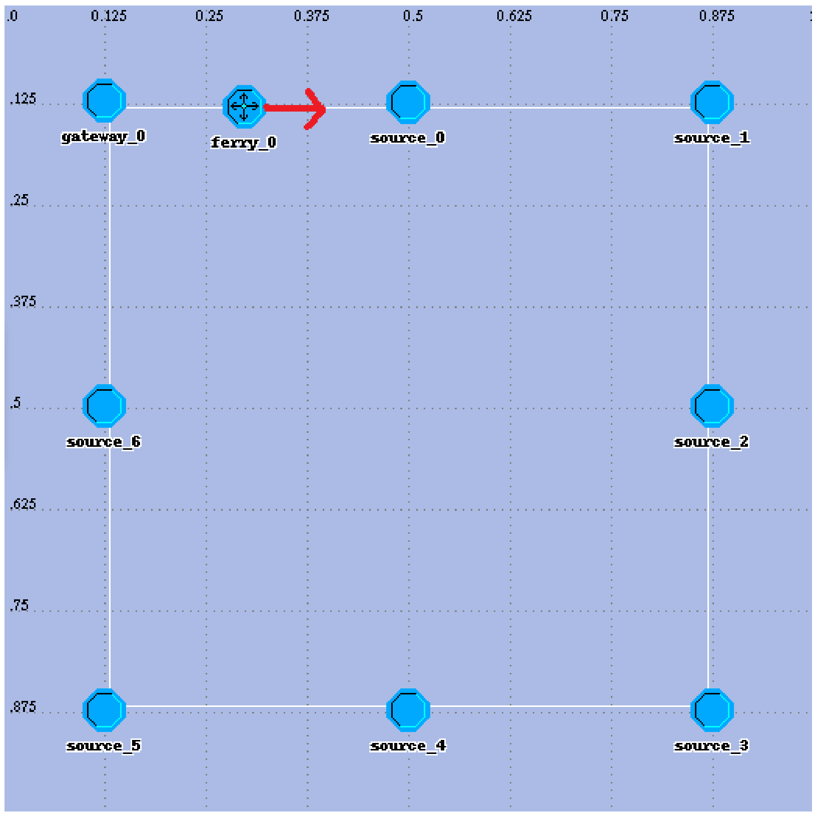


Figure 2.4: Network Model - Validation Simulation

2.3.2 Validation Simulation Results

A statistic measuring the number of updates received per second was created and set to be collected for the ferry and gateway nodes. The simulation was run and the results for the ferry are shown in figure 2.5. From it, we can see that the ferry is receiving updates each time it passes by a source node.

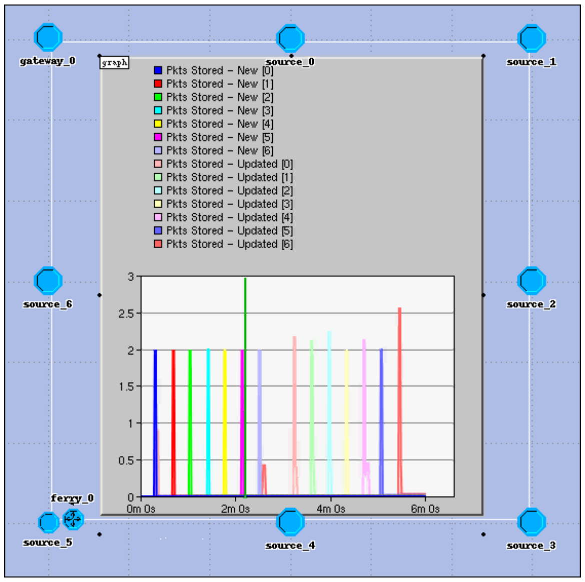


Figure 2.5: Update Packets Received by the Ferry Node

Results collected for the gateway may be seen in 2.6. From the figure, it is clear that there are two spikes in the graph which corresponds to the ferry transmitting the updates it has collected. The ferry node traverses its path twice in this simulation which is why there are two spikes. Each source node sends three update packets to the ferry node as it passes. Since there are seven source nodes, this accounts for the 21 packets received by the gateway node, which can be seen in figure 2.6.

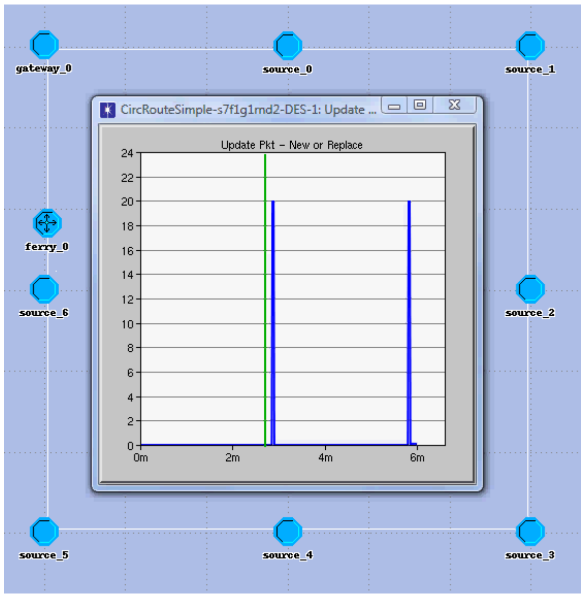


Figure 2.6: Gateway receives the packet as the ferry node passes by its range of transmission

3

Simulation

This chapter provides an overview of the two simulations which were created. These simulations are intended to be as realistic as possible and involve random movement of ferry nodes. The ferries were assumed to be vehicles which defined their speed. A number of scenarios were tested for each simulation, the parameters varied are explained in section 3.2.

3.1 Network Model

This section outlines the two network models defining each simulation. Their difference lies on the number of ferry and gateway nodes. The parameters common between each are explained in section 3.1.3.

3.1.1 Scenario 1

The first simulation was created with one gateway node, one ferry node, and ten source nodes. The network model may be seen in figure 3.1. The gateway is placed in the center of the map and the ferry node starts next to it. Source nodes are more or less evenly distributed and are placed such that they are out of direct communication range.

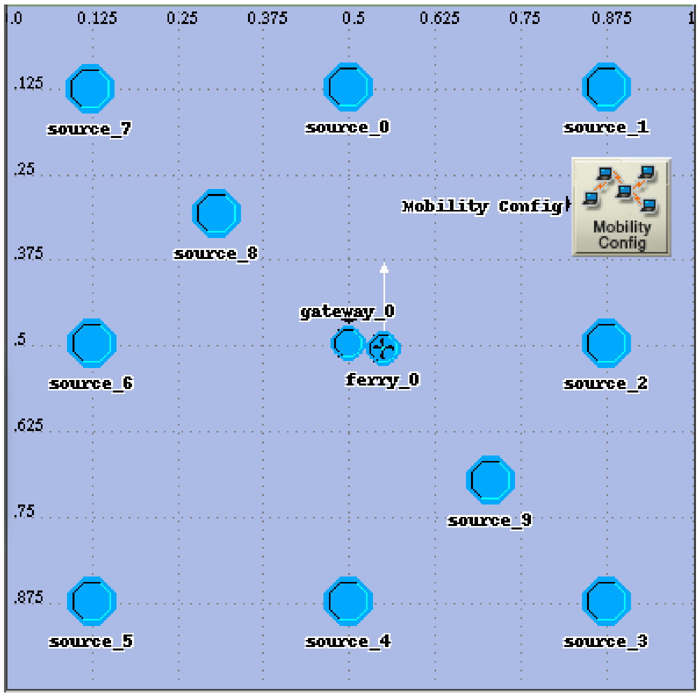


Figure 3.1: Simulation 1 - Network Model (1 Gateway, 1 Ferry)

3.1.2 Scenario 2

The second simulation was created with two gateway nodes, two ferry node, and ten source nodes. The network model may be seen in figure 3.2. The gateways are placed in opposing quadrants, while both ferries start from the center. Source nodes and gateways are more or less evenly distributed and are placed such that they are out of direct communication range.

3.1.3 Common Settings

Some of the settings and characteristics common to all the topologies are the following:

- All ferries move in random directions and have a varying speeds of 36kph - 72kph in uniform distribution.
- The size of both maps is 1km x 1km
- Properties are updated every 2 seconds with a variance of 0.1 seconds

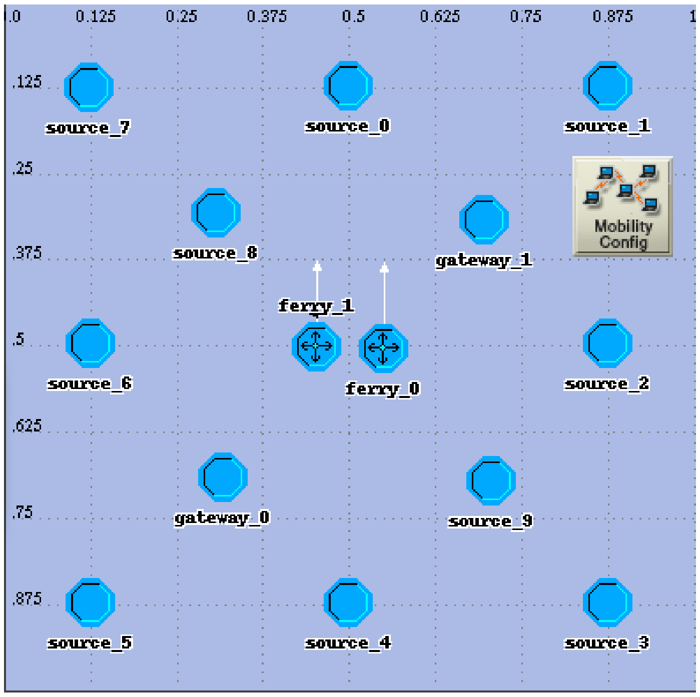


Figure 3.2: Simulation 2 - Network Model (2 Gateways, 2 Ferries)

- Simulations are run for 90 minutes. Property updates were disabled for the last thirty minutes in an effort to obtain statistics valid for a simulation of indefinite length.

3.2 Metrics and Results of Interest

Two metrics are of primary interest when analyzing the network, update success rate and delay.

3.2.1 Update Success Rate

Update success rate, or alternatively update loss, is of central importance in the network. It is primarily affected by memory limits imposed by the ferry but is also affected by the number of ferry and gateway nodes. Defining success rate is somewhat complicated as updates may be intentionally discarded before they reach a gateway if they are out of date. Additionally, updates may be duplicated multiple time as ferries exchange messages. Finally, discarding updates with a key update number less than the most recent key update number received by the gateway is desired behaviour. As such, the following conditions are used to determine success rate which is measured as *success*, *failure* or *no value* for each key update generated by every source node property.

- For updates which reach the gateway:
    - If the update has a key update number greater than the last update received by the gateway (for a given source id and property key) the update counts as a *success*.
    - If the update has a key update number equal to or less than the last update received by the gateway (for a given source id and property key) the update is not considered and counts as *no value*.
- For updates which do not reach the gateway and are discard by every ferry node:
    - If the update has a key update number greater than the last update received by the gateway (for a given source id and property key) the update counts as a *failure*.
    - If the update has a key update number equal to or less than the last update received by the gateway (for a given source id and property key) the update is not considered and counts as *no value*.

3.2.2 Delay

The number of ferries and gateways is the primary parameter affecting delay, however, ferry memory limits also play a role. Delay is defined as the time an update takes to reach the gateway after it has been generated by a source node. Only updates which are successfully delivered (as defined in section 3.2.1) count towards delay. As such, it is important for results of delay to be considered within the context of update success rate. It should be noted that the lower bound on delay is the time it physically takes the ferry to move between the source node and gateway.

3.2.3 Simulation Parameters Varied

The following parameters were varied to create additional scenarios for each simulation as presented in section 3.1. Their impact on success rate and delay (from sections 3.2.2 and 3.2.1) were considered.

Memory Limit

The memory limit, also referred to as capacity, is the buffer size of the ferry. It limits the number of unique updates which can be stored at once. It is set in number of updates, not bytes, and hence is somewhat unrealistic. It is sufficient for the purposes of this scenario however.

Seed - Effect of Randomness

Since ferry movement is random and the number of gateways is limited, it is important to consider multiple random when simulating in OPNET.

Source Node Storage

The node models and algorithm presented thus far has assumed that only ferries store updates. A modified node model which allows source nodes to store updates was also considered. The effect of enabling source node storage was examined.

4

Results

Simulation results are presented in terms of success rate (section 4.1) and delay (section 4.2).

4.1 Success Rate

As discussed in section 3.2.1, memory limit is the most important parameter affecting success rate.

4.1.1 Simulation 1

The update success rate for the first simulation with one ferry and one gateway is shown in 4.1. Success rate is shown as a function of ferry memory capacity. Results for two separate seeds are shown since ferry movement, and hence delivery times and success rate, is heavily affected by randomness. The spike in success rate shown for the first simulation (seed of 128) at a memory capacity of four is an artifact of this randomness. It can be seen from this figure that success rate increases rapidly with memory capacity. There is a leveling effect seen when memory capacity increases beyond 30. Since there are ten source nodes each with three properties and the storage process is intelligent about keeping only the most current updates, no packets must be discarded (see section 2.2.2). The fact that success rate never reaches 1, or

100%, is an artifact of the limited simulation time. Were the simulation to run forever and ferries to visit every source node, success rate would approach 1.

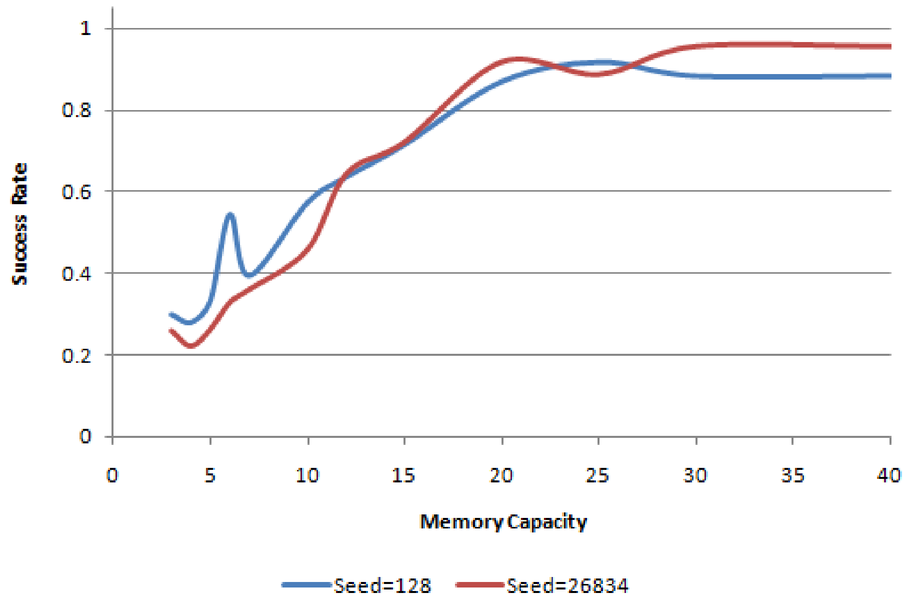


Figure 4.1: Success Rate vs Memory Capacity - Simulation 1 (1 Ferry, 1 Gateway), Source Storage Disabled

4.1.2 Simulation 2

The update success rate for second simulation, with two ferries and two gateways, is shown in figure 4.2. As in section 4.1.1, success rate is shown as a function of ferry memory capacity and results for two separate seeds are shown. It can be seen that variability in the success rate between the two seed values is lower than the first scenario. Increasing the number of ferries and gateways increases the likely hood a ferry will pass by the node and decreases variance.

4.1.3 Comparison of Success Rate

A comparison of success rate between simulations 1 and 2 is show in figure 4.3. It can be seen the additional ferry and gateway significantly increase success rate; this result is expected. As discussed in section 4.1.1, the success rate should be 1 for memory

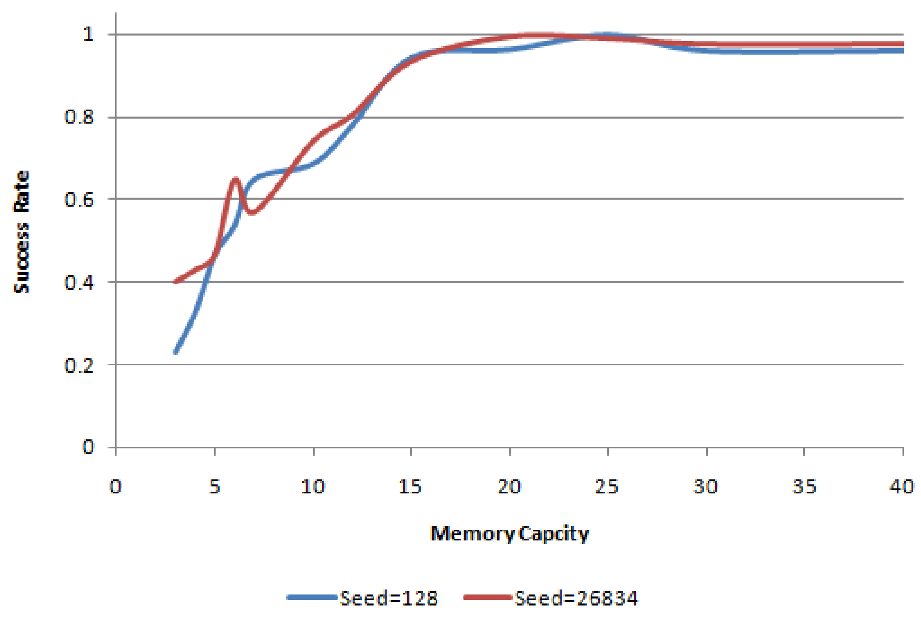


Figure 4.2: Success Rate vs Memory Capacity - Simulation 2 (2 Ferries, 2 Gateways), Source Storage Disabled

capacities beyond 30. The fact that it is not is a result of limited simulation time.

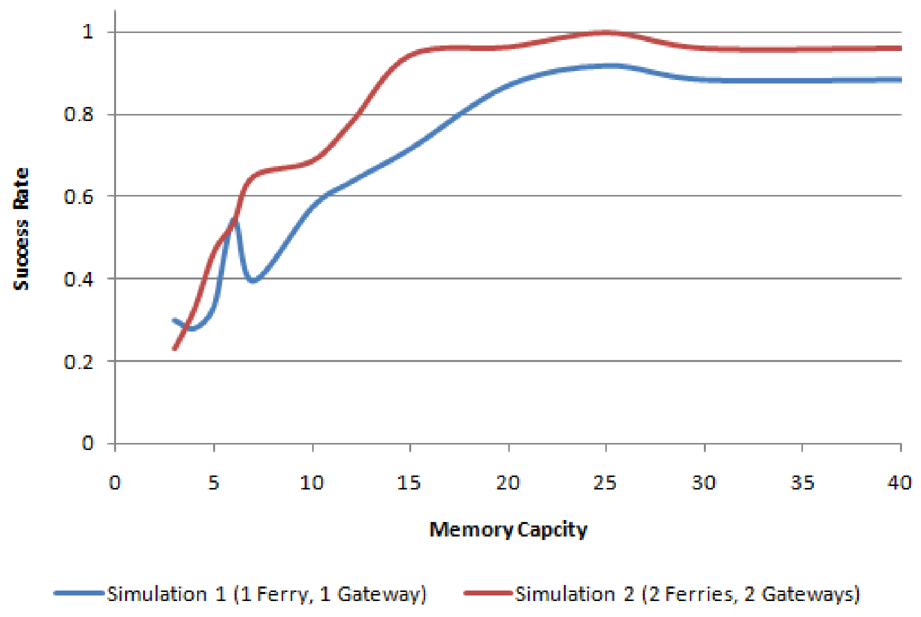


Figure 4.3: Success Rate vs Memory Capacity - Simulation 1 and 2 , Seed of 128 and Source Storage Disabled

4.1.4 Effect of Source Node Storage

As discussed in section 3.2.3, source nodes can be configured to receive updates from ferries, store them and retransmit them; in essence, acting as stationary ferries. The impact of enabling source node storage on success rate in simulation 2 can be seen in 4.4. It can be seen that although success rate increased, the change was not drastic. This result is somewhat expected given that there were only two ferries and all source nodes were relatively close to gateways. It is expected that the impact of enabling source node storage would become more pronounced by increasing the number of ferries or increasing the distance between source nodes and gateways while keeping the relative spacing of source nodes constant Enabling source node storage in simulation 1 was seen to have no impact on success rate. This is expected as the algorithm used to discard packets in the presence of memory constrains would prevent a ferry re-storing up an update it had previously discarded.

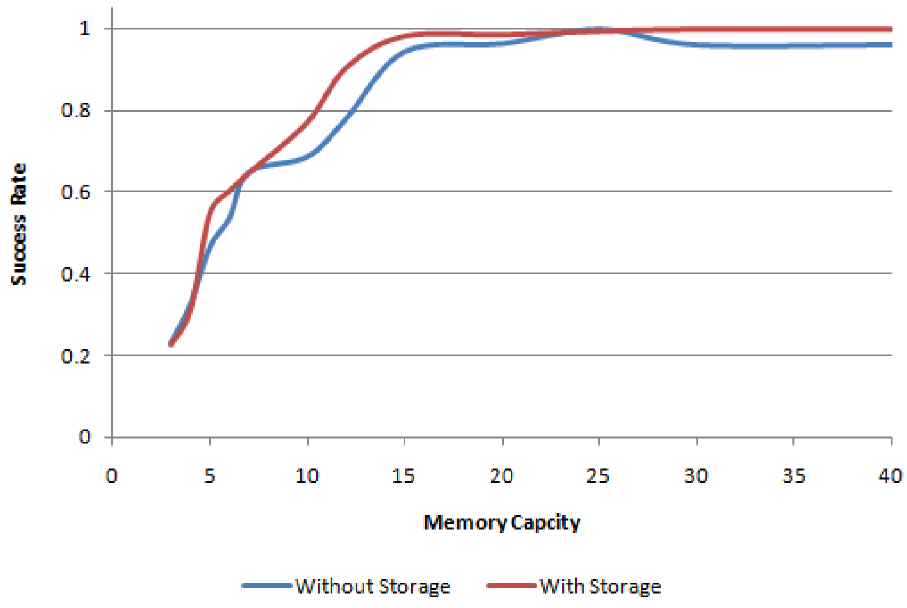


Figure 4.4: Effect of Source Node Storage - Success Rate vs Memory Capacity - Simulation 2, Seed of 128

4.2 Delay

The time between a property value changing and when that change is registered by the gateway is of interest and examined. Delay is presented as a cumulative distribution function. A CDF was chosen over a PDF as it is more meaningful in the presence of random ferry movement. It is important to note that delay results presented here only account for updates successfully delivered. Updates lost as defined in section 3.2.1, with the exception of the very last update, do not count towards the measured delay. As such, results showing greater delay may not indicate the given scenario has better performance. These results must be considered within the context of loss as presented in section 4.1.

4.2.1 Simulation 1

Figures 4.5 and 4.6 show the delay for simulation 1 (1 ferry and 1 gateway) with memory limits of 3 and 30 updates imposed by the ferry respectively. Two seeds are shown for each in order to illustrate the effects of randomness. When memory is limited (shown in figure 4.5), it can be seen more than half of all updates are delivered within 400 seconds. This result is somewhat misleading as the success rate is very poor for this memory setting (see section 4.1.1). The results presented in figure 4.6 are far more meaningful. A gradual increase in the delay is seen with approximately 80% of updates being delivered within 500 seconds.

4.2.2 Simulation 2

Figures 4.7 and 4.8 show the delay for simulation 2 with memory limits of 3 and 30 updates imposed by the ferries respectively. As in section 4.1.2, randomness (as distinguished between the two seed values) has less effect on the output than simulation 1. Comparing the memory constrained cases between simulations (figures 4.5 and 4.7), it is clear that the second simulation has reduced delay and increased success rate. It can be seen from figure 4.8 that 90% of updates have a delay of approximately 300 seconds when memory is not constrained.

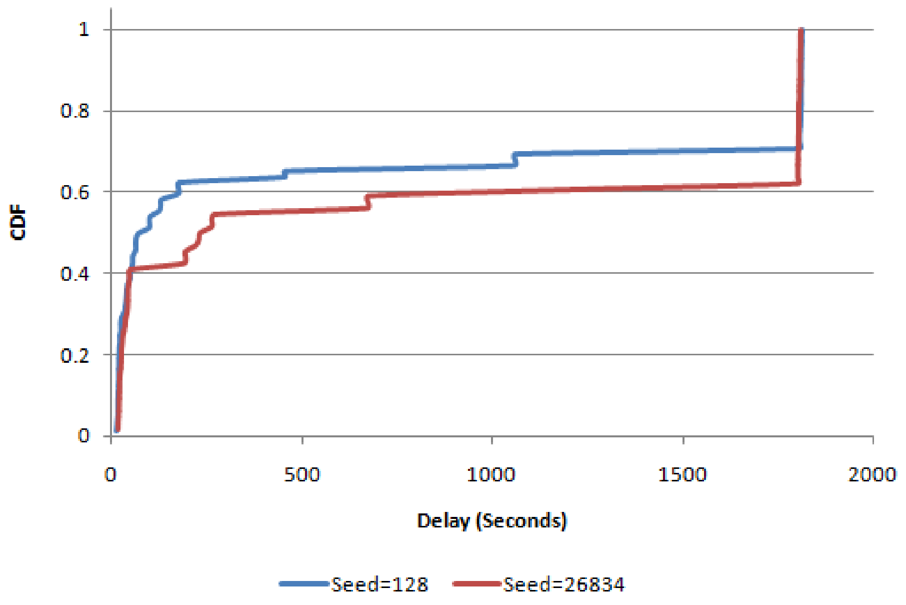


Figure 4.5: Delay - Simulation 1 (1 Ferry, 1 Gateway), Memory Capacity of 3

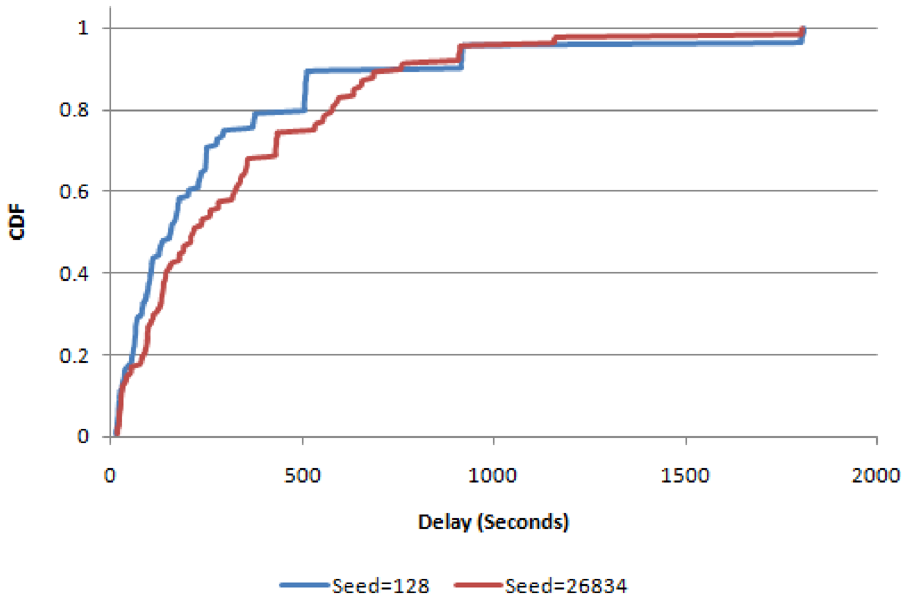


Figure 4.6: Delay - Simulation 1 (1 Ferry, 1 Gateway), Memory Capacity of 30

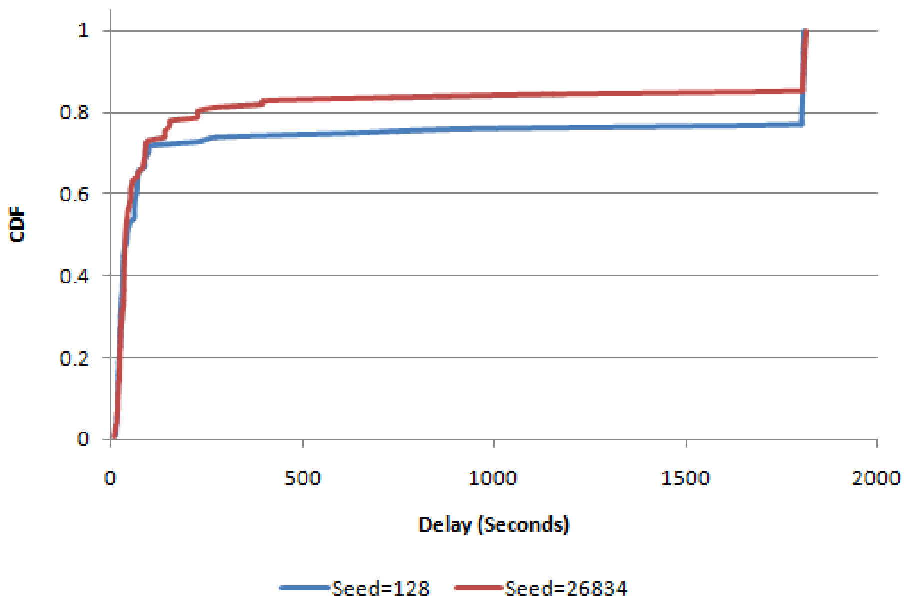


Figure 4.7: Delay - Simulation 2 (2 Ferries, 2 Gateways), Memory Capacity of 3

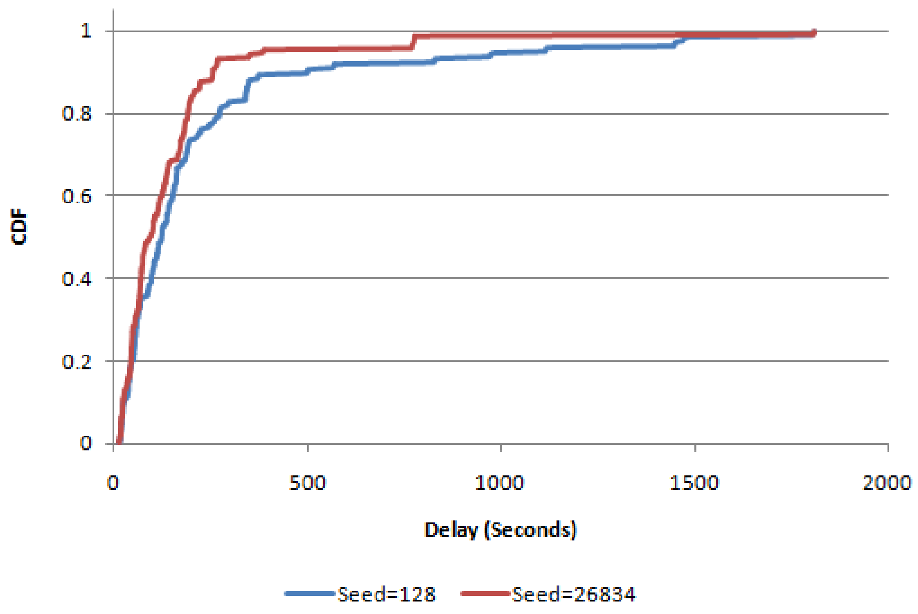


Figure 4.8: Delay - Simulation 2 (2 Ferries, 2 Gateways), Memory Capacity of 30

4.2.3 Comparison of Simulation

Figures 4.9 and 4.10 provide a comparison of delay between the two scenarios. It can be seen that the second scenario, with an additional ferry and gateway, has significantly reduced average delay. Furthermore, very few updates take longer than approximately 300 seconds to reach the gateway. When ferry memory capacity is limited to 5 updates, the additional ferry and gateway is seen to increase the probability updates are received in a timely manner.

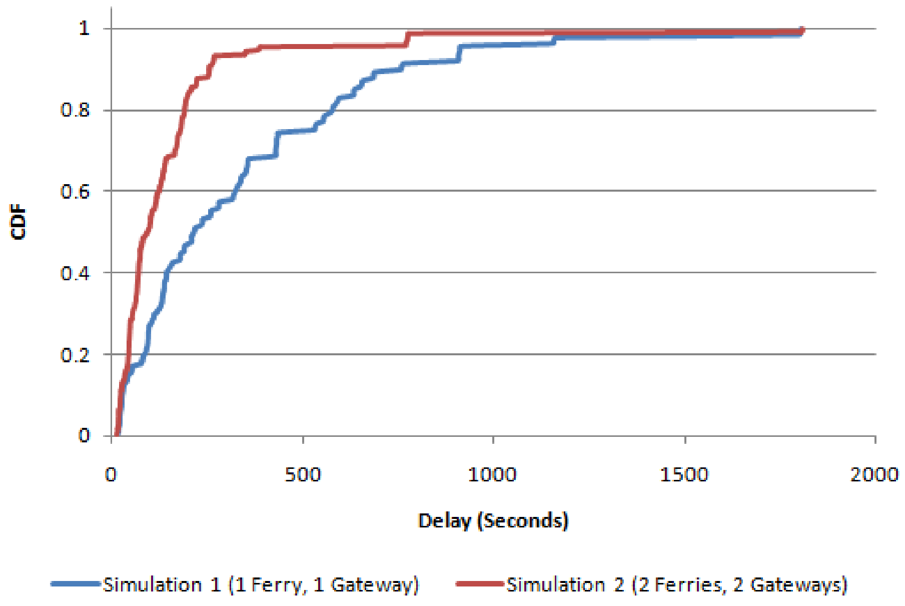


Figure 4.9: Delay - Simulation 1 and 2, Memory Capacity of 30, Seed of 26834

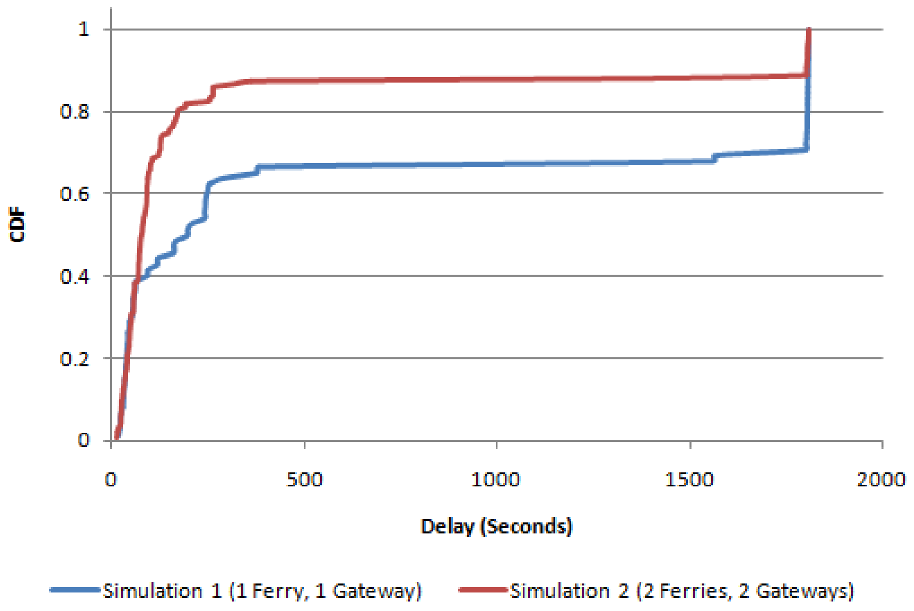


Figure 4.10: Delay - Simulation 1 and 2, Memory Capacity of 5, Seed of 128

5

Conclusion

First and foremost, OPNET has been shown to be a suitable tool for analyzing message ferrying. The node models created to analyze the specialized 'state monitor' network were tested and validated. A more complicated examination was performed with a network model involving ten source nodes and varying numbers of ferry and gateway nodes. Statistics measuring update success and delay were defined, implemented and collected during an OPNET simulation.

5.1 Results

A number of general conclusions can be drawn from the results presented in section 4. Adding gateways and ferries was seen to reduces delay, reduces the memory requirements of ferries to achieve a desired success rate, and decreases variability in delay (see section 4.1.2). As such, any message ferrying network should have a maximum number of ferries and gateways. The success rate was seen to marginally improve when enabling source storage. This improvement is expected to increase with additional ferries. As such, it may be concluded that networks with few ferries need not implement this feature, however it should be enabled for networks with many ferries.

5.2 Future Work

There are four main categories for future work and improvements to the OPNET model in order to study task oriented message ferrying.

5.2.1 Algorithm Improvements

Many aspects of the ferrying algorithm implemented in this network are simplistic. For example, there is no reverse communication from the gateway to message ferries indicating updates have been delivered and update messages may be discarded. The implementation of an update acknowledgment mechanism could significantly improve performance and memory utilization. Additionally, a more intelligent algorithm used by ferries to discard packets could also improve overall network performance.

5.2.2 Model Improvements

Many assumptions and simplifications were made when considering update and data transfer between nodes. For example, near instantaneous data transfer, little to no packet loss, and a strict communication range of 60 meters was assumed. Incorporating an existing point to point protocol for reliable wireless data transfer, such as WiFi or ZigBee would provide more realistic results.x

5.2.3 Statistic Improvements

Due to the unique nature of the network, common ways to measure statistics are not valid. As such, custom logic was required to produce all statistics. Measurement of only two statistics was implemented, delay and update success rate. Adding additional statistics, such as number of active packets in the network and arrival order, would provide additional insights into the networks behaviour.

5.2.4 Applicability and Network Model

The simulations that were presented involves roughly even source node placement and random ferry movement. It is unlikely that a real network would have these characteristics. Creating and simulating a real-world network model and application would provide more realistic results.

References

[1] R. Patra, K. Fall, and S. Jain, “Routing in a delay tolerant network,” in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, 2004.

[2] Y. Chen, W. Zhao, M. Ammar, and E. Zegura, “Hybrid routing in clustered DTNs with message ferrying,” in *Proceedings of the 1st international MobiSys workshop on Mobile opportunistic networking*, College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332, 2007.

[3] W. Zhao, M. Ammar, and E. Zegura, “A message ferrying approach for data delivery in sparse mobile ad hoc networks,” in *Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing*, College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332, 2004.

[4] S.-L. Wu and Y.-C. Tseng, *Wireless Ad Hoc Networking.* Auerbach Publications, 2007, pp. 439–459.

[5] T. Massey, “Message Ferry Architecture and Implementation,” in *Georgia Institute of Technology*, 2004, http://www.cs.ucla.edu/~tmassey/MF_Master_Proj.pdf.

[6] J. A. Davis, A. H. Fagg, and B. N. Levine, “Wearable Computers as Packet Transport Mechanisms in Highly-Partitioned Ad-Hoc Networks,” in *In Proceedings of the 5th IEEE international Symposium on Wearable Computers*, IEEE Computer Society, Washington, DC, 141, October 2001.

[7] R.C.Suganthe and P.Balasubramanie, "Efficient Routing For Intermittently Connected Mobile Ad hoc Network," in *IJCSNS International Journal of Computer Science and Network Security, 184 VOL.8 No.11, November 2008*, 2008, http://paper.ijcsns.org/07_book/200811/20081126.pdf.

Appendix A

Code: Source Property Process

A.1 Overview

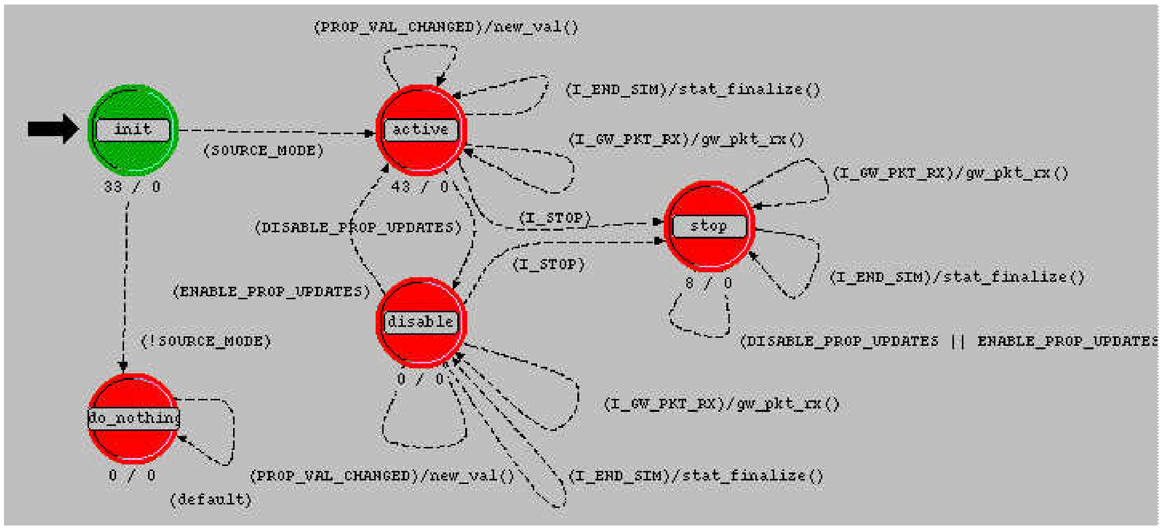
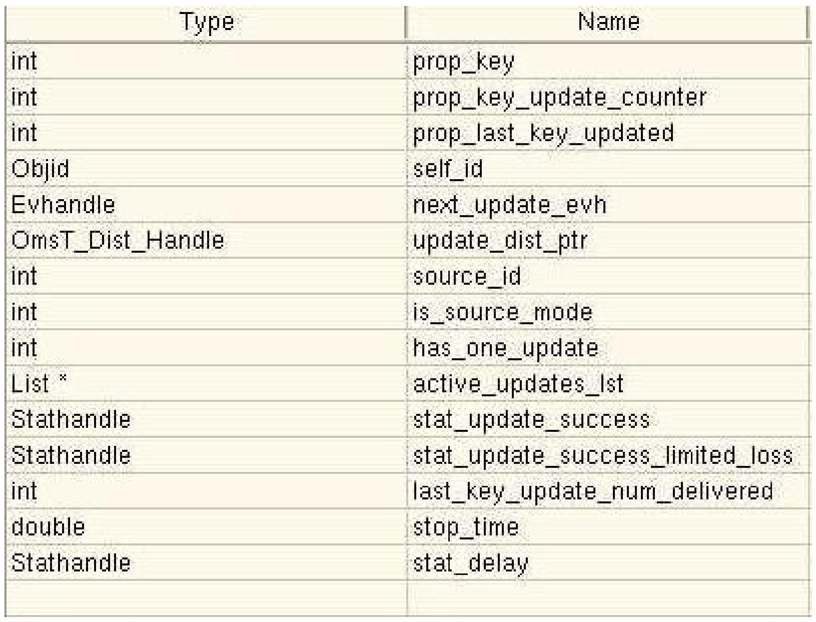

(PROP_VAL_CHANGED)/new_val()
(I_END_SIM)/stat_finalize()
init
active
(SOURCE_MODE)
(I_GW_PKT_RX)/gw_pkt_rx()
33 / 0
43 / 0
(I_GW_PKT_RX)/gw_pkt_rx()
(DISABLE_PROP_UPDATES)
(I_STOP)
stop
(I_STOP)
(I_END_SIM)/stat_finalize()
(ENABLE_PROP_UPDATES)
8 / 0
disable
(!SOURCE_MODE)
(DISABLE_PROP_UPDATES || ENABLE_PROP_UPDATES
0 / 0
do_nothing
(I_GW_PKT_RX)/gw_pkt_rx()
0 / 0
(PROP_VAL_CHANGED)/new_val()
(I_END_SIM)/stat_finalize()
(default)

Figure A.1: Source property process model

A.2 Local variables



Type	Name
int	prop_key
int	prop_key_update_counter
int	prop_last_key_updated
Objid	self_id
Evhandle	next_update_evh
OmsT_Dist_Handle	update_dist_ptr
int	source_id
int	is_source_mode
int	has_one_update
List *	active_updates_lst
Stathandle	stat_update_success
Stathandle	stat_update_success_limited_loss
int	last_key_update_num_delivered
double	stop_time
Stathandle	stat_delay

Figure A.2: State variables of source property process

A.3 Header Block

```
#include <oms_dist_support.h>

//Interrupt Codes (codes have no meaning and are random)
#define IC_PROP_VAL_CHANGED  39
#define IC_UPDATES_DISABLE   83
#define IC_UPDATES_ENABLE 84
#define IC_STOP 21
#define IC_GW_PKT_RX 99
#define SOURCE_MODE (is_source_mode)

//Interrupts
#define PROP_VAL_CHANGED (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() == IC_PROP_VAL_CHANGED)
#define DISABLE_PROP_UPDATES (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() == IC_UPDATES_DISABLE)
#define ENABLE_PROP_UPDATES (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() == IC_UPDATES_ENABLE)
#define I_GW_PKT_RX (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() == IC_GW_PKT_RX)
#define I_END_SIM (op_intrpt_type() == OPC_INTRPT_ENDSIM)
#define I_STOP (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() == IC_STOP)

typedef struct
{
int update_counter_number;
int pkts_alive;
int has_one_store;
int gateway_rx;
double generated_timestamp;
int mark_for_delete;
int discard_reason;
} active_update_tacker;

void new_val(void);
void schedule_update(void);
void gw_pkt_rx(void);
void stat_finalize(void);
```

A.4 Function Block

```
void new_val(void)
{
FIN (new_val ());
prop_key_update_counter++;
schedule_update();
FOUT
}
void schedule_update(void)
{
double next_update_time;
FIN(schedule_update());
next_update_time = oms_dist_outcome (update_dist_ptr);

if (next_update_time <0)
{
next_update_time = 0;
}

next_update_evh      = op_intrpt_schedule_self (op_sim_time () + next_update_time, IC_PROP_VAL_CHANGED);
FOUT;
}
void gw_pkt_rx()
{
Ici *iciptr;
int sourceid;
int key_update_number;
int action;
int discard_reason;
int tracker_index;
double generated_timestamp;
active_update_tacker *pTracker;
char msg[255];
int i;

FIN(gw_pkt_rx());
iciptr = op_intrpt_ici ();

if(iciptr == OPC_NIL)
{
op_sim_end("Null ICI", "", "", "");
}
op_ici_attr_get (iciptr, "source_id", &sourceid);
op_ici_attr_get (iciptr, "key_update_number", &key_update_number);
op_ici_attr_get (iciptr, "action", &action);
op_ici_attr_get (iciptr, "discard_reason", &discard_reason);
op_ici_attr_get (iciptr, "generated_timestamp", &generated_timestamp);

//Basic field checks
```

```
if(sourceid != source_id)
{
op_sim_end("Bad source id for ICI", "", "", "");
}
else if(key_update_number > prop_key_update_counter || key_update_number <= 0)
{
op_sim_end("Bad key_update_number for ICI", "", "", "");
}
//Find the tracker
pTracker = OPC_NIL;
for(tracker_index = 0; tracker_index < op_prg_list_size(active_updates_lst); tracker_index++)
{
active_update_tacker *pTrackerTemp;
pTrackerTemp = (active_update_tacker *)op_prg_list_access(active_updates_lst, tracker_index);

if(pTrackerTemp->update_counter_number == key_update_number)
{
pTracker = pTrackerTemp;
break;
}
}
if(pTracker == OPC_NIL)
{
int i;
char msg1[255];
char msg2[255];
char msg3[255];
printf("Current list state\n");

for(i = 0; i < op_prg_list_size(active_updates_lst); i++)
{
active_update_tacker *pTrackerTemp;
pTrackerTemp = (active_update_tacker *)op_prg_list_access(active_updates_lst, i);
sprintf(msg1, "update_counter_number=%d\n", pTrackerTemp->update_counter_number);
printf(msg1);
}
sprintf(msg1, "Tracker index=%d, List size=%d", tracker_index, op_prg_list_size(active_updates_lst));
sprintf(msg2, "key_update_number=%d, prop_key_update_counter=%d", key_update_number, prop_key_update_counter);
sprintf(msg3, "Action=%d", action);
op_sim_end("Could not find tracker", msg1, msg2, msg3);
}
if(action == 1)
{
//Gateway received packet
if(pTracker->gateway_rx)
{
op_sim_end("Two gateway rx interrupts", "", "", "");
}
else if(pTracker->pkts_alive <= 0)
{
op_sim_end("pkts_alive problem", "", "", "");
}
pTracker->gateway_rx = 1;
op_stat_write(stat_delay, op_sim_time() - generated_timestamp);
op_stat_write(stat_update_success, 1.0);
op_stat_write(stat_update_success_limited_loss, 1.0);

if(last_key_update_num_delivered >= key_update_number)
{
op_sim_end("Problem with gateway", "", "", "");
}
```

```
last_key_update_num_delivered = key_update_number;
}
else if (action == 2)
{
//Store
if(pTracker->pkts_alive < 0)
{
op_sim_end("Thats strange...", "", "", "");
}
pTracker->pkts_alive++;
pTracker->has_one_store = 1;
}
else if (action == 3)
{
//Discard
pTracker->pkts_alive--;
pTracker->discard_reason = discard_reason;
}
else
{
op_sim_end("Bad action", "", "", "");
}
has_one_update = 1;

for(i = 0; i < op_prg_list_size(active_updates_lst); i++)
{
pTracker = (active_update_tacker *)op_prg_list_access(active_updates_lst, i);

if(pTracker->pkts_alive <= 0)
{
pTracker->mark_for_delete--;
}
if(pTracker->mark_for_delete <= 0)
{
active_update_tacker *pTrackerTemp;

if(pTracker->pkts_alive < 0 &&  pTracker->has_one_store)
{
op_sim_end("Thats strange...", "2", "", "");
}
if(pTracker->gateway_rx == 0)
{
//Nothing left - update has been lost
op_stat_write(stat_update_success, 0.0);

if(pTracker->discard_reason == 1)
{
//Update
}
else if(pTracker->discard_reason == 2)
{
//Mem full
if(pTracker->update_counter_number > last_key_update_num_delivered)
{
op_stat_write(stat_update_success_limited_loss, 0.0);
}
}
else
{
op_sim_end("Bad discard reason", "", "", "");
}
}
pTrackerTemp = (active_update_tacker *)op_prg_list_remove(active_updates_lst, i);

if(pTrackerTemp != pTracker)
{
op_sim_end("AHA,not another error!", "", "", "");
}
op_prg_mem_free(pTracker);
}
}
op_ici_destroy(iciptr);
FOUT;
}
void stat_finalize()
{
Stathandle oneup;
int i;

FIN(stat_finalize());
oneup = op_stat_reg("One Update",OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
op_stat_write(oneup, has_one_update);

for(i = 0; i < op_prg_list_size(active_updates_lst); i++)
{
active_update_tacker *pTracker;
pTracker = (active_update_tacker *)op_prg_list_access(active_updates_lst, i);

if(pTracker->gateway_rx == 0)
{
//Receiving should already have been taken care of
op_stat_write(stat_update_success, 0.0);

if(pTracker->update_counter_number > last_key_update_num_delivered)
{
op_stat_write(stat_update_success_limited_loss, 0.0);
}
}

if(pTracker->update_counter_number == prop_last_key_updated)
{
if(pTracker->gateway_rx == 0)
{
op_stat_write(stat_delay, op_sim_time() - pTracker->generated_timestamp);
}
}
}
FOUT;
}
```

A.5 init State: Enter Executives

```
char msg[100];
char updatedist_str [128];
self_id = op_id_self();
op_ima_obj_attr_get (self_id, "Source ID", &source_id);
op_ima_obj_attr_get (self_id, "Property Key", &prop_key);
op_ima_obj_attr_get (self_id, "Property Update Interval", updatedist_str);
op_ima_obj_attr_get (self_id, "Stop Time", &stop_time);
op_ima_obj_attr_get (self_id, "Enable Properties", &is_source_mode);
active_updates_lst = op_prg_list_create();
has_one_update = 0;
prop_key_update_counter = 0;
prop_last_key_updated = 0; //So it gets updated right away
update_dist_ptr = oms_dist_load_from_string (updatedist_str);
stat_delay = op_stat_reg("Delay",OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
stat_update_success = op_stat_reg("Update Success",OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
stat_update_success_limited_loss = op_stat_reg("Update Success - Losses by buffer full",OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

if(is_source_mode)
{
schedule_update();
if(stop_time > 0)
{
op_intrpt_schedule_self (op_sim_time() + stop_time, IC_STOP);
}
}
```

A.6 active State: Enter Executives

```
Packet *pPkt;
if(prop_key_update_counter != prop_last_key_updated)
{
active_update_tacker *pTracker;
int i;

//Error check
for(i = 0; i < op_prg_list_size(active_updates_lst); i++)
{
//Might be able to just check the tail instead

pTracker = (active_update_tacker *)op_prg_list_access(active_updates_lst, i);
if(pTracker->update_counter_number == prop_last_key_updated)
{
if(pTracker->has_one_store == 0)
{
op_sim_end("Did not receive store interrupt", "", "", "");
}
}
}
prop_last_key_updated = prop_key_update_counter;
pPkt = op_pk_create_fmt("keyupdate");
op_pk_nfd_set_int32(pPkt, "source_id", source_id);
op_pk_nfd_set_int32(pPkt, "key", prop_key);
op_pk_nfd_set_int32(pPkt, "key_update_number", prop_key_update_counter);
op_pk_nfd_set_dbl(pPkt, "generated_timestamp", op_sim_time());
op_pk_nfd_set_objid(pPkt, "source_prop_objid", self_id);
pTracker = (active_update_tacker *) op_prg_mem_alloc (sizeof (active_update_tacker));
pTracker->update_counter_number = prop_key_update_counter;
pTracker->pkts_alive = 0;
pTracker->has_one_store = 0;
pTracker->gateway_rx = 0;
pTracker->generated_timestamp = op_sim_time();
pTracker->mark_for_delete = 3;
op_prg_list_insert(active_updates_lst, pTracker, OPC_LISTPOS_TAIL);
op_pk_send(pPkt, 0); //Output stream
}
```

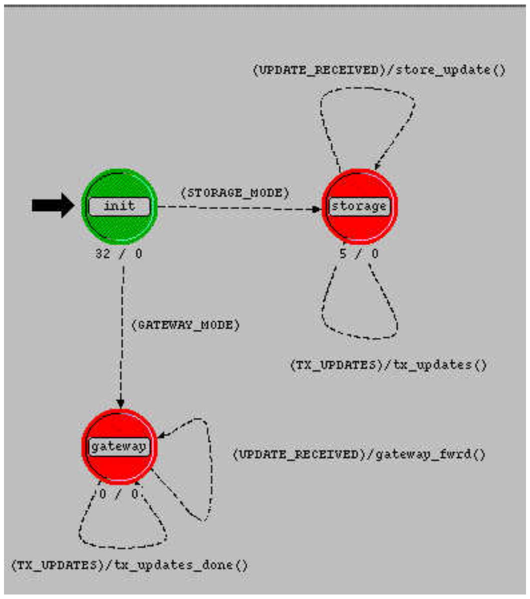
A.7 stop State: Enter Executives

```
char msg_str[255];
if (op_ev_valid (next_update_evh) == OPC_TRUE)
{
sprintf(msg_str, "[%d] Stopping property updates @ %d\n", source_id, op_sim_time());
printf(msg_str);
op_ev_cancel (next_update_evh);
}
```

Appendix B

Code: Storage Process

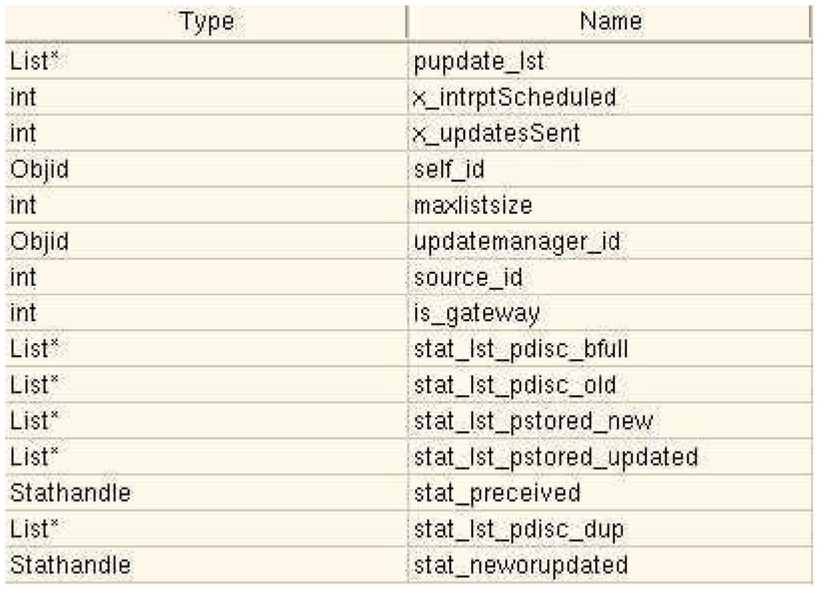
B.1 Overview



(UPDATE_RECEIVED)/store_update()
init
(STORAGE_MODE)
storage
32 / 0
5 / 0
(GATEWAY_MODE)
(TX_UPDATES)/tx_updates()
gateway
(UPDATE_RECEIVED)/gateway_fwrd()
0 / 0
(TX_UPDATES)/tx_updates_done()

Figure B.1: Storage process model

B.2 Local variables



Type	Name
List*	pupdate_lst
int	x_intrptScheduled
int	x_updatesSent
Objid	self_id
int	maxlistsize
Objid	updatemanager_id
int	source_id
int	is_gateway
List*	stat_lst_pdisc_bfull
List*	stat_lst_pdisc_old
List*	stat_lst_pstored_new
List*	stat_lst_pstored_updated
Stathandle	stat_preceived
List*	stat_lst_pdisc_dup
Stathandle	stat_neworupdated

Figure B.2: State variables of storage process

B.3 Header Block

```
#define MAX_SRC_IDS 10
#define GATEWAY_MODE (is_gateway)
#define STORAGE_MODE (!GATEWAY_MODE)
#define STRM_UM_IN 0
#define STRM_UM_OUT 0
#define STRM_GW_OUT 1
//Interrupt Codes (random numbers)
#define IC_DUMP_UPDATES  73
#define IC_DUMP_UPDATES_DONE  74
#define IC_SOURCPROP_ACTION 99
#define TX_UPDATES  (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() == IC_DUMP_UPDATES)
#define UPDATE_RECEIVED (op_intrpt_type() == OPC_INTRPT_STRM)

int written_global_storage_stat = 0;
List *create_stat_lst_loc(const char *);
void store_update(void);
void tx_updates(void);
void gateway_fwrd(void);
```

B.4 Function Block

```
void store_update(void)
{
Packet *pkt;
Packet *lstPkt;
```

```
char message_str [255];
Objid prop1_id;
int key;
int sourceid;
int key_updnm;
int pos_index;
double gen_ts;
int newkey;
int newsourceid;
int newkey_updnm;
double newgen_ts;
int listsize;
int i, j, k;
double temp;
Ici *iciptr;
Objid source_prop_id;
FIN (store_update ());
pkt = op_pk_get (op_intrpt_strm ());

//get info
op_pk_nfd_get(pkt, "key", &newkey);
op_pk_nfd_get(pkt, "source_id", &newsourceid);
op_pk_nfd_get(pkt, "key_update_number", &newkey_updnm);
op_pk_nfd_get(pkt, "generated_timestamp", &newgen_ts);
listsize = op_prg_list_size(pupdate_lst);

if(newsourceid < 0 || newsourceid >= MAX_SRC_IDS)
{
op_sim_end("Bad sourceid", "", "", "");
}
op_stat_write(stat_preceived, 1.0);

//Search & Compare 'key_update_number'; replace if newer
for(i = 0; i < listsize; i++)
{
lstPkt = (Packet *)op_prg_list_access (pupdate_lst, i);
op_pk_nfd_get(lstPkt, "key", &key);
op_pk_nfd_get(lstPkt, "source_id", &sourceid);
op_pk_nfd_get(lstPkt, "key_update_number", &key_updnm);
op_pk_nfd_get(lstPkt, "generated_timestamp", &gen_ts);

//COMPARE for matching source/key
if(newsourceid == sourceid)
{
if(newkey == key)
{
if(newkey_updnm > key_updnm) //if key is newer we update
{
if(newsourceid != source_id)
{
op_stat_write(*((Stathandle *)op_prg_list_access (stat_lst_pstored_updated, sourceid)), 1.0);
op_stat_write(stat_neworupdated, 1.0);
}
op_pk_nfd_get(lstPkt, "source_prop_objid", &source_prop_id);
iciptr = op_ici_create ("prop_action");
op_ici_attr_set (iciptr, "source_id", sourceid);
op_ici_attr_set (iciptr, "key_update_number", newkey_updnm);
op_ici_attr_set (iciptr, "action", 2);
op_ici_install(iciptr);
op_intrpt_schedule_remote (op_sim_time (), IC_SOURCPROP_ACTION, source_prop_id);
iciptr = op_ici_create ("prop_action");
op_ici_attr_set (iciptr, "source_id", sourceid);
op_ici_attr_set (iciptr, "key_update_number", key_updnm);
op_ici_attr_set (iciptr, "action", 3); //3 = discard
op_ici_attr_set (iciptr, "discard_reason", 1); //1 = update
op_ici_install(iciptr);
op_intrpt_schedule_remote (op_sim_time (), IC_SOURCPROP_ACTION, source_prop_id);

op_prg_list_remove (pupdate_lst, i);
op_prg_list_insert(pupdate_lst, pkt, OPC_LISTPOS_TAIL);
op_pk_destroy(lstPkt);
FOUT;
}
else
{
if(newkey_updnm == key_updnm)
{
op_stat_write(*((Stathandle *)op_prg_list_access (stat_lst_pdisc_dup, sourceid)), 1.0);
}
else
{
op_stat_write(*((Stathandle *)op_prg_list_access (stat_lst_pdisc_old, sourceid)), 1.0);
}
//Dont need to trigger the source prop intrupt because this pkt was never stored
op_pk_destroy(pkt);
FOUT;
}
}
}
} //forloop

op_pk_nfd_get(pkt, "source_prop_objid", &source_prop_id);
iciptr = op_ici_create ("prop_action");
op_ici_attr_set (iciptr, "source_id", newsourceid);
op_ici_attr_set (iciptr, "key_update_number", newkey_updnm);
op_ici_attr_set (iciptr, "action", 2); //2 = store
op_ici_install(iciptr);
op_intrpt_schedule_remote (op_sim_time (), IC_SOURCPROP_ACTION, source_prop_id);
op_prg_list_insert(pupdate_lst, pkt, OPC_LISTPOS_TAIL);
listsize = op_prg_list_size(pupdate_lst);

if(newsourceid != source_id)
{
op_stat_write(*((Stathandle *)op_prg_list_access (stat_lst_pstored_new, newsourceid)), 1.0);
op_stat_write(stat_neworupdated, 1.0);
}
//See if we need to get rid of something
if(listsize > maxlistsize)
{
//set first packet for temp
lstPkt = (Packet *)op_prg_list_access (pupdate_lst, 0);
op_pk_nfd_get(lstPkt, "generated_timestamp", &gen_ts);
temp = gen_ts;

//find oldest timestamp
for(j = 0; j < listsize; j++)
{
lstPkt = (Packet *)op_prg_list_access (pupdate_lst, j);
op_pk_nfd_get(lstPkt, "generated_timestamp", &gen_ts);
if(gen_ts < temp)
{
```

```
temp = gen_ts; //replace if older
}
}

//delete packet with oldest timestamp, temp,
for(k = 0; k < listsize; k++)
{
lstPkt = (Packet *)op_prg_list_access (pupdate_lst, k);
op_pk_nfd_get(lstPkt, "generated_timestamp", &gen_ts);
op_pk_nfd_get(lstPkt, "source_id", &sourceid);
op_pk_nfd_get(lstPkt, "key_update_number", &key_updnm);

if(temp == gen_ts)
{
op_stat_write(*((Stathandle *)op_prg_list_access (stat_lst_pdisc_bfull, sourceid)), 1.0);
op_pk_nfd_get(lstPkt, "source_prop_objid", &source_prop_id);
iciptr = op_ici_create ("prop_action");
op_ici_attr_set (iciptr, "source_id", sourceid);
op_ici_attr_set (iciptr, "key_update_number", key_updnm);
op_ici_attr_set (iciptr, "action", 3); //3 = discard
op_ici_attr_set (iciptr, "discard_reason", 2); //2 = Memory Full
op_ici_install(iciptr);
op_intrpt_schedule_remote (op_sim_time (), IC_SOURCPROP_ACTION, source_prop_id);
op_prg_list_remove (pupdate_lst, k);
op_pk_destroy(lstPkt);

break;
}
}
}
FOUT;
}
void tx_updates(void)
{
int i;
int lstSize;
Packet *pkt;
Packet *pPktCopy;
char message_str [255];
FIN (tx_updates ());

lstSize = op_prg_list_size (pupdate_lst);
for (i = 0; i < lstSize; i++)
{
pkt = (Packet *) op_prg_list_access (pupdate_lst, i);
pPktCopy = op_pk_copy(pkt);
op_pk_send(pPktCopy, STRM_UM_OUT);
}
op_intrpt_schedule_remote(op_sim_time(), IC_DUMP_UPDATES_DONE, updatemanager_id);
FOUT;
}
void tx_updates_done(void)
{
FIN (tx_updates_done ());
op_intrpt_schedule_remote(op_sim_time(), IC_DUMP_UPDATES_DONE, updatemanager_id);
FOUT;
}
void gateway_fwrd()
{
FIN (update_gateway ());
op_stat_write(stat_preceived, 1.0);
```

```
op_pk_send(op_pk_get(STRM_UM_IN), STRM_GW_OUT);
FOUT;
}
List *create_stat_lst_loc(const char *statName)
{
List *lst;
int stat_size_asdf;
int i;
char msg1[255];
char msg2[255];

FIN(create_stat_lst_loc());
op_stat_dim_size_get(statName, OPC_STAT_LOCAL, &stat_size_asdf);
if(stat_size_asdf != MAX_SRC_IDS)
{
sprintf(msg1, "stat_size_asdf: %d", stat_size_asdf);
sprintf(msg2, "MAX_SRC_IDS: %d", MAX_SRC_IDS);
op_sim_end("Bad stat dimension", statName, msg1, msg2);
}
lst = op_prg_list_create();
for(i = 0; i < MAX_SRC_IDS; i++)
{
Stathandle *sth_temp;
sth_temp = (Stathandle *) op_prg_mem_alloc (sizeof (Stathandle));
*sth_temp = op_stat_reg (statName, i, OPC_STAT_LOCAL);

op_prg_list_insert(lst, sth_temp, OPC_LISTPOS_TAIL);
}
FRET(lst);
}
```

B.5 init State: Enter Executives

```
int i;
self_id = op_id_self();
printf("REGISTERING STATS\n");
stat_lst_pstored_new = create_stat_lst_loc("Pkts Stored - New");
stat_lst_pstored_updated = create_stat_lst_loc("Pkts Stored - Updated");
stat_lst_pdisc_old = create_stat_lst_loc("Pkts Discarded - Old");
stat_lst_pdisc_bfull = create_stat_lst_loc("Pkts Discarded - Buffer Full");
stat_lst_pdisc_dup = create_stat_lst_loc("Pkts Discarded - Duplicate");
stat_preceived = op_stat_reg("Pkts Received",OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
stat_neworupdated = op_stat_reg("Pkts Stored - New or Updated",OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

pupdate_lst = op_prg_list_create ();  //allocate an empty list
op_ima_sim_attr_get (OPC_IMA_INTEGER, "Capacity", &maxlistsize);
if(written_global_storage_stat == 0)
{
written_global_storage_stat = 1;
op_stat_write_scalar("Storage Capacity", maxlistsize);
}
op_ima_obj_attr_get (self_id, "Source ID", &source_id);
op_ima_obj_attr_get (self_id, "Is Gateway", &is_gateway);
updatemanager_id = op_id_from_name (op_topo_parent(self_id), OPC_OBJTYPE_PROC, "update_manager");
```

B.6 storage State: Enter Executives

```
if(maxlistsize <= 0)
{
//Typically when this node is a gateway
op_sim_end("Invalid max list size", "", "", "");
}
```

Appendix C

Code: Update Manager Process

C.1 Overview

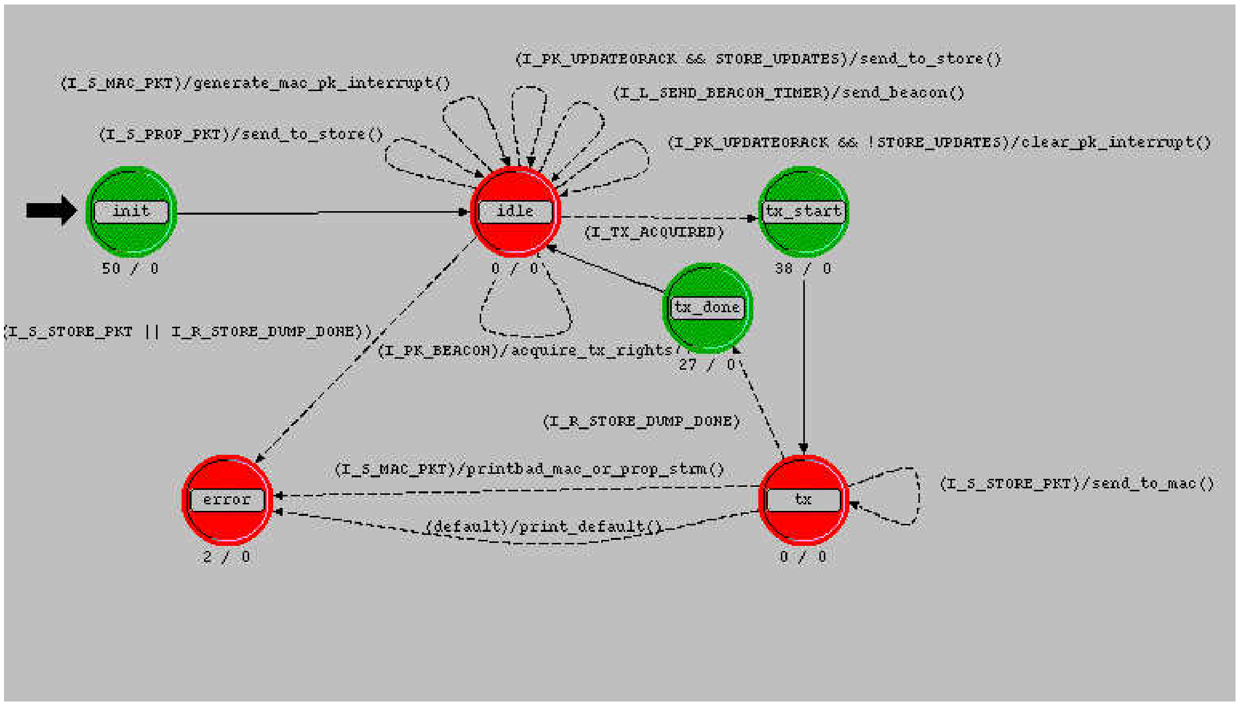
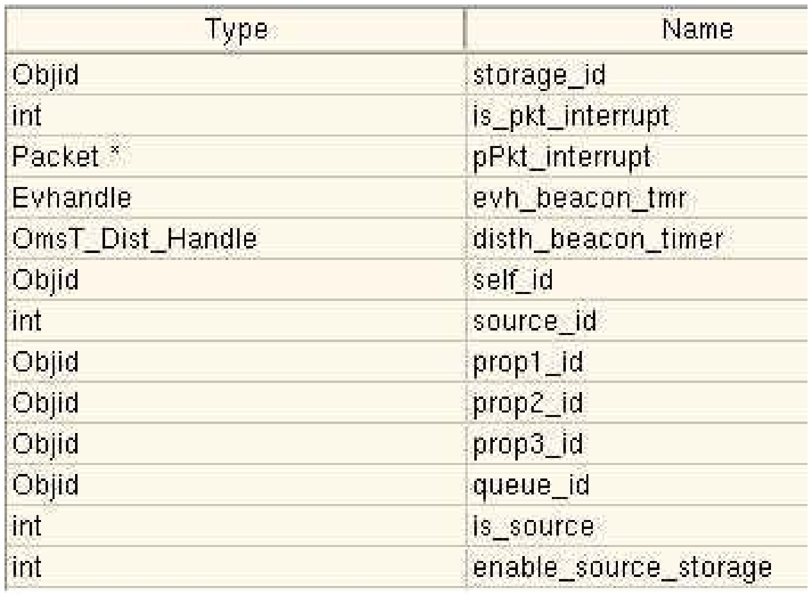

(I_PK_UPDATEORACK && STORE_UPDATES)/send_to_store()
(I_S_MAC_PKT)/generate_mac_pk_interrupt()
(I_L_SEND_BEACON_TIMER)/send_beacon()
(I_S_PROP_PKT)/send_to_store()
(I_PK_UPDATEORACK && !STORE_UPDATES)/clear_pk_interrupt()
init
50 / 0
idle
0 / 0
tx_start
38 / 0
(I_TX_ACQUIRED)
tx_done
27 / 0
(I_S_STORE_PKT || I_R_STORE_DUMP_DONE))
(I_PK_BEACON)/acquire_tx_rights()
(I_R_STORE_DUMP_DONE)
(I_S_MAC_PKT)/printbad_mac_or_prop_strm()
(I_S_STORE_PKT)/send_to_mac()
error
2 / 0
tx
0 / 0
(default)/print_default()

Figure C.1: Update manager process model

C.2 Local variables



Type	Name
Objid	storage_id
int	is_pkt_interrupt
Packet *	pPkt_interrupt
Evhandle	evh_beacon_tmr
OmsT_Dist_Handle	disth_beacon_timer
Objid	self_id
int	source_id
Objid	prop1_id
Objid	prop2_id
Objid	prop3_id
Objid	queue_id
int	is_source
int	enable_source_storage

Figure C.2: State variables of update manager process

C.3 Header Block

```
#include <oms_dist_support.h>
//STREAMS
#define STRM_IN_P1 0
#define STRM_IN_P2 1
#define STRM_IN_P3 2
#define STRM_IN_STORE 3
#define STRM_OUT_STORE 0
#define STRM_IN_MAC 4
#define STRM_OUT_MAC  1

//INTERRUPT CODES
#define IC_REQ_STORE_DUMP 73
#define IC_STORE_DUMP_DONE 74
#define IC_PROP_UPDATES_DISABLE 83
#define IC_PROP_UPDATES_ENABLE 84
#define IC_SEND_BEACON_TIMER 42
#define IC_PK_UPDATEORACK 37
#define IC_PK_BEACON 38
#define IC_Q_DISABLE 54
#define IC_Q_ENABLE 55
#define IC_TX_ACQUIRED 67

//INTERRUPTS

//Stream
#define I_S_PROP_PKT (op_intrpt_type() == OPC_INTRPT_STRM && (op_intrpt_strm() == STRM_IN_P1 || op_intrpt_strm() == STRM_IN_P2 || op_intrpt_strm() == STRM_IN_P3))
```

```
#define I_S_STORE_PKT (op_intrpt_type() == OPC_INTRPT_STRM && op_intrpt_strm() == STRM_IN_STORE)
#define I_S_MAC_PKT (op_intrpt_type() == OPC_INTRPT_STRM && op_intrpt_strm() == STRM_IN_MAC)

//Packet op_intrpt_strm() == STRM_IN_P1
#define I_PK_UPDATEORACK (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() == IC_PK_UPDATEORACK)
#define I_PK_BEACON (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() == IC_PK_BEACON)
#define I_TX_ACQUIRED ((op_intrpt_type() == OPC_INTRPT_SELF || op_intrpt_type() == OPC_INTRPT_REMOTE) && op_intrpt_code() == IC_TX_ACQUIRED)

//Remote
#define I_R_STORE_DUMP_DONE (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() == IC_STORE_DUMP_DONE)

//Local (self)
#define I_L_SEND_BEACON_TIMER  (op_intrpt_type() == OPC_INTRPT_SELF && op_intrpt_code() == IC_SEND_BEACON_TIMER)

//OTHER
#define STORE_UPDATES (!is_source || enable_source_storage)
List *tx_rights_lst = OPC_NIL;

//PROTOTYPES

//Beacon control
void enable_beacon();
void disable_beacon();
void send_beacon();
void send_beacon_timed();
void reset_beacon_timer();
//Property control
void enable_prop_updates();
void disable_prop_updates();
void disable_q();
void enable_q();
//Storage control
void request_storage_dump();
//Packet redirection
void send_to_store();
void send_to_mac();
void generate_mac_pk_interrupt();
void clear_pk_interrupt(void);
void printbad_mac_or_prop_strm(void);
void print_default(void);
void acquire_tx_rights(void);
```

C.4 Function Block

```
void schedule_beacon()
{
double next_becon_time = 0;
FIN(schedule_beacon());
while(next_becon_time <= 0.1)
{
next_becon_time = oms_dist_outcome (disth_beacon_timer);
}
evh_beacon_tmr = op_intrpt_schedule_self (op_sim_time () + next_becon_time, IC_SEND_BEACON_TIMER);
FOUT;
}
void enable_beacon()
```

```
{
FIN(enable_beacon());
schedule_beacon();
FOUT;
}
void disable_beacon()
{
FIN(disable_beacon());
if (op_ev_valid (evh_beacon_tmr) == OPC_TRUE)
{
op_ev_cancel (evh_beacon_tmr);
}
FOUT;
}
void reset_beacon_timer()
{
FIN(reset_beacon_timer());
disable_beacon();
enable_beacon();
FOUT;
}
void send_beacon()
{
char message_str[255];
Packet *pPkt;
FIN(send_beacon());
pPkt = op_pk_create_fmt("beacon");
op_pk_nfd_set_int32(pPkt, "source_id", source_id);
op_pk_send(pPkt, STRM_OUT_MAC);
schedule_beacon();
FOUT;
}
//Property control
void enable_prop_updates()
{
FIN(enable_prop_updates());
op_intrpt_schedule_remote(op_sim_time(), IC_PROP_UPDATES_ENABLE, prop1_id);
op_intrpt_schedule_remote(op_sim_time(), IC_PROP_UPDATES_ENABLE, prop2_id);
op_intrpt_schedule_remote(op_sim_time(), IC_PROP_UPDATES_ENABLE, prop3_id);
FOUT;
}
void disable_prop_updates()
{
FIN(disable_prop_updates());
op_intrpt_schedule_remote(op_sim_time(), IC_PROP_UPDATES_DISABLE, prop1_id);
op_intrpt_schedule_remote(op_sim_time(), IC_PROP_UPDATES_DISABLE, prop2_id);
op_intrpt_schedule_remote(op_sim_time(), IC_PROP_UPDATES_DISABLE, prop3_id);
FOUT;
}
void request_storage_dump()
{
FIN(request_sotrage_dump());
op_intrpt_schedule_remote(op_sim_time(), IC_REQ_STORE_DUMP, storage_id);
FOUT;
}
void send_to_store()
{
Packet *pPktToForward;
FIN(send_to_store());
if(is_pkt_interrupt)
{
```

```
is_pkt_interrupt = 0;
if(pPkt_interrupt == OPC_NIL)
{
op_sim_end("Nill interrupt pkt", "", "", "");
}
pPktToForward = pPkt_interrupt;
pPkt_interrupt = OPC_NIL;
}
else
{
if(pPkt_interrupt != OPC_NIL)
{
op_sim_end("Not nill interrupt pkt", "", "", "");
}
pPktToForward = op_pk_get(op_intrpt_strm());
}
op_pk_send(pPktToForward, STRM_OUT_STORE);
FOUT;
}
void send_to_mac()
{
char message_str[255];
Packet *pPktToForward;
FIN(send_to_mac());
if(is_pkt_interrupt)
{
is_pkt_interrupt = 0;
if(pPkt_interrupt == OPC_NIL)
{
op_sim_end("Nill interrupt pkt", "", "", "");
}
pPktToForward = pPkt_interrupt;
pPkt_interrupt = OPC_NIL;
}
else
{
if(pPkt_interrupt != OPC_NIL)
{
op_sim_end("Not nill interrupt pkt", "", "", "");
}
pPktToForward = op_pk_get(op_intrpt_strm());
}
op_pk_send(pPktToForward, STRM_OUT_MAC);
FOUT;
}
void clear_pk_interrupt()
{
FIN(clear_pk_interrupt());
if(is_pkt_interrupt)
{
is_pkt_interrupt = 0;
if(pPkt_interrupt == OPC_NIL)
{
op_sim_end("Nill interrupt pkt", "", "", "");
}
op_pk_destroy(pPkt_interrupt);
pPkt_interrupt = OPC_NIL;
}
else
{
op_sim_end("clear_pk_interrupt called wrong", "", "", "");
```

```
}
FOUT;
}
void generate_mac_pk_interrupt()
{
char message_str[255];
char format_name[255];
FIN(generate_mac_pk_interrupt());
reset_beacon_timer(); //To prevent a bad state
if(pPkt_interrupt != OPC_NIL)
{
op_sim_end("Not nill interrupt pkt", "", "", "");
}
else if (is_pkt_interrupt)
{
op_sim_end("Pkt interrupt flag set (bad)", "", "", "");
}
else if(op_intrpt_strm() != STRM_IN_MAC)
{
op_sim_end("generate_mac_pk_interrupt called for non mac stream interrupt", "", "", "");
}
pPkt_interrupt = op_pk_get(STRM_IN_MAC);
is_pkt_interrupt = 1;
op_pk_format (pPkt_interrupt, format_name);
if (strcmp (format_name, "beacon") == 0)
{
op_intrpt_schedule_self(op_sim_time(), IC_PK_BEACON);
}
else if (strcmp (format_name, "keyupdate") == 0)
{
op_intrpt_schedule_self(op_sim_time(), IC_PK_UPDATEORACK);
}

FOUT;
}
void disable_q()
{
FIN(disable_q());
op_intrpt_schedule_remote(op_sim_time(), IC_Q_DISABLE, queue_id);
FOUT;
}
void enable_q()
{
FIN(enable_q());
op_intrpt_schedule_remote(op_sim_time(), IC_Q_ENABLE, queue_id);
FOUT;
}
void print_default()
{
FIN(print_default());
printf("DEFAULT\n");
FOUT;
}
void acquire_tx_rights()
{
int i;
FIN(acquire_tx_rights());
if(is_pkt_interrupt)
{
is_pkt_interrupt = 0;
```

```
if(pPkt_interrupt == OPC_NIL)
{
op_sim_end("Nill interrupt pkt", "", "", "");
}
op_pk_destroy(pPkt_interrupt);
pPkt_interrupt = OPC_NIL;
}
else
{
if(pPkt_interrupt != OPC_NIL)
{
op_sim_end("Not nill interrupt pkt", "", "", "");
}
}
for(i = 0; i < op_prg_list_size(tx_rights_lst); i++)
{
if(op_prg_list_access(tx_rights_lst, i) == &self_id)
{
if(i = 0)
{
op_sim_end("NOOOOO.......", "", "", "");
}
FOUT;
}
}
op_prg_list_insert(tx_rights_lst, &self_id, OPC_LISTPOS_TAIL);
if(op_prg_list_size(tx_rights_lst) == 1)
{
op_intrpt_schedule_self(op_sim_time(), IC_TX_ACQUIRED);
}
FOUT;
}
```

C.5 init State: Enter Executives

```
char beacon_dist_str[128];
self_id = op_id_self();
op_ima_obj_attr_get (self_id, "Source ID", &source_id);
op_ima_obj_attr_get (self_id, "Beacon Interval", beacon_dist_str);
disth_beacon_timer = oms_dist_load_from_string (beacon_dist_str);
op_ima_obj_attr_get (self_id, "Is Source", &is_source);
op_ima_sim_attr_get (OPC_IMA_INTEGER, "Enable Source Storage", &enable_source_storage);
storage_id = op_id_from_name (op_topo_parent(self_id), OPC_OBJTYPE_PROC, "storage");
prop1_id = op_id_from_name (op_topo_parent(self_id), OPC_OBJTYPE_PROC, "prop1");
prop2_id = op_id_from_name (op_topo_parent(self_id), OPC_OBJTYPE_PROC, "prop2");
prop3_id = op_id_from_name (op_topo_parent(self_id), OPC_OBJTYPE_PROC, "prop3");
queue_id = op_id_from_name (op_topo_parent(self_id), OPC_OBJTYPE_PROC, "hold_queue");

if(tx_rights_lst == OPC_NIL)
{
printf("Creating TX rights list\n");
tx_rights_lst = op_prg_list_create();
}

//Property stream priorities
```

```
op_intrpt_priority_set (OPC_INTRPT_STRM, STRM_IN_P1, 15);
op_intrpt_priority_set (OPC_INTRPT_STRM, STRM_IN_P2, 15);
op_intrpt_priority_set (OPC_INTRPT_STRM, STRM_IN_P3, 15);
//Lower than property inputs
op_intrpt_priority_set (OPC_INTRPT_STRM, STRM_IN_STORE, 10);
op_intrpt_priority_set (OPC_INTRPT_STRM, STRM_IN_MAC, 8);
//Absolute highest - controlled by stream interrupts
op_intrpt_priority_set (OPC_INTRPT_SELF, IC_PK_BEACON, 20);
op_intrpt_priority_set (OPC_INTRPT_SELF, IC_PK_UPDATEORACK, 20);
op_intrpt_priority_set (OPC_INTRPT_SELF, IC_TX_ACQUIRED, 20);
//Lower than STRM_IN_STORE
op_intrpt_priority_set (OPC_INTRPT_REMOTE, IC_STORE_DUMP_DONE, 9);
//Absolute lowest
op_intrpt_priority_set (OPC_INTRPT_SELF, IC_SEND_BEACON_TIMER, 0);

enable_beacon();
```

C.6 tx start State: Enter Executives

```
if(op_prg_list_access(tx_rights_lst, OPC_LISTPOS_HEAD) != &self_id)
{
op_sim_end("Dont have TX rights", "", "", "");
}
if(is_pkt_interrupt)
{
is_pkt_interrupt = 0;
if(pPkt_interrupt == OPC_NIL)
{
op_sim_end("Nill interrupt pkt", "", "", "");
}
op_pk_destroy(pPkt_interrupt);
pPkt_interrupt = OPC_NIL;
}
else
{
if(pPkt_interrupt != OPC_NIL)
{
op_sim_end("Not nill interrupt pkt", "", "", "");
}
}
if(op_pk_get(STRM_IN_MAC) != OPC_NIL)
{
op_sim_end("STRM_IN_MAC - Packet waiting", "", "", "");
}
disable_q();
disable_prop_updates();
disable_beacon();
request_storage_dump();
```

C.7 tx done State: Enter Executives

```
if(op_pk_get(STRM_IN_STORE) != OPC_NIL)
{
op_sim_end("Store stream not empty", "", "", "");
}
enable_q();
enable_prop_updates();

//Allow the node we just received updates from to transmit
enable_beacon();

if(op_prg_list_remove(tx_rights_lst, OPC_LISTPOS_HEAD) != &self_id)
{
op_sim_end("Dont have TX rights", "Leave", "", "");
}
if(op_prg_list_size(tx_rights_lst)>0)
{
if(op_prg_list_access(tx_rights_lst, OPC_LISTPOS_HEAD) == &self_id)
{
op_sim_end("Stupid OPNET", "This is actually your fault", "", "");
}
op_intrpt_schedule_remote(op_sim_time(), IC_TX_ACQUIRED, *((int *)op_prg_list_access(tx_rights_lst, OPC_LISTPOS_HEAD)));
}
```

C.8 error State: Enter Executives

```
//Unrecoverable error
op_sim_end("Unexpected state", "", "", "");
```

Appendix D

Code: MAC Process

D.1 Overview

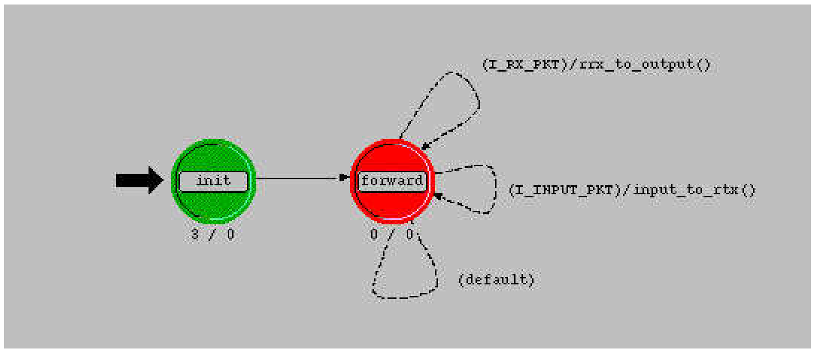
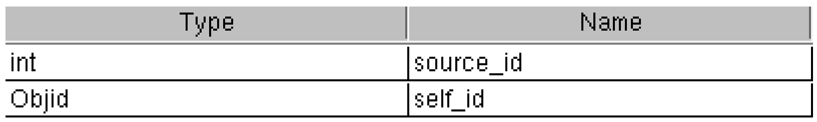


Figure D.1: MAC process model

D.2 Local variables



Type	Name
int	source_id
Objid	self_id

Figure D.2: State variables of MAC process

D.3 Header Block

```
//Streams
#define STRM_INPUT 0
#define STRM_RRX 1
#define STRM_OUTPUT  0
#define STRM_RTX 1
//Interrupts
#define I_RX_PKT (op_intrpt_type() == OPC_INTRPT_STRM && op_intrpt_strm() == STRM_RRX)
#define I_INPUT_PKT (op_intrpt_type() == OPC_INTRPT_STRM && op_intrpt_strm() == STRM_INPUT)
void rrx_to_output(void);
void input_to_rtx(void);
```

D.4 Function Block

```
void rrx_to_output(void)
{
Packet *pPkt;
Objid pktsource;
char message_str[255];
FIN(rrx_to_output());
pPkt = op_pk_get(STRM_RRX);
op_pk_nfd_get(pPkt, "mac_source", &pktsource);
if(pktsource == self_id)
{
op_pk_destroy(pPkt);
}
else
{
op_pk_send(pPkt, STRM_OUTPUT);
}
FOUT;
}
void input_to_rtx(void)
{
Packet *pPkt;
FIN(input_to_rtx());
pPkt = op_pk_get(STRM_INPUT);
op_pk_nfd_set_objid(pPkt, "mac_source", self_id);
op_pk_send(pPkt, STRM_RTX);
FOUT;
}
```

D.5 init State: Enter Executives

```
self_id = op_id_self();
op_ima_obj_attr_get (self_id, "Source ID", &source_id);
```

Appendix E

Code: Hold Queue Process

E.1 Overview

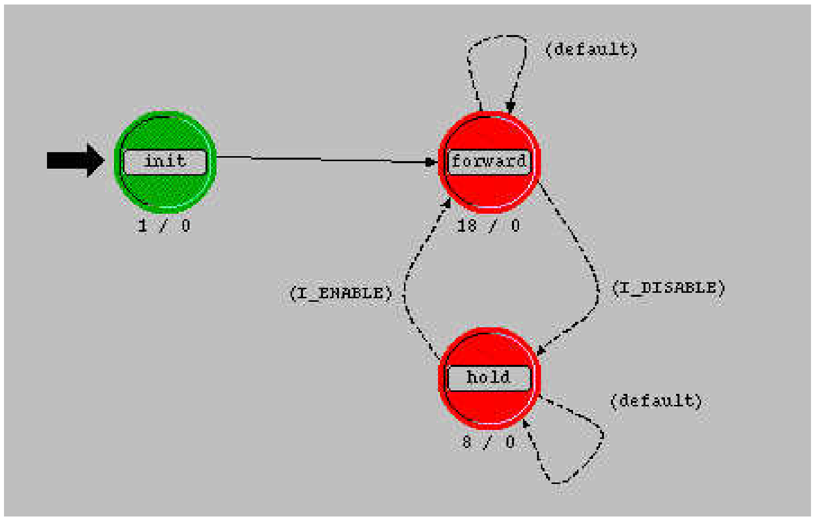
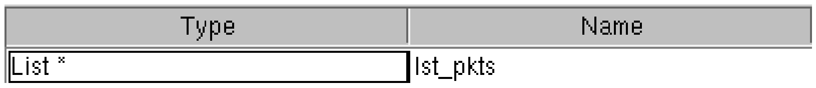


Figure E.1: Hold queue process model

E.2 Local variables



Type	Name
List *	lst_pkts

Figure E.2: State variables of hold queue process

E.3 Header Block

```
#define IC_DISABLE 54
#define IC_ENABLE 55
#define I_DISABLE (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() == IC_DISABLE)
#define I_ENABLE (op_intrpt_type() == OPC_INTRPT_REMOTE && op_intrpt_code() == IC_ENABLE)
```

E.4 Function Block

E.5 init State: Enter Executives

```
lst_pkts = op_prg_list_create ();
```

E.6 forward State: Enter Executives

```
Packet *pPkt;
int len;
len = op_prg_list_size(lst_pkts);

while(len > 0)
{
printf("[HOLD] - Sending buffered pkt\n");
op_pk_send(op_prg_list_remove (lst_pkts, OPC_LISTPOS_HEAD), 0);
printf("\t[HOLD] - Done sending buffered pkt\n");
len = op_prg_list_size(lst_pkts);
}
pPkt = op_pk_get(0);
while(pPkt != OPC_NIL)
{
op_pk_send(pPkt, 0);
pPkt = op_pk_get(0);
}
```

E.7 hold State: Enter Executives

```
Packet *pPkt;
pPkt = op_pk_get(0);

while(pPkt != OPC_NIL)
{
op_prg_list_insert(lst_pkts, pPkt, OPC_LISTPOS_TAIL);
pPkt = op_pk_get(0);
}
```

Appendix F

Code: Gateway Receiver Process

F.1 Overview

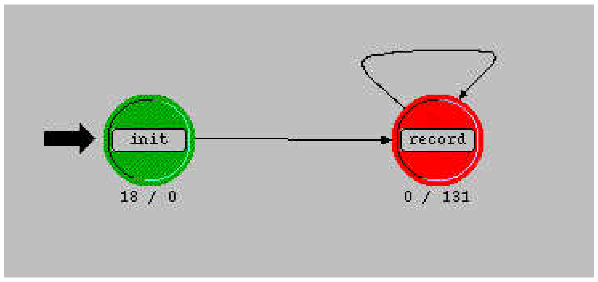


Figure F.1: Gateway receiver process model

F.2 Header Block

```
#define MAX_SRC_IDS 10
#define IC_SOURCPROP_RX  99
int has_inited = 0;
int pkt_received[MAX_SRC_IDS*3];
int hack_pkt_keyupnum[MAX_SRC_IDS*3];
Stathandle stat_neworreplace;
Stathandle stat_counterchange;
List *create_stat_lst(char *);
```

F.3 Function Block

```
List *create_stat_lst(char *statName)
{
List *lst;
int stat_size;
int i;
FIN(create_stat_lst(char *statName));
op_stat_dim_size_get (statName, OPC_STAT_GLOBAL, &stat_size);
if(stat_size != MAX_SRC_IDS)
{
op_sim_end("Bad stat dimension", statName, "", "");
}
lst = op_prg_list_create();
for(i = 0; i < MAX_SRC_IDS; i++)
{
Stathandle *sth_temp;
sth_temp = (Stathandle *) op_prg_mem_alloc (sizeof (Stathandle));
*sth_temp = op_stat_reg (statName, i, OPC_STAT_GLOBAL);
op_prg_list_insert(lst, sth_temp, OPC_LISTPOS_TAIL);
}
FRET(lst);
}
```

F.4 init State: Enter Executives

```
int stat_size_temp;
int i;
if(has_inited == 0)
{
printf("INITIALIZING gateway statistics\n");
has_inited = 1;
for(i = 0; i < MAX_SRC_IDS*3; i++)
{
pkt_received[i] = 0;
hack_pkt_keyupnum[i] = -1;
}
stat_neworreplace = op_stat_reg("Update Pkt - New or Replace" ,OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
stat_counterchange = op_stat_reg("Update Counter Change" ,OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
}
```

F.5 record State: Exit Executives

```
Packet *pPkt;
int key;
int sourceid;
int key_updnm;
double generated_timestamp;
int is_update;
int array_index;
Objid source_prop_id;
char message_str [255];

pPkt = op_pk_get(op_intrpt_strm());
if(pPkt == OPC_NIL)
{
op_sim_end("Nil stream pkt", "", "", "");
}
op_pk_nfd_get(pPkt, "source_id", &sourceid);
op_pk_nfd_get(pPkt, "key", &key);
op_pk_nfd_get(pPkt, "key_update_number", &key_updnm);
op_pk_nfd_get(pPkt, "source_prop_objid", &source_prop_id);
op_pk_nfd_get(pPkt, "generated_timestamp", &generated_timestamp);
//printf("\tEnd getting fields\n");

//Check the fields
if(sourceid < 0 || sourceid >= MAX_SRC_IDS)
{
op_sim_end("Bad source id", "", "", "");
}
else if(key_updnm < 0)
{
op_sim_end("Bad key_updnm", "", "", "");
}
else if(key < 1 || key > 3)
{
op_sim_end("Bad key", "", "", "");
}
array_index = sourceid*3 + (key-1);
is_update = 0;
if(pkt_received[array_index])
{
int oldkey_updnm = hack_pkt_keyupnum[array_index];
if(oldkey_updnm < key_updnm)
{
is_update = 1;
op_stat_write(stat_counterchange, key_updnm - oldkey_updnm);
hack_pkt_keyupnum[array_index] = key_updnm;
}
}
else
{
is_update = 1;
pkt_received[array_index] = 1;
hack_pkt_keyupnum[array_index] = key_updnm;
}
```

```
if(is_update)
{
Ici *iciptr = op_ici_create ("prop_action");
op_ici_attr_set (iciptr, "source_id", sourceid);
op_ici_attr_set (iciptr, "key_update_number", key_updnm);
op_ici_attr_set (iciptr, "action", 1); //Gateway rx code
op_ici_attr_set (iciptr, "generated_timestamp", generated_timestamp);
op_ici_install(iciptr);
op_intrpt_schedule_remote (op_sim_time (), IC_SOURCPROP_RX, source_prop_id);
op_stat_write(stat_neworreplace, 1.0);
}
op_pk_destroy(pPkt);
```