2014-01-01

Neighbor Discovery Message Hold Times for Mobile Ad Hoc Networks

Joshua Lee Mccartney
University of Texas at El Paso, limaner2002@gmail.com

Neighbor Discovery Message Hold Times for Mobile Ad Hoc Networks

Joshua Lee McCartney

Department of Computer Science

APPROVED:

_____
Patricia J. Teller, Ph.D., Chair

_____
Michael McGarry, Ph.D., Co-Chair

_____
Shirley Moore, Ph.D.

_____
Bess Sirmon-Taylor, Ph.D.
Interim Dean of the Graduate School

Neighbor Discovery Message Hold Times for Mobile Ad Hoc Networks

by

Joshua Lee McCartney

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2014

Contents

List of Figures

Chapter 1

Introduction

In order to discuss the problem addressed by this thesis, a little background on networking and routing is needed. Accordingly, Section 1.1 provides a brief introduction to computer networks and a description of the two types of network routing algorithms. In addition, Section 1.2 briefly discusses wireless networks; Section 1.3 introduces a type of wireless network called a *Mobile Ad-Hoc Network* or MANET for short; and Section 1.3.1 describes how the two types of network routing algorithms are extended to work with MANETs and compares them. Finally, Section 1.4 presents the contributions of this thesis and its organization.

1.1 Communication Networks and the Routing Problem

A communications network is composed of devices that can communicate with each other using interconnection *links* with associated *costs*. A graph $G(V,E,C)$ can be used as an abstract model of a communications network, where V, the vertices of the graph, is the set of devices or nodes, and E, the edges of the graph, is the set of interconnection links between devices and C is the set of costs associated with the links (edges). Considering the network shown in Figure 1.1, if device A wishes to communicate with device C with minimum cost, A must do so by sending messages to device B, which then *forwards* them to C. Likewise messages originating from C and destined for A are sent to device B, which then forwards them to A. The sequence of links used to send messages between two devices through one or more links and zero or more intermediate devices is called a *path*. Defining a path with minimal cost between two devices is called *routing*. The number of *hops* is the number of links through which the messages travel, in the afore-mentioned example this number is two. A *one-hop neighbor* is a device that can be reached through one link.

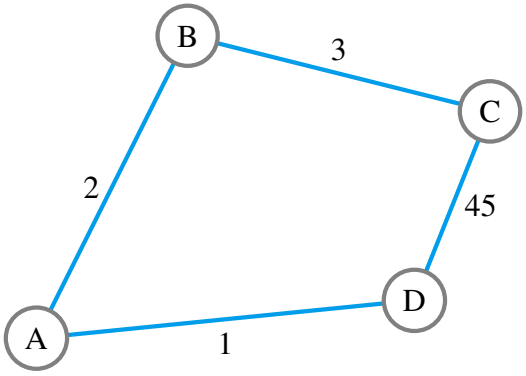


Figure 1.1: A network modeled as a weighted graph with four vertices and four edges.

A network *topology* is an abstract representation of how the devices and links are distributed to compose the network.

1.1.1 Routing Tables

The *routing table* resides on certain devices called routers. The routing table is used to store information about the network the router is on. Different routing algorithms can have different routing table formats, but all routing tables typically contain information on the destinations in the network and an associated cost of sending packets to a particular destination. For the network depicted in Figure 1.1, the routing table could look like that shown in Figure 1.2. As an example, for device A, the Dest. column lists the devices with which A can communicate; the Next Hop column lists the directly connected (one-hop neighbor) devices to which A should send packets in order to communicate with the destination device in the same row of the Dest. column; and the Cost column represents the cost of sending a message or packet to the corresponding destination device. For instance, if A wishes to send data to C, it first looks up C in the Dest. column and identifies that it should send packets to B. When B receives packets from A, it examines the destination in each packet and identifies the destination as C; looks up how to reach C in its routing table and, accordingly, sends the packets to the device in the corresponding Next Hop column, which in this case is C.

Dest.	Next Hop	Cost
B	*B*	2
D	*D*	1
C	*B*	5

(a) Routing table for *A*

Dest.	Next Hop	Cost
A	*A*	2
D	*A*	3
C	*C*	3

(b) Routing table for *B*

Dest.	Next Hop	Cost
A	*B*	5
B	*B*	3
D	*B*	6

(c) Routing table for *C*

Dest.	Next Hop	Cost
A	*A*	1
B	*A*	3
C	*A*	6

(d) Routing table for *D*

Figure 1.2: Example of the contents of routing tables for devices *A*, *B*, *C*, and *D* connected by the network pictured in Figure 1.1.

1.1.2 Message Broadcasting

Before going any further, the idea of *message broadcasting* needs a quick introduction. A message broadcast is a message that is sent to and processed by all nodes in the network that are able to receive it. Considering the network shown in Figure 1.1, if *A* were to broadcast a message, it would be picked up and processed by both nodes *B* and *D*. Then both *B* and *D* would forward the message to *C*. Message broadcasts can be used by routing algorithms to share with devices in the network information that is necessary to allow them to build or update their routing tables.

1.1.3 Out-of-Order Message Delivery

A common problem in networking is known as *out-of-order* message delivery. This can be caused by incorrect routing in which a message was sent out to an incorrect shortest path. This problem has been addressed by such protocols as the *Transmission Control Protocol (TCP)* by use of *sequence numbers*. For example, consider the network shown in Figure 1.1 and suppose two messages with sequence numbers 1 and 2 are to be sent from *A* to *C*. Further suppose that message 1 is incorrectly sent through *D* to *C* while message 2 is correctly sent through *B* to *C* and message 2 arrives at *C* before message 1 does. *C* can inspect the sequence number of the first message and determine that message 1 should have arrived first and, thus, upon the arrival of message 1 can rearrange them so that they are in the order that they were meant to be received. The details of how a protocol such as TCP can correct for out-of-order messages is discussed by

Kurose and Ross in [8].

If the message is too large to be sent into the network, the message itself can also be split up into smaller units called packets. These packets can also be delivered out-of-order and also must carry sequence numbers to correct for this case.

1.1.4 Routing Algorithms

There are two types of algorithms that can be used to solve the network routing problem [8], i.e., the problem of finding a path with minimum cost between two devices in a network, which often is called the shortest path problem: *Distance Vector* (DV) and *Link State* (LS) algorithms.

Distance Vector (DV)

In a DV routing algorithm [8], initially each device only has information about the links to its one-hop neighbors and the associated costs. As time progresses, the distance vector for a node x contains the:

- cost $c(x,y)$: cost of transmitting a message/packet from x to y, for every node y that is directly connected to x, i.e., is a one-hop neighbor of x;
- cost $c(x,d)$: cost of transmitting a message/packet from x to d, for every node d in the network that is a possible destination from x; and
- D_v: the distance vector of each node v that is a one-hop neighbor of x.

When calculating the shortest path to a destination device to which it wants to communicate, a device exchanges *distance vectors* with all of its one-hop neighbors. When a device receives a distance vector from another device, it updates the corresponding locally-stored distance vector using Equation (1), which is described below, and then notifies its one-hop neighbors if any information in the distance vector has changed. This process continues until no distance vectors require updating. In this way all the nodes in the network receive updated network topology

information. However, DV algorithms can suffer from the so-called *count-to-infinity* problem, which is described below, since *routing loops* can form.

As mention above, when a device x receives a distance vector from any of its neighbors v, it updates its corresponding locally-stored distance vector using Equation (1), the Bellman-Ford equation as follows:

$$D_x(y) = \min_v[c(x,v) + D_v(y)] \quad \text{for each node } y \text{ in } N$$

The Count-to-Infinity Problem

Consider the network shown in Figure 1.1. The distance vector for each of the devices can be found by looking at the routing tables shown in Figure 1.2. So for device A, $D_A(y)$ is shown in 1.2a where y is one of the devices listed in the destination column. Now suppose that the link between B and C goes down. B will eventually detect this link failure and notify A that the link has gone down and will update its distance vector as follows: $D_B(C) = \min[c(B,C) + D_B(C), c(B,A) + D_A(C)] = \min[\infty + 0, 2 + 5] = 7$. After some time, B will then send out its new distance vector to A. A will then update its distance vector as $D_A(C) = \min[c(A,B) + D_B(C), c(A,D) + D_D(C)] = \min[2+7, 1+6] = 7$. D will update its distance vector as $D_D(C) = \min[c(D,A) + D_A(C), c(D,C) + D_C(C)] = \min[1+7, 45+0] = 8$ and then notifies A. A will then update once again and notify D, and this will continue for infinitely many iterations. Basically the problem is that D still thinks that the best way to get to C is to go through A and B, and B knows that the best way to get to C is through A and D. This ping-ponging is known as a routing loop and can be solved using a relaxation property of Bellman-Ford [8], which is beyond the scope of this thesis.

Link State (LS)

In a Link State (LS) [8] routing algorithm, each device has a full information about all other devices in the network, including each device's one-hop neighbors and the costs associated with all links in the entire network. This information is used by each device to determine the shortest

path to every other device. In order to facilitate this, each device periodically broadcasts to every other device in the network the identities of its one-hop neighbors and the costs of the associated links. If a change in a link is detected by a one-hop neighbor, it broadcasts this information to all of its one-hop neighbors so that they can update their routing tables accordingly . Exactly how a change is detected is dependent on the particular protocol being used. As the number of devices participating in the network increases, so does the overhead of maintaining the global network information. The broadcasting of messages to set up and maintain route information is referred to as *control* message traffic.

1.2 Wireless Networks

Wireless networks have been a topic of research ever since their emergence in the 1970s and have become increasingly popular since the 1990s. This is especially true in the past two decades during which the use of mobile devices such as laptops, tablets, and cell phones has flourished. Wireless networks make it possible for these untethered devices to communicate. There are two types of wireless networks: infrastructured and infrastructureless. The wireless devices in an *infrastructured network* communicate through a central device, commonly a router or access point. In order for a device to maintain connectivity with the other devices in this type of network, it must remain within range of the central communication device. Accordingly, infrastructured networks have central points of failure; if the central communication device goes down, the entire wireless network effectively goes down. In contrast, wireless devices in an *infrastructureless network* communicate with each other directly without having to go through a central device.

1.3 MANETs

A *Mobile Ad Hoc Network* (MANET) is an infrastructureless network that is comprised of wireless-enabled mobile devices that dynamically establish communication links between each other in real time. These mobile devices, which typically have short communication ranges, are

free to move and, thus, may move in and out of range of the other devices in the network. Thus, there needs to be a way to establish and re-establish device connectivity. In addition, each device must be able to transmit data to other devices in the network reliably and in a timely fashion.

The military utilizes wireless mobile ad hoc networks (MANETs) [2] [1] to establish communication networks on the battlefield. These wireless MANETs require self-configuring properties as they are deployed rapidly without the use of established infrastructure. The foundational self-configuring property of these battlefield wireless MANETs is the ability for the network to self-discover its topology. This topology discovery process begins with the process of neighbor discovery.

A routing protocol is used to manage communication within a MANET. Such a protocol obtains information (such as device identities and link costs to their neighbors) about the network and uses it to determine the shortest path between devices in order to affect timely data transmissions. However, since the devices of a MANET are mobile, the links and shortest paths between devices may change over time. Accordingly, the routing protocol must dynamically keep track of the connectivity of devices and quickly refine the shortest paths between devices.

1.3.1 MANET Routing Protocols

Several MANET routing protocols have been developed to handle reliable connectivity reactively and proactively. A *reactive protocol*, e.g., Ad hoc On-Demand Distance Vector (AODV) [10], is a DV algorithm that only changes topology information on demand, i.e., when a device needs to transmit data. A *proactive protocol*, e.g., Optimized Link-State Routing (OLSR) [3], is an LS algorithm that configures all routes, including the shortest routes, between all devices before any data is transmitted on the network.

AODV

An example of how an AODV protocol calculates a route is shown in Figure 1.3. If mobile device *A* wishes to communicate with mobile device *G*, *route request* messages (rreq) are broadcast by

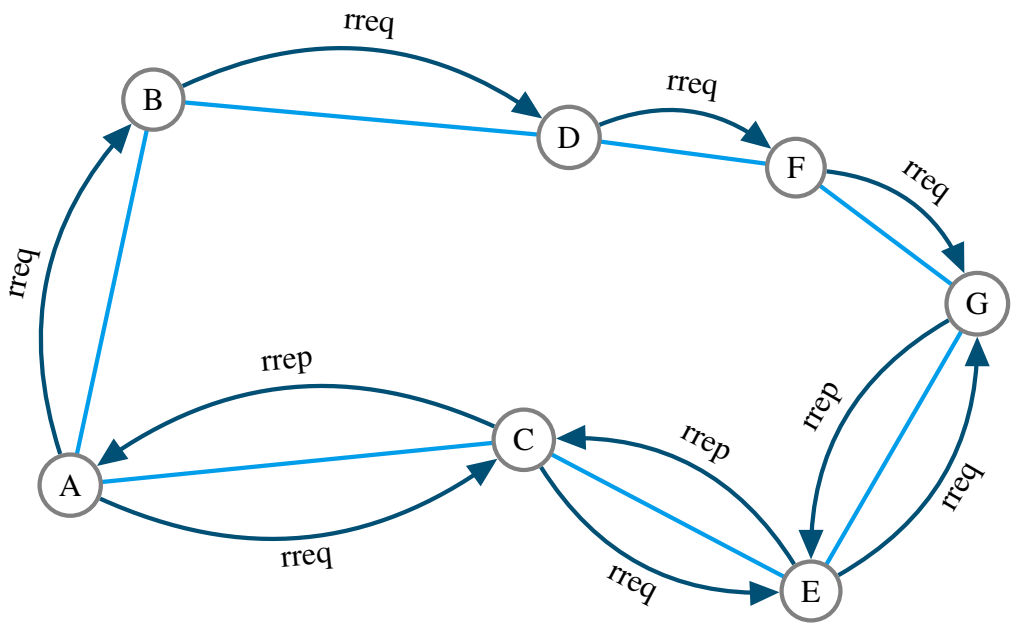


Figure 1.3: AODV route setup between *A* and *G* before any data transfer can take place.

A to all devices in the network. The rreq keeps a record of every node that it reaches as it is forwarded throughout the network. Once *G* receives the rreq, it looks at the path recorded in the rreq and sends a *route reply* message (rrep) back along the path traversed by the rreq from *A*. In the case when *G* first receives the rrep from device *E* and then *G* receives the rrep from device *F*, it notices that the rrep from *F* has the same sequence number as the one it previously received from *E* and, thus, *G* discards the rrep from *F*. If at some later time, *A* and *G* need to communicate, and the network topology has not changed in a way that affects any of the links along the path *A*-*C*-*E*-*G*, the process of sending rreq messages is not necessary. However, if something were to happen and one of the links, say *C*-*E*, on the path from *A* to *G* were to go down, *A* would not receive any acknowledgment that *G* was able to receive any messages. This means that a new route needs to be found and thus another rreq message would be broadcast and eventually the route *A*-*B*-*D*-*F*-*G* would be used.

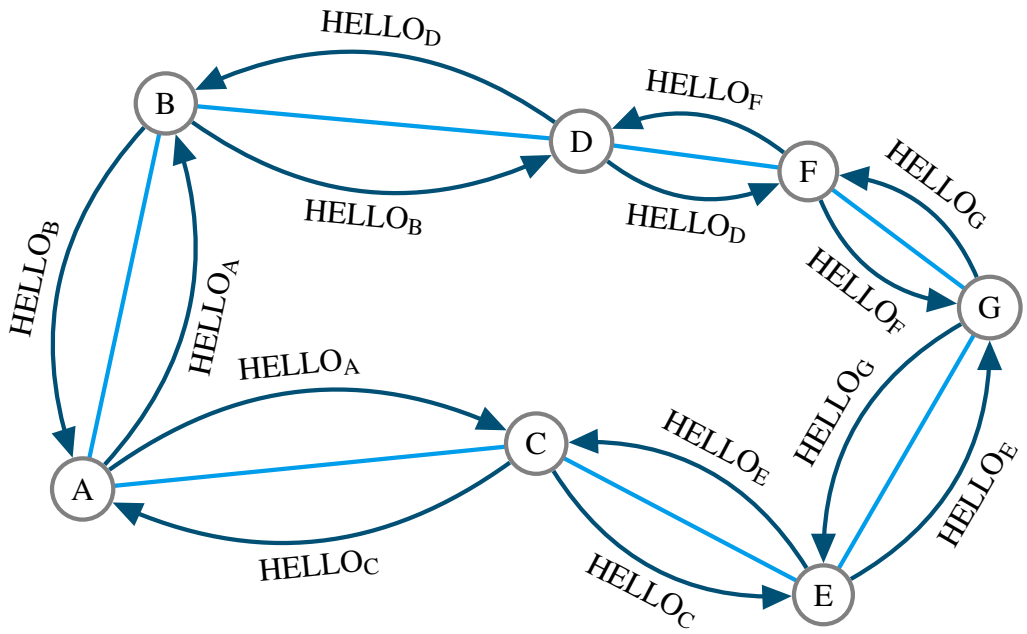


Figure 1.4: OLSR route setup between *A* and *G*.

OLSR

This section provides a brief introduction to the OLSR protocol. A more detailed discussion of how the various control messages are used to perform neighbor discovery and route calculation is presented in Chapter 2.

As discussed in Section 1.1, OLSR is an LS algorithm. It uses two types of messages: (1) *HELLO* messages to detect the presence of links between devices and (2) *Topology Control* (TC) messages to inform other devices about all the links in the network. Figure 1.4 shows an example network topology that employs the OLSR protocol. The network links, which are represented by the light lines, are detected by the HELLO messages, which are represented by the curved arrows. The label "HELLO$_i$", identifies a HELLO message that originated from device i. Note that although HELLO messages never travel any farther than one hop, i.e., one link, these messages allow devices to know how to reach other devices that are up to a maximum of two hops apart. As an example, referring to Figure 1.4, if device *A* wants to communicate with device *G*, more

information about the network links is required than that provided solely by HELLO messages, which only identify what devices are one hop away from A. This is where the TC messages come in. Figure 1.5 shows a TC message generated by A that is broadcast to all devices in the network to allow every device to know that A has a link to devices B and C. Similarly, B, C, D, E, and F follow the same process. In this way, each device in the network has an up-to-date complete information about the network topology and can identify the shortest path to any other device in the network.

Devices running the OLSR protocol elect a subset of devices as *Multi-Point Relays* (MPRs). Only devices that have been elected as MPRs forward both TC messages and data destined for other devices. Allowing only MPRs to forward traffic helps reduce the load on the network and is the heart of the OLSR protocol. [3]

Comparison of AODV and OLSR Algorithms

Performing route calculations only when necessary, as in AODV protocols, helps reduce unnecessary overhead. The disadvantage of this, however, is an increase in communication latency [7] – only after a route is determined can the data be transmitted. It should also be clear that if the devices are highly mobile, and consequently many links are changing, the on-demand method of route calculation can be problematic. Performing the route calculations before they are needed, as in OLSR protocols, helps reduce the latency associated with DV algorithms. However, as discussed in Section 1.1, LS algorithms, e.g., OLSR, can accrue a significant amount of overhead associated with control messages, especially for networks consisting of a large number of devices. The U.S. Army, which funds this research through the Army High Performance Computing Research Center, is interested in MANETs with low latency; thus, this thesis and future work employs OLSR as the protocol of choice in our research.

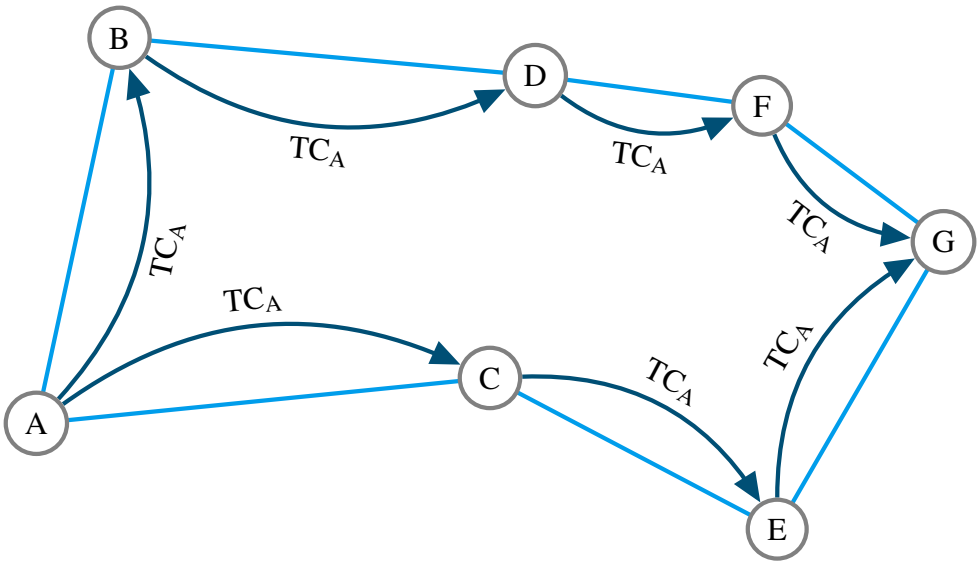


Figure 1.5: TC messages being broadcast throughout the network

1.4 Thesis Contributions and Organization

As mentioned at the end of the previous section, OLSR provides lower latency than the AODV algorithm, but at the cost of a higher amount of control traffic. Higher amounts of control traffic can increase the length of time it takes to receive a message, due to the message latency and the loss of data, i.e., packet loss. Both these phenomena can have an adverse effect on the performance of mobile devices in the network. Consequently, it is desirable to reduce the amount of control traffic, while minimally impacting network performance and, thus, device performance. One way to reduce the amount of control traffic is by changing the intervals at which control messages, such as HELLO and TC messages, are broadcast over the network. It is, therefore, necessary to understand the impact of control message parameter configuration on network performance.

The goal of this thesis is to facilitate the creation of an adaptive OLSR-based routing protocol such that parameters are automatically configured based on knowledge of device mobility. An understanding of how the OLSR parameter settings affect network performance is needed before such a protocol can be constructed. Accordingly, this thesis research resulted in:

- the design and development of a physical experimental testbed,
- the exploration of how one of these control message parameters, the HELLO_VALIDITY time, affects the network in terms of packet loss, and
- the quantification of the relationship between the HELLO message transmission interval and the HELLO_VALIDITY time.

As described in Chapter 5, through experiments conducted using the physical testbed that we built, we found that setting the HELLO_VALIDITY time to a value 25% to 50% larger than the HELLO message transmission interval appears to be the best choice.

The remainder of the thesis is organized as follows. Chapter 2 presents the OLSR protocol in more depth, while Chapter 3 provides an overview of other work that has been done with regard to understanding the impact of OLSR parameter settings on network performance and compares this work with ours. The experimental methodology followed in our research is described in Chapter 4, which provides a detailed description of the experimental platform that was built for this and follow-on research, and an explanation of the experiments that were conducted and the measurements that were used to quantify the impact of the HELLO_VALIDITY time on network performance. Results of the experiments are presented and discussed in Chapter 5. Chapter 6 ends the thesis with conclusions and future work.

Chapter 2

Overview of Optimized Link-State Routing (OLSR)

A brief introduction to Optimized Link-State Routing, OLSR, an optimization of Link State routing, was given in Section 1.3.1. There we introduced how OLSR uses HELLO and Topology Control (TC) messages to detect the presence of links between devices in a network and informs the other devices in the network about all the links in the network, and briefly mentioned the role of Multi-Point Relays (MPRs). This chapter goes into more depth about the these two topics, which are the portions of the OLSR protocol that are relevant to the research conducted as part of this thesis. For complete details of the full OLSR protocol, refer to rfc3626[3] hosted by the Internet Engineering Task Force (IETF).

This chapter has four sections. First, Section 2.1 describes how neighboring devices are discovered, a process called *link sensing* or *neighbor discovery*. As the devices in a network discover their neighbors, they disperse their neighborhood information (device identities and link costs) to every other node in the network via TC messages, which are discussed in more detail in section 2.2. And, as described in Section 2.3, when a device discovers information about the topology of the network, it adds that information to its routing table, which eventually includes all routes, including the shortest, between all devices in the network. Finally, Section 2.4 discusses MPRs or MPR nodes, in particular, how a device becomes an MPR and how MPRs reduce network traffic.

However, before going any further, we present the basic format of an OLSR data packet, which is pictured in Figure 2.1. As shown in the figure, a data packet has nine fields plus the payload, i.e., the data being transmitted. The fields of an OLSR data packet are described below:

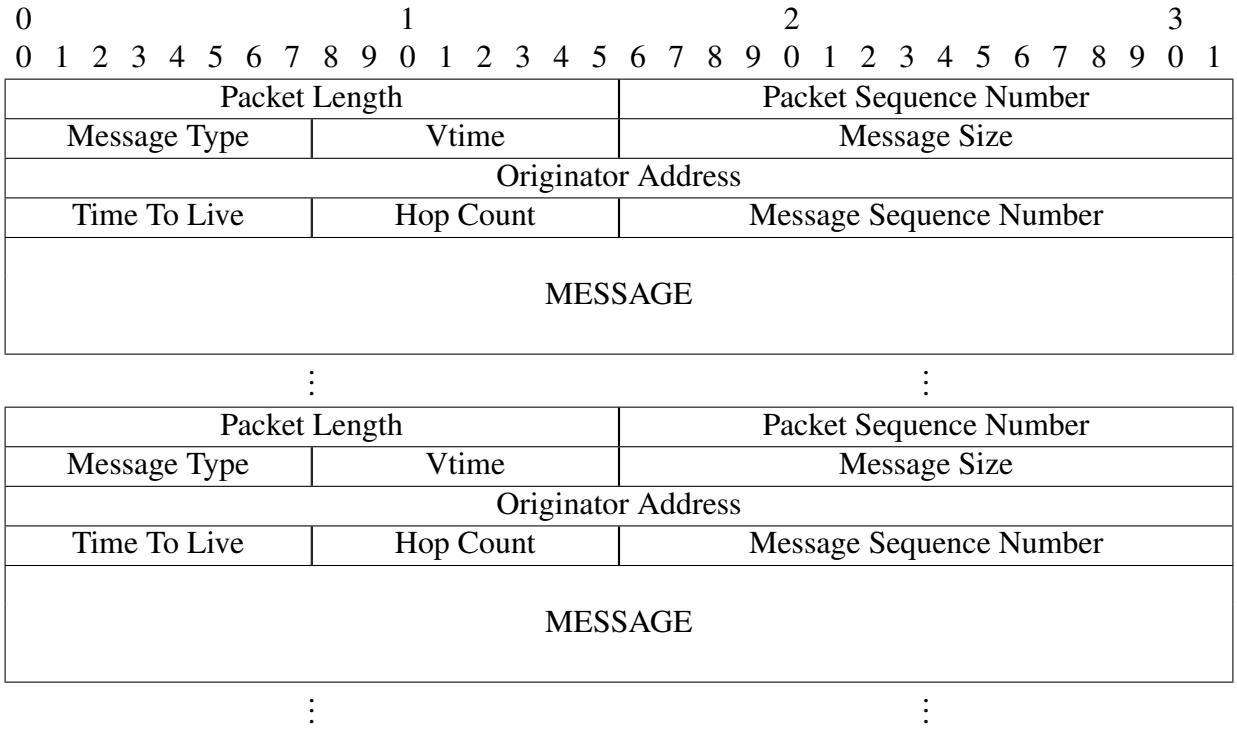


Figure 2.1: Basic OLSR Message/Packet Format

1. **Packet Length:** length of the packet in bytes.

2. **Packet Sequence Number:** sequence number of the packet. This is used to reorder packets that have been received out-of-order as described in Section 1.1.3.

3. **Message Type:** type of OLSR message

4. **Vtime:** validity time of this message

5. **Message Size:** size of the message in bytes measured from the beginning of the "Message Type" field to the beginning of the "Message Type" field of the next packet, or until the end of this packet.

6. **Originator Address:** IP address of the device that created this message, i.e., the source IP address.

7. **Time To Live (TTL):** an integer that represents the maximum number of hops this message will travel. Each time the message is forwarded by a device, this field is decremented by 1. When the value reaches 0, an ICMP error is generated and the packet is dropped.

8. **Hop Count:** distance (in hops) between the source device and the device where the message currently is. Each time the message is forwarded by a device, this field is incremented by one.

9. **Message Sequence Number:** sequence number of the message. This is used to reorder messages that have been received out-of-order as described in Section 1.1.3.

2.1 Link Sensing and Neighbor Discovery

As mentioned in Section 1.3, the foundational self-configuring property of MANETs is the ability for the network to self-discover its topology. In order to do this, the process begins with neighbor discovery. In OLSR this is achieved through the use of HELLO messages. When performing neighbor discovery, an implementation of OLSR may make use of underlying link-layer acknowledgments, however this is optional and not necessary for OLSR to work properly. The HELLO messages are OLSR messages that are periodically broadcast with the Message Type set to HELLO and the Time to Live set to one. Setting the Time to Live value to one ensures that only devices that are directly in range, i.e., potential one-hop neighbors, will process such a message and will not forward it to other devices in the network. The period of time between HELLO message broadcasts is the *HELLO_INTERVAL* (δ).

When a node x creates a HELLO message, which is pictured in Figure 2.2, it populates the fields of the message as described below:

1. **Reserved:** all zeros to comply with rfc3626 [3].

2. **Htime:** amount of time after receipt of this message that this message is considered to be valid (τ). The units and how this field is interpreted are left up to the implementation.

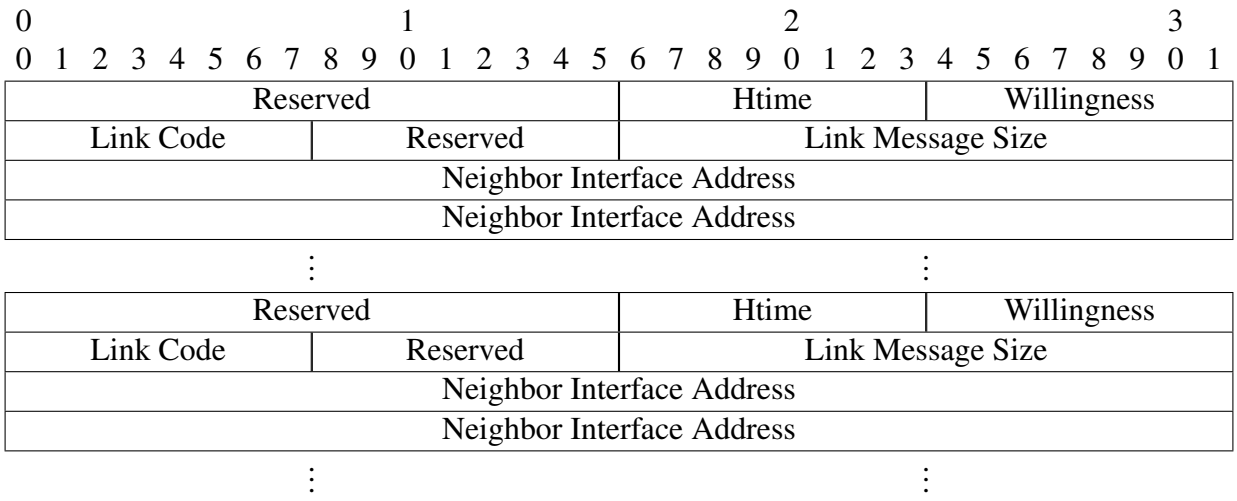


Figure 2.2: OLSR HELLO message format

3. **Willingness:** willingness of a device to be elected as an MPR. According to rfc3626, the possible values for this field are any integer from 0 to 7 with 0 being list willing and 7 most willing.

4. **Link Code:** status of the neighbor, i.e. symmetric (two-way communication) or asymmetric (one-way communication).

5. **Reserved:** all zeros to comply with rfc3626 [3].

6. **Link Message Size:** size (in bytes) of this message as measured from the beginning of the "Link Code" field to the "Link Code" field of the next message, or until the end of this message.

7. **Neighbor Interface Address:** list of all IP addresses of the neighbors of the source device.

To exemplify how HELLO messages are used, Figure 2.3 shows a timing diagram of two devices establishing links to each other using OLSR HELLO messages, which is described below. While it may appear that devices *a* and *b* are sending messages to each other, they are, in fact, broadcasting these HELLO messages. The diagram only shows the process in the context of these two devices.

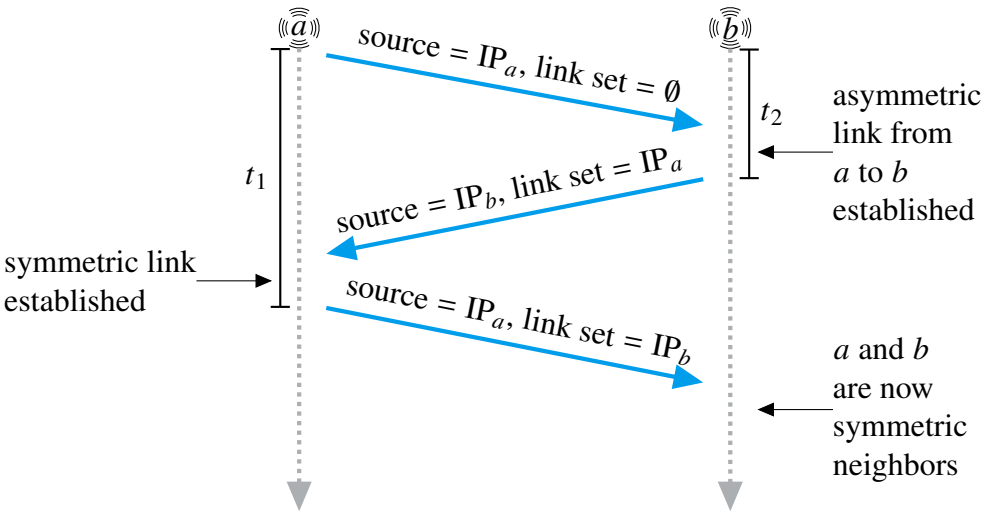


Figure 2.3: Timing diagram of two devices establishing communication links between themselves using the OLSR protocol. The broadcasting of HELLO messages is only triggered by reaching times $t_0 + t_1 \leq \delta$ and $t_0 + t_2 \leq \delta$.

Initially we assume that the two devices are not aware of each other or of any other devices in the network. As shown, first, at time t_0, device a broadcasts a HELLO message with its IP address as the source. When device b receives the message, it adds the IP address of a to its *one-hop neighbor set* and, in this way, an asymmetric (one-way) link from a to b is established. Next, at time $t_0 + t_2$, device b broadcasts a HELLO message with its IP address as the source and the IP addresses of its one-hop neighbors, in this case only a. When device a receives this message, since the HELLO message from b contains the IP address of a as one of the neighbors of b, a determines that b is within range and adds b to its one-hop neighbor set or *link set* and, thus, a symmetric link is established between a and b and two-way communication is possible. Note that at this point in time, only a is aware that two-way communication is possible. However, a is scheduled to broadcast another HELLO message at time $t_0 + t_1$ and that message will include b in the link set of a. Accordingly, when device b receives that message, it notes that it is in the one-hop neighbor set of a and, thus, notes that the link with a is symmetric. In this way, both a and b successfully discover each other as symmetric neighbors.

It is important to note that the HELLO messages in the previous example are not sent as a result of receiving a HELLO message. They are broadcast at a particular frequency defined by the HELLO message transmission interval, δ. In this example, they are broadcast as a result of reaching times t_0, $t_0 + t_1$ and $(t_0 + t_2) \leq \delta$. It could easily have happened that time $t_0 + t_2$ occurred before b received the HELLO message from device a. In this case, b would have broadcast a HELLO message that contained b as its source and a $\emptyset$ as its link set. Then, upon receipt of this message a would have added b to its link set as an asymmetric one-hop neighbor. In the meantime, upon receipt of the HELLO message from a, b would have added a to its link set as an asymmetric one-hop neighbor. At time $t_0 + t_1$, a would have broadcast another HELLO message that would have caused b to realize that it was in the link set of a and to indicate that a was a symmetric one-hop neighbor. When b broadcasts its next HELLO message then a will realize that b is a symmetric one-hop neighbor as well.

This same process can be used to detect neighbors up to two hops away. For example, consider the same scenario as illustrated in Figure 2.3 and described earlier, but this time assume that device b already has another neighbor c that is not within range of a. The process is the same, except that when b broadcasts its HELLO message, its link set will include both a and c. And, when a receives and processes this message, it will realize that c is not in its link set. When no record of c is found in either set, it will record c in its *two-hop neighbor set*.

2.2 Topology Discovery

The previous section briefly discussed link sensing and neighbor discovery. This section describes how that information is disseminated to all devices in the network and the following section, Section 2.3, describes how it is used to construct routing tables and, thus, routes.

In addition to HELLO messages, each node that has been selected as an MPR node (selection of these nodes is discussed in Section 2.4) broadcasts *Topology Control (TC)* messages to every device in the network. As shown in Figure 2.4, a TC message contains the following fields: (1) source: IP address of the originator of the message, (2) neighbors: the set of IP addresses of the

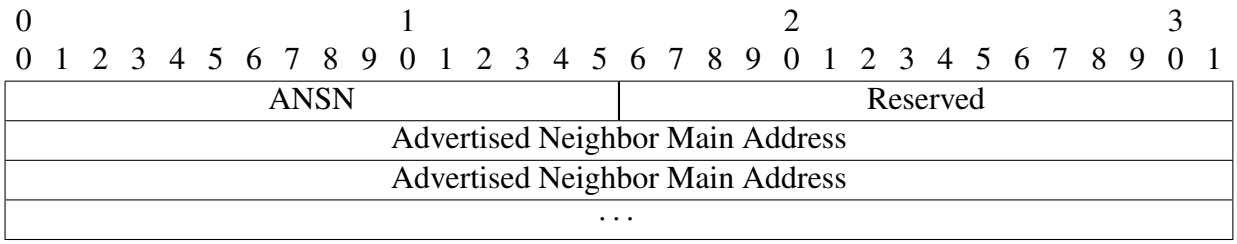


Figure 2.4: The OLSR TC Message Format

one-hop neighbor devices of the originator, and (3) ansn: advertised neighbor sequence number, which is increased each time the originator detects a change in its neighbor set.

A receiving device examines TC messages and uses the transmitted information to create its *Topology Information Base*, TIB. This information base contains a set of at least one *Topology Tuple* for each destination in the network. As shown in Figure 2.5, each tuple contains four entries: (1) dest: address of the destination device, (2) last: address of a device that is one-hop away from the device with dest address (usually the MPR node of the destination device), (3) seq: message sequence number, and (4) time: time at which this tuple expires.

To demonstrate what happens when a TC message is received, assume the network topology shown in Figure 2.7 and consider what happens when the message is received by device g, which contains the Topology Information Base (TIB) shown in Figure 2.5a. The solid lines of Figure 2.7 represent the links of which g is currently aware, while the dashed lines represent the links of which g is not currently aware but of which other devices in the network may be aware, in particular devices a, b, and c. First, note that since devices b and e are the only MPR nodes in this network, TC messages will originate only from them. Assume that b transmits a TC message that contains b as its source, devices a, c, and e as its one-hop neighbors, and 32 as its sequence number. Device e receives this message and since it is a TC message and, thus, is meant for all devices in the network, it rebroadcasts it to devices a, d, f, g and h. When g receives the message, it uses the new information regarding a to create a new tuple for a with (dest: a, last: b, seq: 32, and time: t_v) and the information regarding the new device c to create a new tuple for c, i.e., (c, b, 32, t_v). Accordingly, device g has the TIB shown in Figure 2.5b. Note that the tuple for a in the initial TIB of device g, shown in 2.5a, was created using information received from

dest	last	seq	time
a	e	9	t
b	e	9	t
d	e	9	t
e	e	9	t
f	e	9	t
h	e	9	t

(a) The initial information base

dest	last	seq	time
a	e	9	t
b	e	9	t
d	e	9	t
e	e	9	t
f	e	9	t
h	e	9	t
a	b	32	t_v
c	b	32	t_v

(b) the information base after receipt of TC message from device b

Figure 2.5: The topology information base for device g in the topology shown in Figure 2.7

another device, device e.

2.3 Routing Table Calculation

As the information from the TC messages is received by the devices in the network, each device executes a shortest path algorithm using the links information contained in its Topology Information Base (TIB) to evolve its routing table. Detection of a link change in the network causes the routing table to be regenerated. A link change happens either when the HELLO_VALIDITY time expires or, as discussed in Section 2.1, a new device is detected by a HELLO_MESSAGE.

The resulting routing table looks similar to the one shown in Section 1.1.1. The only difference is the inclusion of a new column that identifies the interface to use to send packets. This information is needed because a device may have multiple interfaces connected to different networks, for example an Ethernet card and a wireless card, and when it needs to send packets to another device, the device must use the correct interface.

As an example, consider the network topology shown in Figure 2.7 and the routing tables for devices b, e and g shown in Figure 2.6. As described in Section 2.2, assume that g just received a TC message that identifies a new device c and added the information regarding c, with device e denoted as "next", to its TIB. Further assume that g currently has no entry for c in its routing

dest	next	dist	iface
a	*a*	1	0
c	*c*	1	2
d	*e*	2	1
e	*e*	1	1
f	*e*	2	1
g	*e*	2	1
h	*e*	2	1

(a) Routing table for device *b*

dest	next	dist	iface
a	*a*	1	2
b	*b*	1	2
c	*b*	2	2
d	*d*	1	1
g	*g*	1	0
h	*h*	1	0
f	*f*	1	0

(b) Routing table for device *e*

dest	next	dist	iface
a	*e*	2	0
b	*e*	2	0
d	*e*	2	0
e	*e*	1	0
f	*e*	2	0
h	*h*	1	1
c	*e*	3	0

(c) Routing table for device *g*

Figure 2.6: OLSR routing tables for devices *b*, *e*, and *g* in the network shown in Figure 2.7. Device *g* adds destination *c* to its routing table after receiving a TC message with the necessary information.

table. The new entry for *c* in the TIB of *g* indicates that *c* can be reached via *b* and the routing table of *g* provides the route to *b*, i.e., via *e*, which is a one-hop neighbor of *g*. As a result, as shown in the shaded entry in Figure 2.6c, *g* adds a new entry for *c* to its routing table, which indicates how it can communicate with the new device *c*.

2.4 MPR Nodes

As we have seen in the previous sections of this chapter, HELLO and TC messages are broadcast. As described in Section 1.1.2, a broadcast message is forwarded by every device that receives it except when the TTL field is 0. In the latter case, which is the case for HELLO messages, the message only reaches the source node's one-hop neighbors, which do not forward it and an error is sent back to the source address. TC messages must be forwarded to ensure that they reach every device in the network so that they can generate or regenerate their routing tables.

MPR nodes are elected based on the one-hop and two-hop neighbor sets of the nodes in the network. Each node selects a set of its *willing* one-hop neighbors to serve as MPR nodes; a node can advertise that it is unwilling to serve as an MPR node and, in this case, it is excluded from the selection. The main objectives of MPR selection are: (1) to enable forwarding of data packets

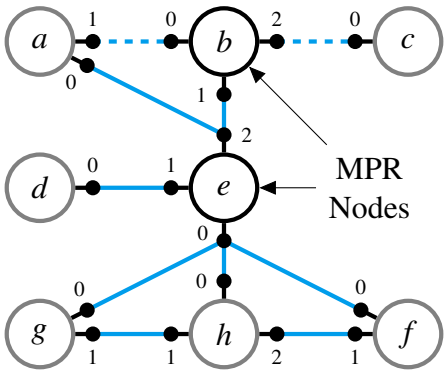


Figure 2.7: An example network topology in which devices b and e are MPR nodes. The numbers near the devices represent the interface number of the device.

such that all two-hop neighbors receive all forwarded data packets and (2) to do this with the smallest number of one-hop neighbor MPR nodes so that the number of retransmissions of data packets is minimized. Much research has been conducted on how to optimally select MPR nodes [10].

To demonstrate how MPR nodes can reduce the amount of traffic in a network, again consider the network shown in Figure 2.7. When device c joins the network, it must have a way to broadcast TC messages to the other seven devices. For the moment, assume that any device in the network – not just MPR nodes – can forward a broadcast message. In this case, device c would broadcast its message to device b, and b would rebroadcast it to devices a, e, and c. Then a would rebroadcast it to e, and b and e would rebroadcast it to d, f, g, a, b, and h. Then both g and f would rebroadcast it to h, and h would forward it to both g and f. As demonstrated, there is much redundancy and consequently bandwidth is consumed for no additional gain. This results in eight total broadcasts for every TC message, but with MPR nodes only two broadcasts by b and e are required for every TC message.

2.5 OLSR Overhead

All the computation described in the previous sections of this chapter must be done at regular intervals in order to account for changes in the network. Even if all devices on the network are idle and no two are communicating with each other, the control traffic still flows throughout the network. This is necessary to ensure that all possible routes are calculated before any two nodes communicate and, thus, reduce message latency. This creates overhead and adds an extra burden to the devices in the network, which in turn adversely affects battery life.

Chapter 3

Related Work

In this chapter we review related work, i.e., research that has been done with regard to understanding the impact of OLSR parameter settings on network performance. We compare our work with each contribution.

In [5], for four sets of OLSR parameter values, the authors experimentally observe the packet loss period resulting from mis-routing. The authors refer to this measure as Route Change Latency (RCL). The average RCL measured in a set of experiments with real devices networked using OLSR was on the order of several seconds. In contrast to the work presented in [5], which did not isolate individual OLSR parameters, our study isolates the *HELLO* message hold time parameter and uncovers its relationship to the *HELLO* message transmission interval w.r.t. number of packet losses.

The authors of [6] present the results of a simulation study conducted to determine how the length of the *HELLO* message and TC message transmission intervals impact network performance. The simulation study varied the density of the network and the speed at which devices move within the network. Network throughput and control message overhead were the performance measures observed. The authors found that with increased device movement there was an increased benefit to adjusting the length of the *HELLO* message transmission interval, i.e., there was increased throughput with a shorter *HELLO* message transmission interval. This is a rather straightforward outcome. The authors found no change in throughput behavior when the device density was increased. Finally, the authors found that the length of the TC message transmission interval had almost no effect on performance. The range of TC message transmission intervals (1 to 10 sec) the authors explored was very likely not wide enough to observe the performance impact of the TC message transmission interval. This study relied on simulation experiments and

did not study the *HELLO* message hold time, while our study utilizes experiments conducted on a physical testbed and focuses on the impact on performance of the *HELLO* message hold time.

The results of another simulation study similar to the one conducted by the authors of [6] are presented in [4]. In this study the authors simulate a MANET that uses OLSR routing with voice traffic between a client device and server device at the edges of the MANET. The authors provide no information about the mobility characteristics that were used in the simulation study. The impact of the length of the *HELLO* and TC message transmission intervals on control message volume and maximum route setup time were studied in isolation. First, the length of the TC interval was fixed to 5 sec, while the length of the *HELLO* message transmission interval was varied among the set { 0.5, 1, 2, 4, 8, 10 } sec. Next, the length of the *HELLO* message transmission interval was fixed to 2 sec, while the length of the TC interval was varied among the same set used for the length of the *HELLO* message transmission interval. For larger *HELLO* message transmission intervals, their results show a negligible reduction of control message volume and a linear increase in the maximum route setup time. For larger TC message transmission intervals, their results show a considerable reduction of control message volume and a marginal linear increase in the maximum route setup time. The findings presented indicate that the length of the *HELLO* message transmission interval has the strongest influence on maximum route setup time, while the length of the TC message transmission interval has the strongest influence on the control message volume. Our study complements this one by focusing on the *HELLO* message hold time and using a physical testbed rather than simulation.

Finally, in [9] the authors propose a mechanism to expedite the process of discovering when two devices are no longer neighbors. Normally two devices discover they are no longer neighbors when their previous neighbor discovery messages expire without receiving any new messages to refresh the neighbor discovery. We will discuss this in greater detail when we explain our experimental results in Chapter 5. This process of discovering a disconnection between two former neighbors leads to a latency in the topology being updated. The authors propose lowering the transmission range of neighbor discovery messages compared to the range used for data packets. In this way, devices will stop receiving neighbor discovery messages before they completely lie

out of each other's transmission range. Since the transmission range for neighbor discovery messages and data are not likely to differ very significantly in practice, this technique is likely to have limited effectiveness. Specifically, it is likely to only be effective when the device movement in a network is quite slow. Our study seeks to expedite the process of discovering when devices are no longer neighbors by identifying the smallest value of the *HELLO* message hold time for which there is a favorable average packet loss period and overall packet loss rate.

For a very recent article on the general problem of neighbor discovery, see [14]. Refer to [11] for one of the first articles on the general problem of neighbor discovery.

Chapter 4

Experimental Methodology

In this chapter we describe the experimental methodology used in this thesis. As mentioned in Chapter 1, our long-term goal is the development of an adaptive OLSR-based routing protocol such that parameters are automatically configured based on knowledge of device mobility. To develop such a protocol requires an understanding of how the setting of OLSR parameters impacts network performance; and this thesis explores how the value of one of these parameters, the HELLO_VALIDITY time (δ), affects network performance in terms of message latency and packet loss. The experiments conducted to quantify these effects are detailed in Section 4.1, along with the performance metrics employed. To perform such experimentation requires a physical testbed, a simulation model, and/or a mathematical model. As described in Section 4.2, as part of the research associated with this thesis, a physical testbed, comprised of six devices, was designed and built. Since our research focuses on MANETs and the OLSR protocol, the devices in the testbed communicate via a wireless network using the OLSR protocol, and the network topology changes over time – some devices are kept in range, while others are not. To simulate this device mobility and the associated changes in the network topology and link losses over time, without actually moving the devices, we use the Linux firewall `iptables` as explained in Section 4.2.1. Data is transmitted over the network using gnu `ping`, which is discussed in Section 4.2.2. Finally, Section 4.3 describes how we stored and processed the large amounts of resultant experimental data and Section 4.4 presents the scripts used to facilitate these experiments, which are included in Appendix C.

4.1 Experiments

This thesis reports the results of fifteen experiments that were conducted to quantify how the value of one of the OLSR parameters, HELLO_VALIDITY (τ) time, affects network performance in terms of message latency and packet loss. These metrics are precise indicators of the impact of incorrect routing, also called mis-routing. When a packet is mis-routed due to incorrect topological information it either: (a) reaches its destination by taking an incorrect longer path through the network, or (b) does not reach its intended destination and is, therefore, lost. Scenario (a) results in increased packet delay and scenario (b) results in an increased packet loss rate. (Note that additional experiments were conducted to ascertain experimental parameters such as run-length described below.)

The experiments explore a space of three different lengths of the HELLO message transmission interval (δ): 2, 4, and 8 sec(sec), each coupled with five different values of HELLO_VALIDITY time (τ): $\delta, 1.1\delta, 1.25\delta, 1.5\delta$, and 2δ, for a total of 15 experiments. The default value of δ for our implementation, olsrd, is 2 sec, and we expect that τ should be larger than but relatively close to δ. The parameters for the TOPOLOGY_CONTROL and TOPOLOGY_VALIDITY intervals were fixed at their default values of 5 and 300 sec, respectively.

For each experiment, four independent trials (replications) were executed. A replication employed a unique set of nine network topology transitions (presented in Appendix A) and a transition was initiated every 120 sec. The set of topologies used for the first replication was designed such that for each transition, only one device changed position. The other three sets of topologies were generated randomly keeping the network connected. In order to perform a run-length of more than nine topology transitions, the transitions go from one to nine, then back down to one and repeat until the number of desired transitions is made. The run-length of a replication was 160 topology changes, i.e., 19,320 sec or 320 minutes. This run-length was selected by conducting a series of experiments from the set of 20, 40, 80, 160, 320, and 640 topology changes and identifying the smallest number of topology changes for which steady-state behavior of the average period of packet loss was observed.

4.2 Physical Testbed

The physical testbed that we designed and built to conduct these experiments as well as follow-on research is comprised of six devices:

- 2 PandaBoards each with:
  - Armv7 Processor rev 3 (v7l)
  - 32GB SD card for storage
  - 1GB of ram
- 1 Optiplex GX790
  - Intel® Core™ i5-2400 CPU @ 3.10GHz
  - 8GB of ram
  - 500GB HDD
- 3 Optiplex GX620
  - Intel® Pentium® 4 CPU 3.40GHz
  - 1GB of ram
  - 80GB HDD

The PandaBoards have their own wireless devices, but in order to ensure wireless device uniformity, each of the machines is equipped with a TP-Link wireless card (TL-WN722N) attached via USB. Ubuntu 12.04.2 LTS was used as the Operating System on each of the devices. This particular version was chosen because it works for the PandaBoards.

As described earlier, since our research focuses on MANETs and the OLSR protocol, the devices in the testbed communicate via a wireless network using the OLSR protocol, in particular olsrd [13]. This implementation of the OLSR protocol was designed to be highly portable and can work on many different operating systems and any wireless card. It has the capability of

using link quality metrics when constructing routes. In addition, devices should be able to move and, accordingly, the network topology should change over time. To simulate this device mobility and the associated changes in the network topology and link losses over time, without actually moving the devices, we use the Linux firewall `iptables` as explained next.

4.2.1 Topology Definition and Transitions

A specific network topology is effected using `iptables`[12], a tool provided by the Linux kernel that allows for the filtering and Network Address Translation (NAT) of data packets. `iptables` consists of rules, called chains or tables, that specify the operations to be carried out for specified packets. Every packet that traverses through a Linux machine travels through at least one chain of rules. Figure 4.3 shows how a packet traverses through the different tables. As shown, a packet entering a host running Linux traverses first through the PREROUTING chain of the raw, mangle, and nat tables. After it has gone through those three chains, a routing decision is made. If the packet is destined for this host, it then takes the path through the INPUT chain of the mangle and filter tables, then to the local process. Otherwise, if a packet is to be sent from this host to another, the packet starts at the local process, a routing decision is made, and then the packet goes to the OUTPUT chain of the raw, mangle, nat, and filter tables. If instead the packet comes in from the network and is destined for another machine other than this one, it goes through the FORWARD chain of the mangle and filter tables. In either of these cases, i.e., a packet is being sent or forwarded from this host, then another routing decision is made and the packet continues through the POSTROUTING chain of the mangle and nat table and then into the network.

IP vs. MAC Addresses

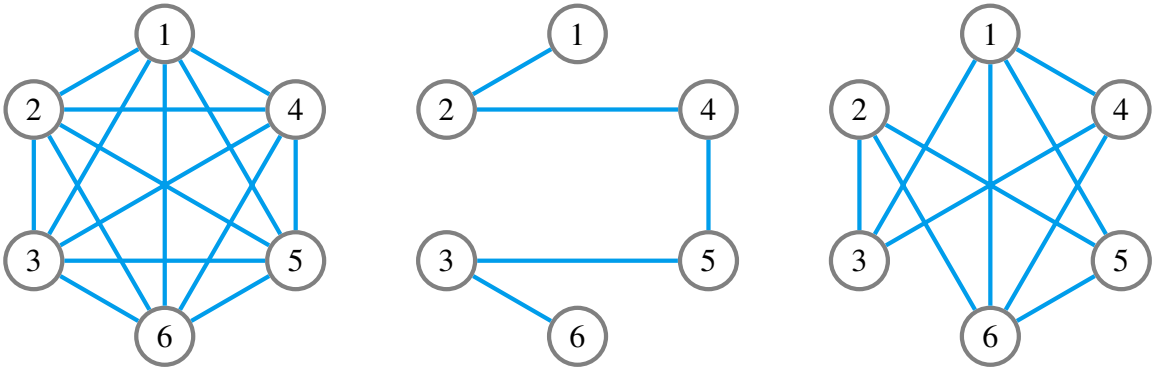
Before defining precisely how `iptables` was used to effect a topology in the physical network, it is worth noting the differences between device IP and MAC addresses. Section 4.2.1 shows how `iptables` rules are populated, but for now assume that the desired topology illustrated in Figure 4.1b has already been setup. Further assume that all routing tables have been populated

by the olsr routing protocol to allow every device to communicate with every other device in the network. Now consider how a link-layer *frame* changes when traversing the network from device 1 to device 6. First, the message is packed into a packet with the source IP address for device 1 and the destination IP address for device 6. As described in Section 1.1.1, device 1 determines where to send the packet by locating device 6 in the dest field of its routing table; it finds that the packet should be forwarded to device 2. The packet is then packed further into an 802.11 frame that contains the MAC address for device 1 as the source and the MAC address for device 2 as the destination, and transmits it. Device 2 then examines the destination IP address and determines that the packet should be forwarded to device 4. Again, the packet is packed into an 802.11 frame, but this time with the MAC address of device 2 as the source and the MAC address of device 4 as its destination. This process continues until the packet reaches the destination IP address, i.e., that of device 6. During this traversal through the network, the source and destination MAC addresses of the 802.11 frame change after every hop, but the source and destination IP addresses of the network layer packet do not. This allows for filtering frames at the link-layer to simulate a loss on a desired link as described in the following section.

Simulating the Topology Transitions

To define a specific network topology, all hosts' iptables rules are configured to drop all frames that contain specific source MAC addresses at the raw INPUT chain. Dropping frames at this chain will prevent any network information from being processed by OLSR or any other processes running on the host. Figure 4.1 shows how the devices of the physical platform are arranged. Since all devices are in range, all devices are connected and can communicate directly as shown in Figure 4.1a. Now assume that we want to transition to the topology shown in Figure 4.1b. Instead of storing the desired topology, the complement topology, which is shown in Figure 4.1c, is stored in each host of our physical testbed and used by `iptables` to filter all traffic that attempts to cross any of those links. To see how the complement topology is stored in code, see the definition of `topology9` in the file `topologies.py` found in Appendix C.

To illustrate how `iptables` can be used to transition to a specific topology, we use the exam-



(a) All devices are within range and, therefore, are connected to each other.

(b) Desired topology

(c) Complement of the desired topology. These are the links that `iptables` blocks in order to create the desired topology.

Figure 4.1: How `iptables` is used to effect the topology transitions.

ple in Figure 4.1b, where the transition is from Figure 4.1a to Figure 4.1b. Assume that device 1 wants to communicate with device 4 and no routing protocol is running on any of the devices. When device 1 creates its packet, it will send it out on the wireless network with the destination specified as device 4. Since all devices are within range, the packet will be examined by device 4 as well as devices 2, 3, 5, and 6 even though the packet is not destined for them. Upon examination, devices 2, 3, 5, and 6 will determine that the packet contains the IP address of device 4 and will drop the packet; however, device 4 will process the message and pass it through to the local process. To simulate the transition to the topology shown in Figure 4.1b, device 4 should drop the packet as well. To do this, the `iptables` on device 4 is setup with rules in the raw table to drop any frames that contain a MAC address from devices 1, 3, or 6 as shown on the fourth row of Figure 4.2. These links, as well as the others in the complement topology shown in Figure 4.1c,must be blocked by the iptables of the host in order to realize the topology. When this is done, when device 1 transmits a frame containing the packet m with source MAC address 1, it will be dropped by device 4 before it has a chance to reach the local process and device 1 will timeout after a period of time specified in the Linux network stack.

For each desired topology G, the complement G^C is stored on each device in a python dictionary or hash-map, which is presented in Appendix C in the file `topologies.py`. After a

Device	Source MAC Address					
	1	2	3	4	5	6
1			drop	drop	drop	drop
2			drop		drop	drop
3	drop	drop		drop		
4	drop		drop			drop
5	drop	drop				drop
6	drop	drop		drop	drop	

Figure 4.2: Representation of `iptables` rules used to affect the topology shown in 4.1b.

topological change period $\Delta = 120$ sec, the topology of the network is changed, i.e., each device involved in the topology change clears all of the rules it has for the current topology, looks into the dictionary that contains the complement graph of the next topology and updates its rules accordingly. Figure 4.2 shows a simplified representation of what the `iptables` rules would look like to affect the topology shown in Figure 4.1b. This is basically an adjacency matrix for the complement topology pictured in in Figure 4.1c, where a "drop" entry indicates that traffic from the associated source MAC address to the associated device should not be allowed to pass through the firewall, i.e., the link does not exist. Accordingly, the links shown in Figure 4.1b are indicated by entries that are blank i.e. traffic that arrives at an associated device from an associated source MAC address should be allowed to through the firewall, i.e., the link exists.

Next we describe how we generate data traffic.

4.2.2 Data Transmission

Data traffic is generated using gnu `ping`. `ping` sends out an `icmp` echo request to a destination host on the network and then waits for an `icmp` echo reply. When the destination host receives the echo request, it responds with an `icmp` echo reply. The source host records the time of the exchange, i.e., starting from the time the echo request was sent and ending with the time the echo reply was received from the destination host. This time is the *round-trip time* (rtt) and is used to determine the packet delay. Each echo request contains a sequence number n and each echo

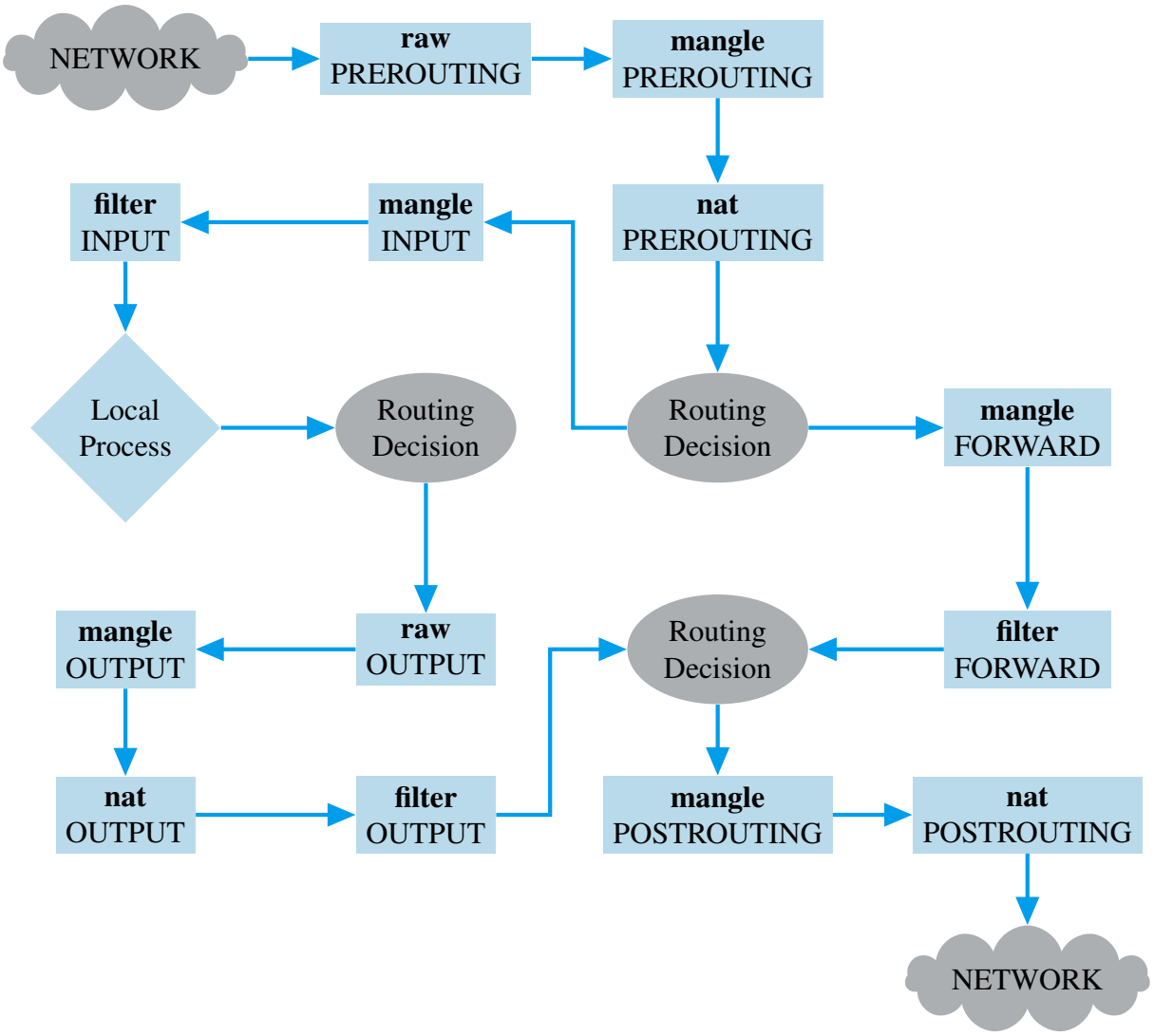


Figure 4.3: How packets traverse through `iptables`

.

reply contains the same sequence number n as the request that initiated the reply. This sequence number is used to determine if an echo request/reply pair was able to traverse a path in the network between a pair of hosts. If an echo request with sequence number n is transmitted and a reply is not received within a time period t (38 msec for these experiments), no attempt to re-transmit the packet is made and the packet is considered *lost*. Another echo request is then transmitted with sequence number $n + 1$. If there is no reply to the request with sequence number $n + 1$ before the timeout value, this packet is also considered lost and another request with sequence number $n + 2$ is transmitted. This process continues until m packets are lost, i.e., until a reply to request $n + m + 1$ is received. The period of time between when request n is transmitted and when reply $n + m + 1$ is received is the *route-change latency* (RCL) [5] or *packet loss period*. This measure can be used to measure the time it takes for the devices running olsrd to update their routes after a change in network topology is made. The only upper limit on the percentage of packets lost is 100%.

The experiments conducted in this thesis create a flow of pings between every pair of devices in the network. There are a total of six devices, every device creates five flows, which results in a total of $6 \times 5 = 30$ flows of pings. Each ping message contains 56 bytes of data and eight bytes for the icmp header for a total message size of 64 bytes. For each path, a ping was transmitted once every second and the timeout for each ping was fixed at 38 msec. Originally `ping` would exit as soon as it received an `icmp` error, but this behavior was undesirable for the purposes of this thesis. It is expected that some errors should be generated for certain parameter configurations and `ping` should continue transmitting, thus, the source code of `ping` was modified to effect this behavior. Since the run-length of a single replication was 19,320 sec, `ping` was configured to exit after 19,320 sec regardless of the number of pings actually transmitted.

4.3 Experimental Data Processing

Originally the data were collected and stored in ASCII format. This proved to be an inefficient means of storing the resultant experimental data, thus, a binary format was selected. The hierar-

chical data format (HDF5) was chosen since it is widely used and takes care of different endianess automatically. Data in an HDF5 file are stored in either a dataset or a group and it supports adding attributes to groups and datasets to help explain what the data represent.

The resulting data for an experiment is grouped in the HDF5 file and the olsr parameters that were used are stored as attributes. For each experiment a dataset is created which for each data transmission records the following:

- timestamp (t) of when an icmp echo request was successfully received,
- size of the packet,
- sequence number, and
- round-trip time (rtt), recorded by the source host (described in the previous section), which is used to calculate the packet delay.

To determine the packet loss period, sequence numbers associated with ping echo requests and replies are used to count the number of consecutive packets lost. The loss period, in terms of time, is calculated by taking the forward finite difference of the sequence numbers and subtracting 1. For example, assume that `ping` is configured to transmit a ping once every 2 seconds and the following sequence numbers $\{1,2,3,8,9,15,16\}$ were recorded. This sequence list would have a forward finite difference of $\{1,1,5,1,6,1\}$. Then 1 is subtracted element-wise resulting in a list of $\{0,0,4,0,5,0\}$ which represents two packet loss periods of 8 sec and 10 sec.

Given the number of packets lost within each loss period and the time length of each loss period of an experiment, a bin was created for each loss period time length. Each bin maintains a count of the number of times a specific packet loss period occurred. The histograms of each of the 30 paths were then quantized by the bin count. The result is a two-dimensional histogram shown in Figure 5.1, where the y-axis represents the probability that a packet loss period of x sec will occur. The darker the color, the more often that loss period with that probability occurred among all of the paths in the network.

4.4 Scripts to Initiate Experiments

Two python scripts, presented in Appendix C, were used to conduct the experiments described in Section 4.1. The simplest one, `startExperiment.py` establishes a temporary ssh session with each of the devices in the network and launches the second script `olsrExperiment.py`. `olsrExperiment.py` is the main script; it is responsible for experiment execution. Its tasks are:

1. Load a configuration file that contains the setup for a set of experiments;
2. Start olsrd with the parameters as defined in the configuration file;
3. Establish a ping session with every other host on the network;
4. Record the data associated with an echo request/reply pair (described in Section 4.2.2), namely:
   (a) timestamp,
   (b) size,
   (c) sequence number, and
   (d) round-trip time (rtt);
5. Coordinate the simulation of broken links with `iptables`; and
6. Send an e-mail notification when the experiment set has finished.

The configuration file mentioned above is another python script that stores the parameters for the HELLO_INTERVAL, HELLO_VALIDITY, TC_INTERVAL, and TC_VALIDITY times. The parameters for each of these is stored as a list so that different combinations of settings can be used for different experiments automatically.

To run an experiment, the following process is followed:

1. Using the configuration file, specify the parameters (see Section 4.1) for olsrd and ping;

2. Execute the `sync.sh` shell (see Appendix C) script, which uses rsync to place all the scripts and configuration files on all of the devices in the testbed;

3. Run the `startExperiment.py` script, which launches `olsrExperiment.py`;

4. Once `olsrExperiment.py` is running on all of the devices, each will do the following:

   (a) Start olsrd with the parameters specified in the configuration file;

   (b) Configure the `iptables` rules for the first topology in the sequence;

   (c) Wait for a random time between 30 and 60 sec to allow olsrd to reach a steady state and help ensure that the topology transitions do not align with the times that control messages are transmitted;

   (d) Synchronize with the other devices in the network – note that this synchronization is only for the python scripts and has no effect on the operation of olsrd;

   (e) Start five ping sessions – one for every other device on the network;

   (f) After the topological change time ($\Delta = 120$ sec) has elapsed, again synchronize over the Ethernet network and configure the `iptables` rules for the next topology; and

   (g) When 320 topological changes have been made, collect the data for each ping that was not lost and record the data in the HDF5 file as described earlier.

When this process is complete, the scripts move onto the next parameter configuration and repeat the process described above. Once there are no more configurations to execute, the devices will exit and an e-mail will be sent out to notify the user that the experiments have completed.

Chapter 5

Results

This chapter presents the results of our experiments. First we focus on our findings related to average packet loss period and then on those related to overall packet loss rate. The data regarding packet delay did not reveal anything conclusive so it is not covered here, however, it is presented in Tables 5.11, 5.12, and 5.13.

5.1 Average Packet Loss Period

Tables 5.1, 5.2, and 5.3 show the average packet loss period for HELLO message transmission intervals (δ) of 2 sec, 4 sec, and 8 sec, respectively. Recall that each experiment has four trials (replications) each with a differing topology sequence. Our primary observation from these data is that when the HELLO message hold time (τ) is set to double the HELLO message transmission interval (δ) the average packet loss period increases rather significantly, as compared to when τ equals δ or when τ takes on its minimal value. This occurs because although neighbors are discovered through neighbor discovery message exchanges, two nodes are no longer considered neighbors only after their previous exchange of neighbor discovery messages has expired. This neighbor discovery expiration time is governed by τ, and the larger the value of τ, the longer it takes for a previous neighbor to no longer be considered a neighbor. If mobility results in two previous neighbors no longer being within each other's transmission range, then a large τ will lead to inaccurate topological information for a longer period of time. In that case the average packet loss period increases. At the other end of the spectrum, if $\tau < \delta$, then the neighbor discovery process will expire before it is potentially refreshed by a new exchange of neighbor discovery messages. It is worth noting that the implementation of OLSR that we used, olsrd [13],

Table 5.1: Average packet loss period for each of the four replications with HELLO_INTERVAL (δ) = 2sec; minimum value is highlighted in boldface.

τ (sec)	Average Packet Loss Period (sec)			
	rep. 1	*rep. 2*	*rep. 3*	*rep. 4*
$\delta = 2$	10.02	12.56	**14.73**	**14.31**
$1.1\delta = 2.2$	10.65	**11.69**	15.56	14.84
$1.25\delta = 2.5$	**9.72**	11.89	20.06	17.96
$1.5\delta = 3$	9.95	14.66	18.12	17.22
$2\delta = 4$	15.55	19.53	22.36	19.50

would not even allow us to set $\tau < \delta$.

To check for statistical significance, we conducted pairwise t-tests between all pairs of experiments. Tables 5.4, 5.5, and 5.6 show the results of these tests for 95% and 80% confidence intervals. If the bounds of the confidence interval (see lower and upper bounds of interval in the tables) do not cross zero then there is a statistically significant difference between the two experiments. We show the confidence intervals for both 95% and 80% confidence intervals. Our primary observation is that the differences in average packet loss period become statistically significant as the difference between the τ values increases. Generally, τ values closer to δ result in smaller average packet loss periods. From the pairwise t-test data in Tables 5.4, 5.5, and 5.6 it appears that the absolute difference in τ values has a larger effect on the statistical significance of the differences in average packet loss period than the relative differences. For this reason there are more statistically significant differences for $\delta = 8$ sec than for $\delta = 2$ sec.

A more precise view of the effect of the value of the τ parameter on the period of packet loss is procured by the distribution of these packet loss periods. Figure 5.1 shows this distribution for $\delta = 2$ sec with each of the values of τ for replication 1. Our primary observation is that as τ increases from 2 sec to 4 sec the distribution of packet loss periods shifts from the majority of

Table 5.2: Average packet loss period for each of the four replications with (δ) = 4sec.

τ **(sec)**	**Average Packet Loss Period (sec)**			
	rep. 1	*rep. 2*	*rep. 3*	*rep. 4*
$\delta = 4.0$	16.36	**17.70**	24.09	**22.09**
$1.1\delta = 4.4$	17.09	19.09	24.72	27.57
$1.25\delta = 5$	**15.66**	20.73	**22.45**	27.78
$1.5\delta = 6$	18.35	25.30	24.92	29.56
$2\delta = 8$	26.11	28.01	30.72	30.78

Table 5.3: Average packet loss period for each of the four replications with (δ) = 8sec.

τ **(sec)**	**Average Packet Loss Period (sec)**			
	rep. 1	*rep. 2*	*rep. 3*	*rep. 4*
$\delta = 8$	22.34	19.11	27.06	24.49
$1.1\delta = 8.8$	**15.55**	**16.51**	**20.77**	**22.27**
$1.25\delta = 10$	21.45	26.54	37.98	31.98
$1.5\delta = 12$	26.65	32.25	35.20	37.48
$2\delta = 16$	35.37	36.38	42.13	34.50

Table 5.4: Pairwise t-test between pairs of τ values with (δ) = 2sec; statistically significant pairs highlighted in boldface.

		95% confidence		**80% confidence**	
τ_1 (sec)	**τ_2 (sec)**	**lower bound**	**upper bound**	**lower bound**	**upper bound**
2.0	2.2	−3.12	3.67	−1.72	2.27
2.0	2.5	−3.67	7.66	−1.33	5.33
2.0	3	−2.42	6.58	−0.57	4.73
2.0	4	**2.59**	**10.06**	**4.13**	**8.52**
2.2	2.5	−4.05	7.49	−1.67	5.12
2.2	3	−2.83	6.43	−0.92	4.53
2.2	4	**2.16**	**9.95**	**3.76**	**8.34**
2.5	3	−6.40	6.56	−3.73	3.89
2.5	4	−1.65	10.30	**0.81**	**7.85**
3	4	−0.64	9.13	**1.37**	**7.12**

the weight of the distribution on shorter periods to a majority of the weight spread across a wider spectrum of loss periods including relatively large loss periods.

As τ increases, the longer it takes for a previous neighbor to no longer be considered a neighbor; i.e., the longer inaccurate topological information persists. As a result, there are longer periods of mis-routing and concomitant longer packet loss periods.

When $\tau \leq \delta$ or even when τ has a value close to δ, neighbor discovery information can expire before it is refreshed. This results in the possibility of two nodes that are still within transmission range not being considered as neighbors for a short period of time. During this short period of time the topological information is inaccurate, which leads to mis-routing for a small period of time. We refer to this phenomenon as *neighbor flapping*.

Table 5.5: Pairwise t-test between pairs of τ values with (δ) = 4sec.

		95% confidence		**80% confidence**	
τ_1 **(sec)**	τ_2 **(sec)**	**lower bound**	**upper bound**	**lower bound**	**upper bound**
4.0	4.4	−4.38	8.49	−1.73	5.84
4.0	5	−4.95	8.14	−2.26	5.45
4.0	6	−1.76	10.71	**0.80**	**8.14**
4.0	8	**4.31**	**13.39**	**6.17**	**11.52**
4.4	5	−7.85	6.93	−4.81	3.89
4.4	6	−4.70	9.53	−1.77	6.60
4.4	8	**1.10**	**12.48**	**3.45**	**10.14**
5	6	−4.34	10.09	−1.37	7.12
5	8	**1.44**	**13.07**	**3.83**	**10.67**
6	8	−1.09	9.83	**1.16**	**7.59**

Table 5.6: Pairwise t-test between pairs of τ values with (δ) = 8sec.

		95% confidence		**80% confidence**	
τ_1 **(sec)**	τ_2 **(sec)**	**lower bound**	**upper bound**	**lower bound**	**upper bound**
8	8.8	−9.43	0.48	−7.39	−1.56
8	10	−2.10	14.57	**1.33**	**11.14**
8	12	**3.54**	**15.75**	**6.05**	**13.24**
8	16	**8.74**	**18.95**	**10.84**	**16.85**
8.8	10	**2.43**	**18.99**	**5.84**	**15.58**
8.8	12	**8.08**	**20.15**	**10.57**	**17.67**
8.8	16	**13.31**	**23.33**	**15.37**	**21.27**
10	12	−5.61	12.43	−1.90	8.71
10	16	−0.76	15.98	**2.69**	**12.53**
12	16	−1.95	10.36	**0.58**	**7.82**

Table 5.7: Overall percent packet loss for each of the four replications with (δ) = 2sec.

τ **(sec)**	**pct. pkt. loss (%)**			
=	*rep. 1*	*rep. 2*	*rep. 3*	*rep. 4*
$\delta = 2.0$	15.10	17.24	27.75	23.53
$1.1\delta = 2.2$	11.82	16.27	20.99	20.46
$1.25\delta = 2.5$	14.04	13.52	24.42	23.80
$1.5\delta = 3$	7.24	**11.78**	**20.39**	22.67
$2\delta = 4$	**4.32**	12.07	22.36	**19.50**

5.2 Overall Packet Loss Rate

To more closely observe the consequences of the neighbor flapping phenomenon we look at the overall packet loss rate. Tables 5.7, 5.8, and 5.9 show the overall packet loss rate for HELLO message transmission intervals (δ) of 2 sec, 4 sec, and 8 sec, respectively. Our primary observation from this data is that the packet loss rate is significantly higher when $\tau = \delta$ compared to the other values for τ. We suspect this to be due to neighbor flapping. To provide more evidence of this we conducted an experiment in which we made no topology changes. In this case, packet loss periods are due to either the neighbor flapping phenomenon or inherent environmental losses due to interference. Table 5.10 shows the overall packet loss rate for this experiment.

The data shows an exponential decrease in the overall packet loss rate as τ increases. The overall packet loss rate drops to a minute 0.71% when τ is 25% larger than δ and settles at 0.28% once τ is 50% larger. These very small loss rates are almost certainly due to environmental losses. Our primary observation is that the neighbor flapping phenomenon disappears when the τ value is at least 25% larger than δ.

Table 5.8: Overall percent packet loss for each of the four replications with (δ) = 4sec.

τ **(sec)**	**pct. pkt. loss (%)**			
=	*rep. 1*	*rep. 2*	*rep. 3*	*rep. 4*
δ = 4.0	15.32	17.32	27.21	24.40
1.1δ = 4.4	9.32	18.95	21.70	24.99
1.25δ = 5	9.73	16.69	**18.29**	26.80
1.5δ = 6	6.96	**16.50**	20.94	29.10
2δ = 8	**5.21**	17.35	22.68	**23.23**

Table 5.9: Overall percent packet loss for each of the four replications with (δ) = 8sec.

τ **(sec)**	**pct. pkt. loss (%)**			
=	*rep. 1*	*rep. 2*	*rep. 3*	*rep. 4*
δ = 8.0	49.87	45.32	52.49	50.04
1.1δ = 8.8	36.12	34.71	39.94	40.65
1.25δ = 10	9.94	**18.49**	30.30	**25.22**
1.5δ = 12	**7.90**	20.82	**25.39**	30.06
2δ = 16	9.37	21.67	29.77	26.05

Table 5.10: Overall packet loss rate for the experiment with no topology changes with (δ) = 8sec.

τ **(sec)**	**overall pkt. loss (%)**
δ = 8	23.84
1.1δ = 8.8	12.44
1.25δ = 10	0.71
1.5δ = 12	0.28
2δ = 16	0.28

Table 5.11: Average packet delay for each of the four replications with (δ) = 2sec.

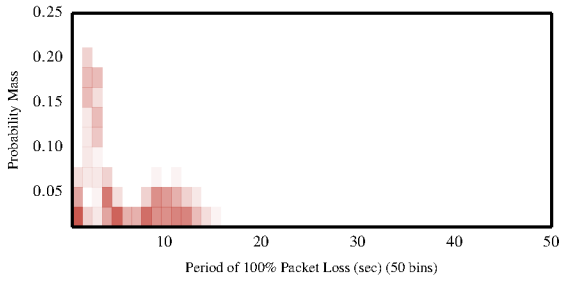
τ **(sec)**		**Average Packet Delay (msec)**			
		replication 1	*replication 2*	*replication 3*	*replication 4*
(δ)	2.0	5.96	**3.82**	4.70	4.03
(1.1δ)	2.2	**4.67**	4.32	**3.95**	4.12
(1.25δ)	2.5	4.71	3.79	4.45	4.75
(1.5δ)	3	4.79	3.60	4.22	4.94
(2δ)	4	5.86	3.44	4.06	**3.85**

Table 5.12: Average packet delay for each of the four replications with (δ) = 4sec.

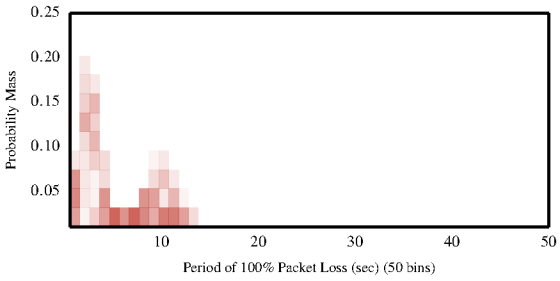
τ **(sec)**		**Average Packet Delay (msec)**			
		replication 1	*replication 2*	*replication 3*	*replication 4*
(δ)	4.0	5.11	4.05	4.52	3.95
(1.1δ)	4.4	**3.97**	3.78	3.79	**3.39**
(1.25δ)	5	4.21	3.84	**3.12**	3.77
(1.5δ)	6	5.01	**2.97**	3.96	4.45
(2δ)	8	6.20	5.07	4.68	4.60

Table 5.13: Average packet delay for each of the four replications with (δ) = 8sec.

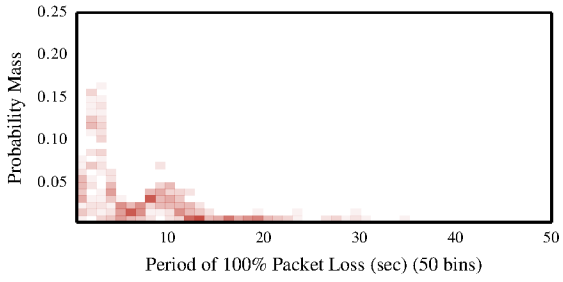
τ **(sec)**		**Average Packet Delay (msec)**			
		replication 1	*replication 2*	*replication 3*	*replication 4*
(δ)	8.0	3.5	3.40	**3.03**	3.43
(1.1δ)	8.8	**2.57**	**3.35**	3.16	3.47
(1.25δ)	10	3.94	3.40	4.06	**3.22**
(1.5δ)	12	3.20	3.87	3.85	3.46
(2δ)	16	6.68	4.78	4.99	4.69



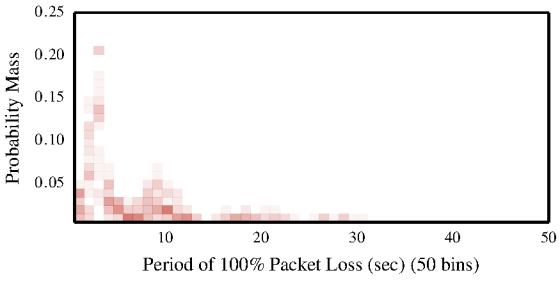
(a) $\tau = \delta = 2\,\text{sec}$



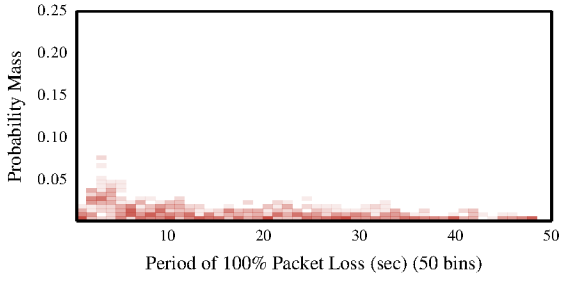
(b) $\tau = 1.1\delta = 2.2\,\text{sec}$



(c) $\tau = 1.25\delta = 2.5\,\text{sec}$



(d) $\tau = 1.5\delta = 3\,\text{sec}$



(e) $\tau = 2\delta = 4\,\text{sec}$

Figure 5.1: Probability of various packet loss periods for various values of τ ($\delta = 2\,\text{sec}$, replication 1).

Chapter 6

Conclusions and Future Work

We constructed a physical testbed that can automatically simulate changes to a network topology over time without the need to physically move the devices. This allows for conducting experiments with a real physical transmission channel and provides a means of conducting repeatable experiments. Since all devices are actually kept in range of each other, a larger amount of collisions can occur as compared to a network where the devices are actually mobile and links are lost due to being out of range with each other.

The testbed was used to conduct a set of experiments to uncover the relationship between the neighbor discovery message transmission interval (also known as the HELLO message transmission interval) and the neighbor discovery message hold time (also known as the HELLO message hold time) w.r.t. the impact on packet loss and packet delay. We found that if the HELLO message hold time is set too close to the HELLO message transmission interval, a phenomenon we refer to as neighbor flapping many short packet loss periods. Recall that when $\tau \leq \delta$ or even when τ has a value close to δ, neighbor discovery information can expire before it is refreshed. This results in the possibility of two nodes that are still within transmission range not being considered as neighbors for a short period of time. During this short period of time the topological information is inaccurate, which leads to mis-routing. At the the other end of the spectrum, if the HELLO message hold time is set too large then it takes an unnecessarily long period of time to remove neighbor links that no longer exist. From our experimental data, it appears that the best tradeoff exists when the HELLO message hold time is set to 25% to 50% larger than the HELLO message transmission interval.

One promising path for future work is to expand the set of parameters studied in the experiments to include, for example, larger HELLO message transmission interval values and relatively

larger HELLO message hold times. A possibly more promising path is to design a set of experiments to investigate the precise impact of the HELLO message transmission interval on network performance.

Bibliography

[1] S. Basagni et al., *Mobile Ad Hoc Networking*, Wiley, 2004.

[2] I. Chlamtac, M. Conti, and J. Liu, "Mobile Ad Hoc Networking: Imperatives and Challenges," in *Elsevier Ad Hoc Networks* 1.1 (July 2003), pp. 13–64.

[3] IETF, *Optimized Link State Routing Protocol (OLSR)*, 2003. URL: http://www.ietf.org/rfc/rfc3626.txt.

[4] S. Demers and L. Kant, "MANETs: Performance Analysis and Management," in *IEEE Military Communications Conference (MILCOM)*, Oct. 2006, pp. 1–7.

[5] C. Gomez, D. Garcia, and J. Paradells, "Improving Performance of a Real Ad-hoc Network by Tuning OLSR Parameters," in *IEEE Symposium on Computers and Communications (ISCC)*, June 2005, pp. 16–21.

[6] Y. Huang, S. Bhatti, and D. Parker, "Tuning OLSR," in *IEEE Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Sept. 2006, pp. 1–5.

[7] P. Jacquet et al., "Optimized Link State Routing Protocol for Ad Hoc Networks," in *Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century. Proceedings. IEEE International*, 2001, pp. 62–68, DOI: 10.1109/INMIC.2001.995315.

[8] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 5th, USA: Addison-Wesley Publishing Company, 2010, ISBN: 0136079679, 9780136079675.

[9] N. Letor and C. Blondia, "Cross-Layer Tuning of the Neighbor Sensing Mechanism in Mobile Ad Hoc Networks," in *International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, Oct. 2010, pp. 784–789.

[10] D. Macone, G. Oddi, and A. Pietrabissa, "MQ-Routing: Mobility-, GPS- and Energy-Aware Routing Protocol in MANETs for Disaster Relief Scenarios," in *Ad Hoc Networks* 11.3 (2013), pp. 861–878, URL: http://dl.acm.org/citation.cfm?id=2459589.

[11] M. McGlynn and S. Borbash, “Birthday Protocols for Low Energy Deployment and Flexible Neighbor Discovery in Ad Hoc Wireless Networks,” in *ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, Oct. 2001, pp. 137–145.

[12] G. N. Purdy, *Linux Iptables - Pocket Reference: Firewalls, Nat and Accounting.* O’Reilly, 2004, pp. I–III, 1–91, ISBN: 978-0-596-00569-6.

[13] A. Tønnesen et al., *olsrd: an Ad Hoc Wireless Mesh Routing Daemon*, Jan. 2013, URL: `http://olsr.org`.

[14] S. Vasudevan et al., “Efficient Algorithms for Neighbor Discovery in Wireless Networks,” in *IEEE/ACM Transactions on Networking* 21.1 (Feb. 2013), pp. 69–83.

Appendices

Appendix A

Topology Sets

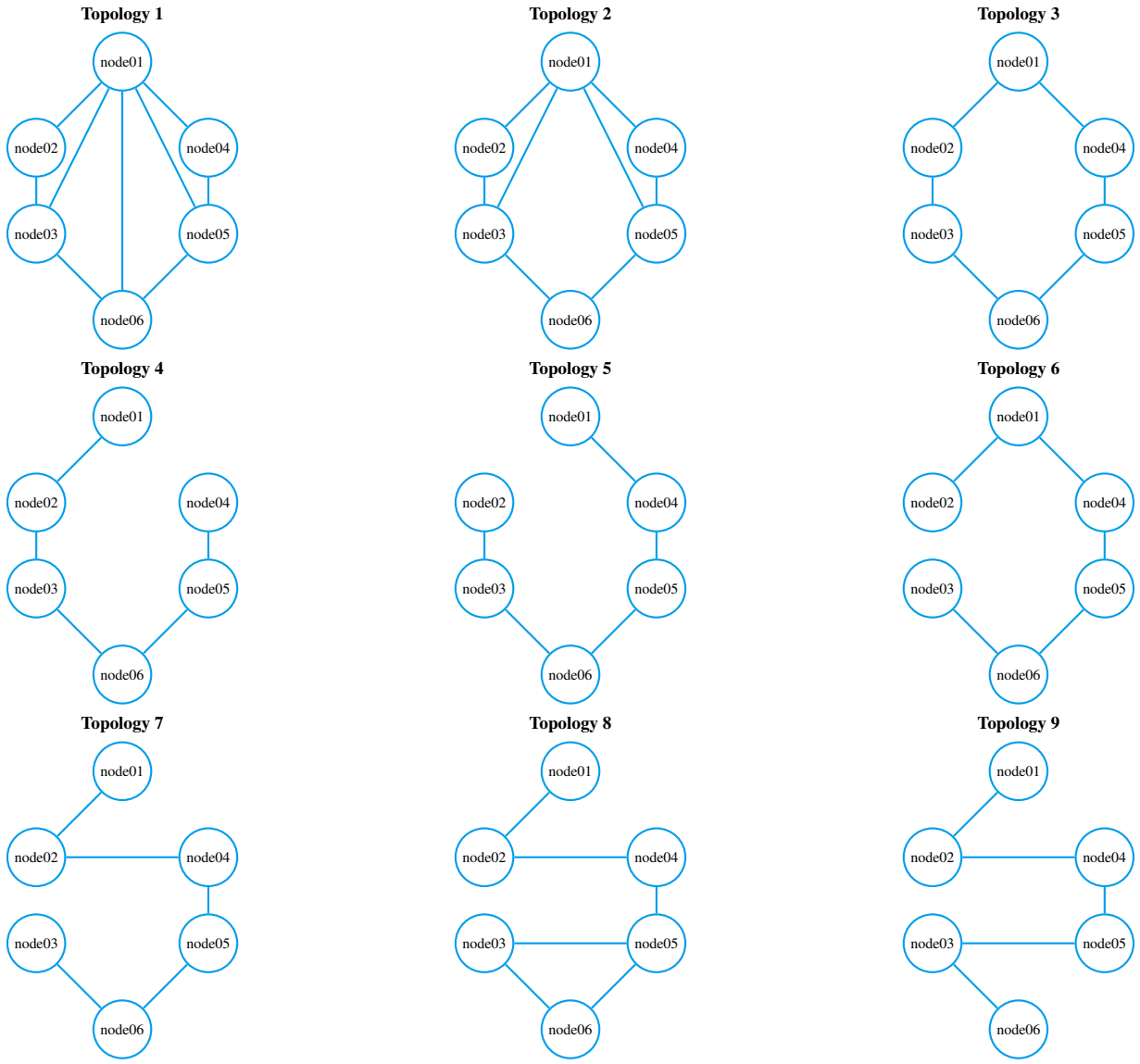


Figure A.1: Topologies for replication 1

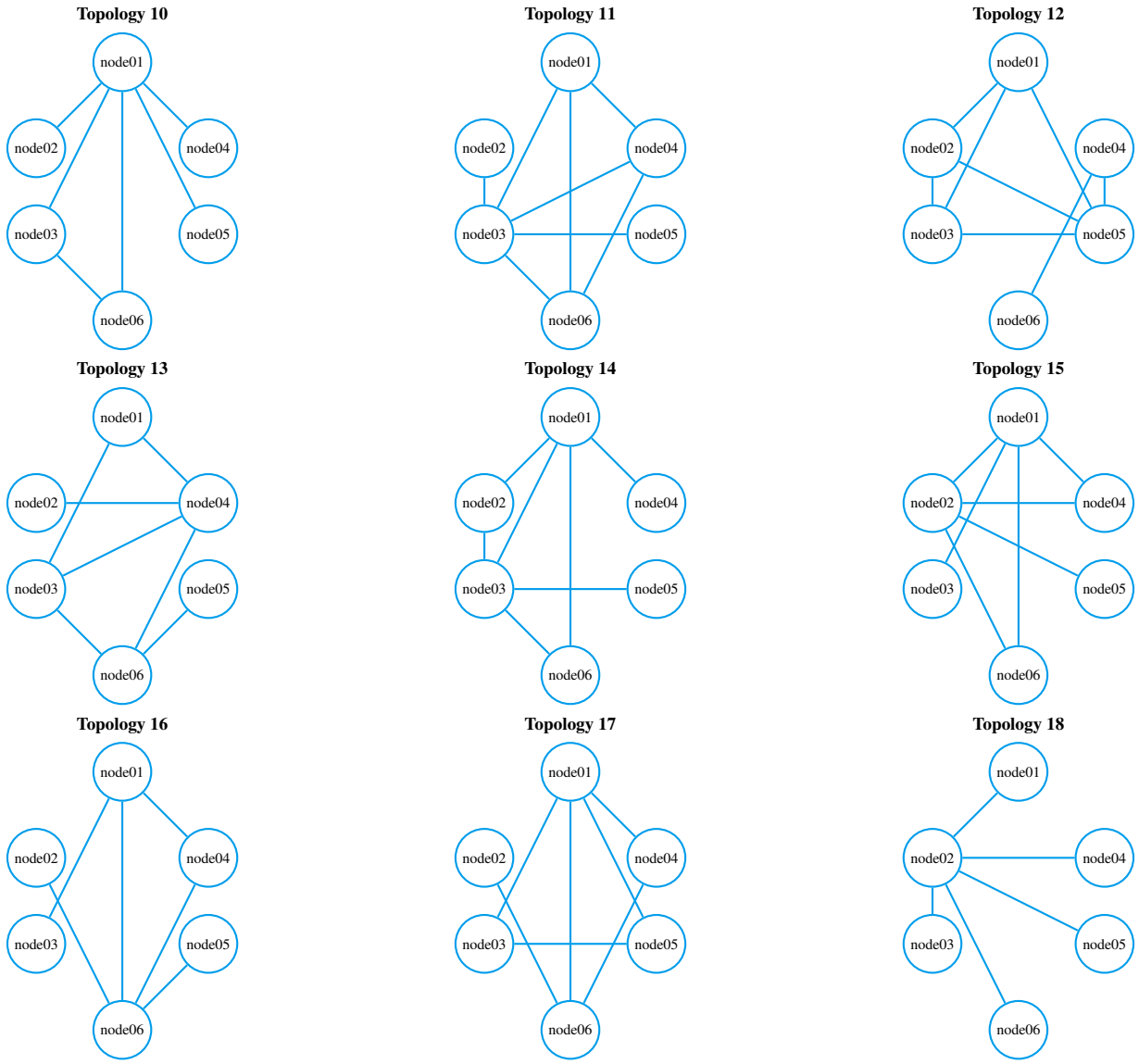


Figure A.2: Topologies for replication 2

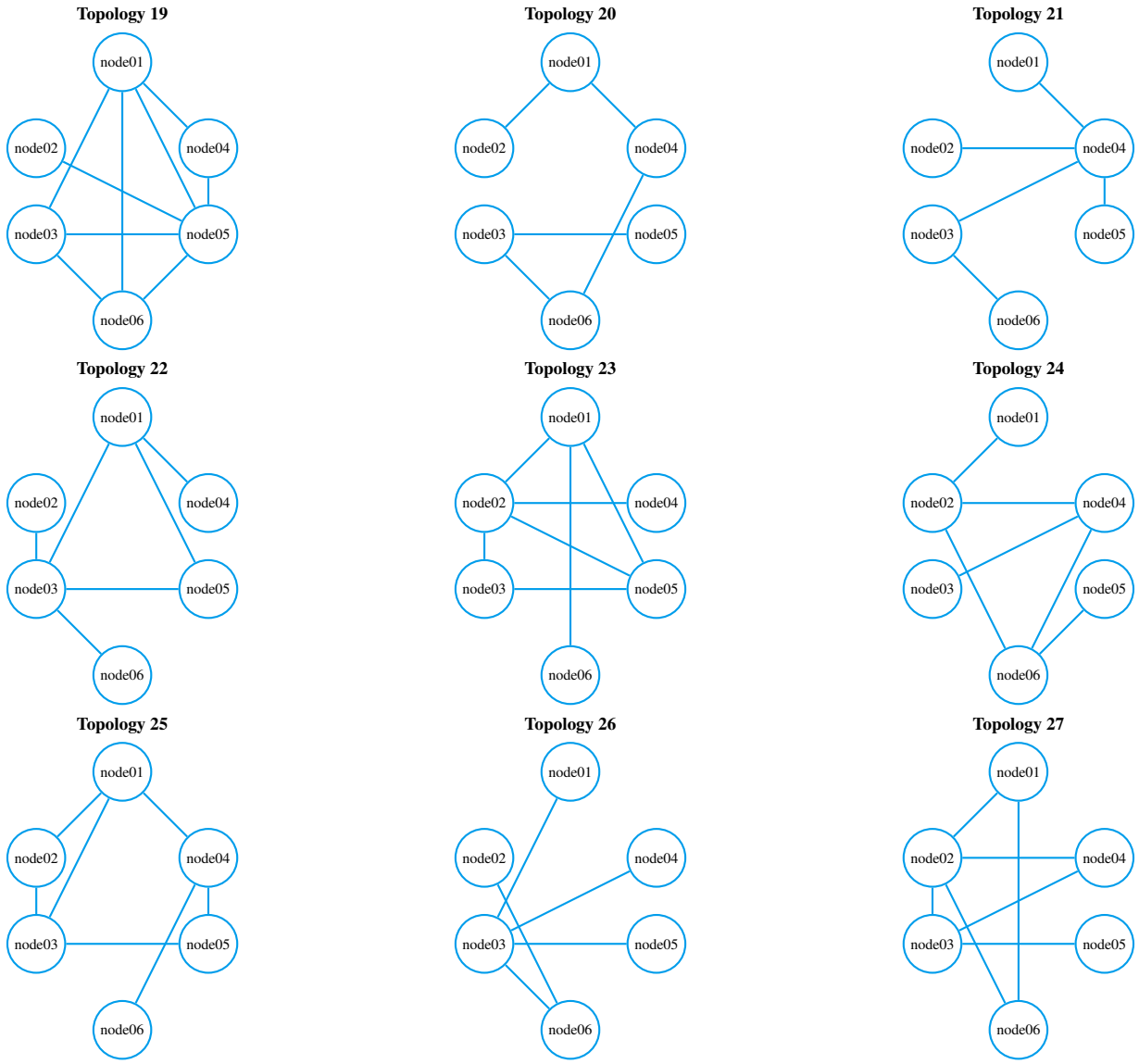


Figure A.3: Topologies for replication 3

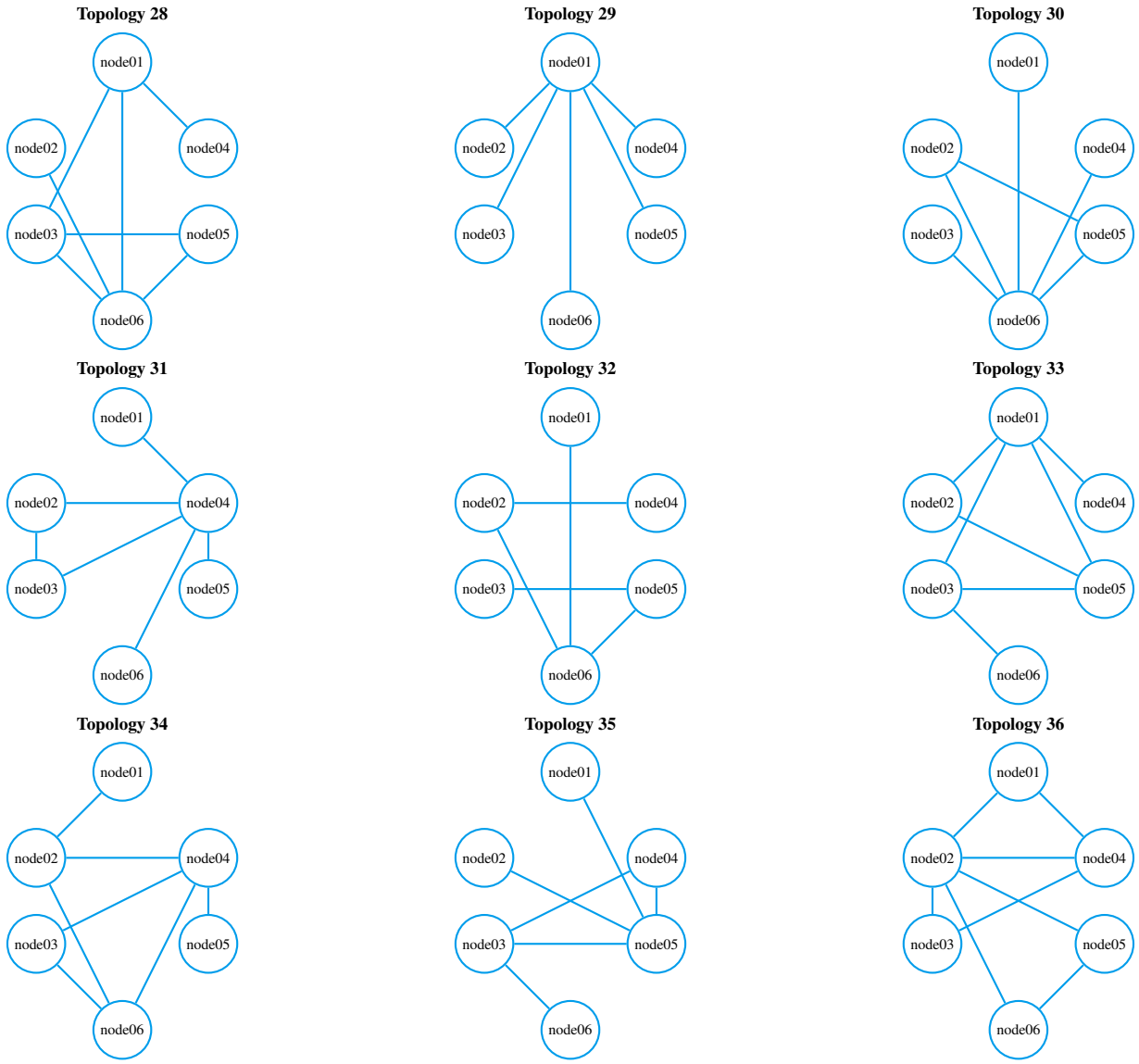
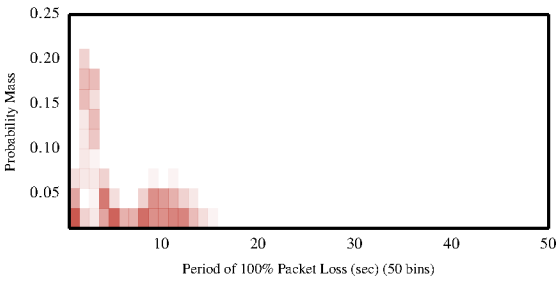


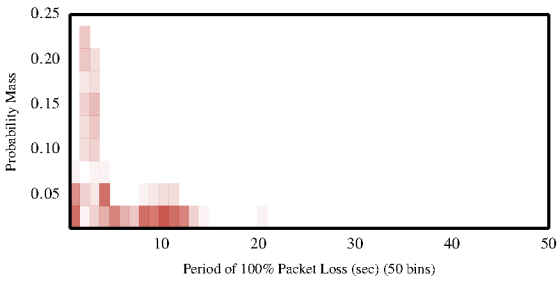
Figure A.4: Topologies for replication 4

Appendix B

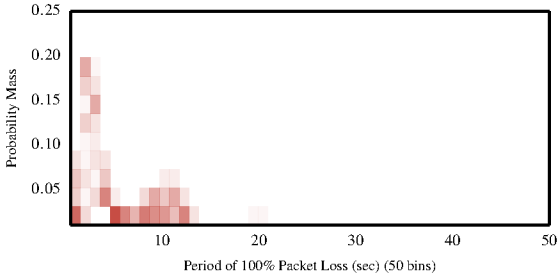
Histograms



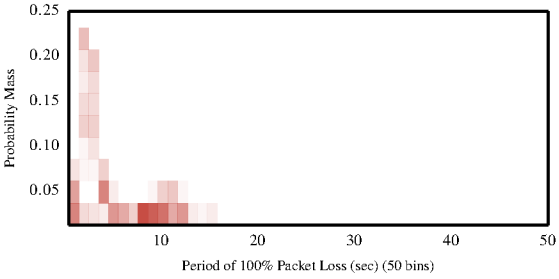
(a) $\tau = 1\delta = 2.0$ sec



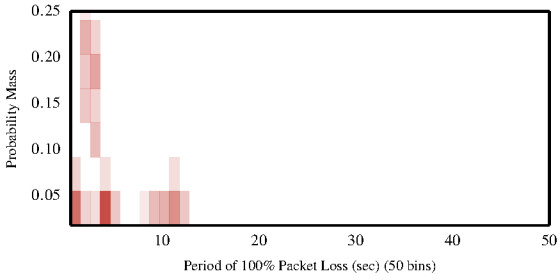
(b) $\tau = 1.0005\delta = 2.001$ sec



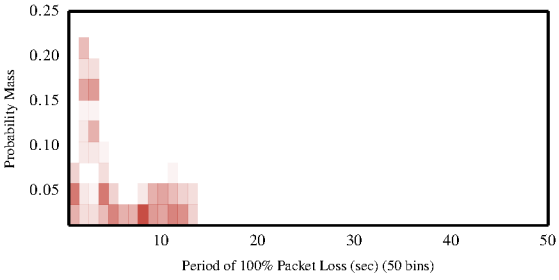
(c) $\tau = 1.0025\delta = 2.005$ sec



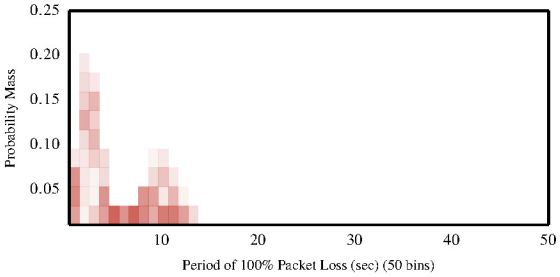
(d) $\tau = 1.005\delta = 2.01$ sec



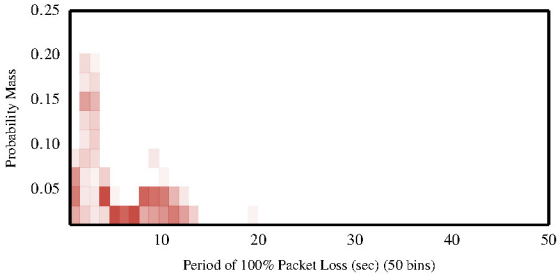
(e) $\tau = 1.025\delta = 2.05$ sec



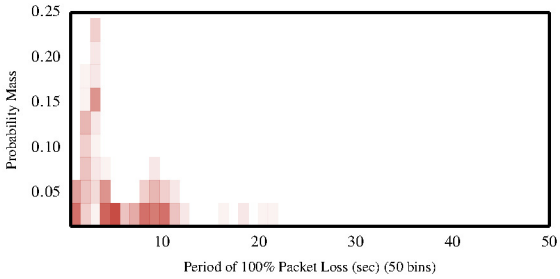
(f) $\tau = 1.05\delta = 2.1$ sec



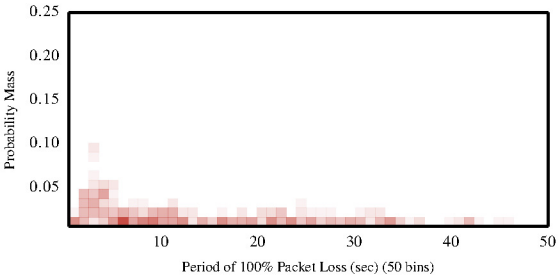
(g) $\tau = 1.1\delta = 2.2$ sec



(h) $\tau = 1.25\delta = 2.5$ sec

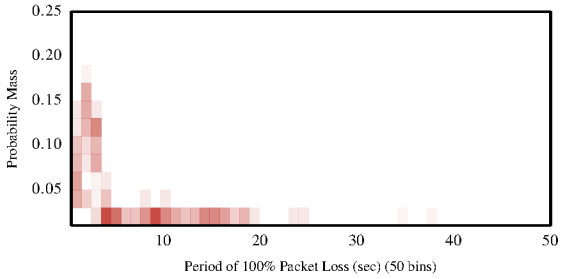


(i) $\tau = 1.5\delta = 3$ sec

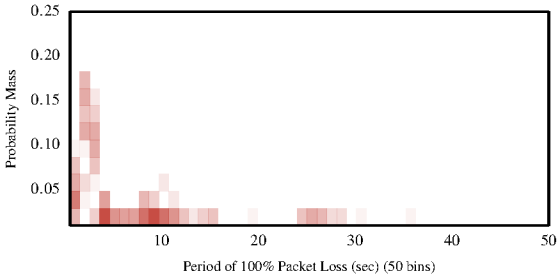


(j) $\tau = 2\delta = 4$ sec

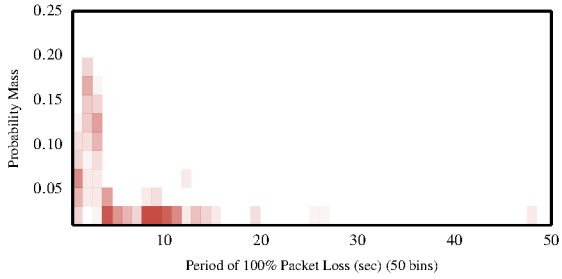
Figure B.1: Probability of various packet loss periods for various values of τ with (δ = 2 sec, replication 1).



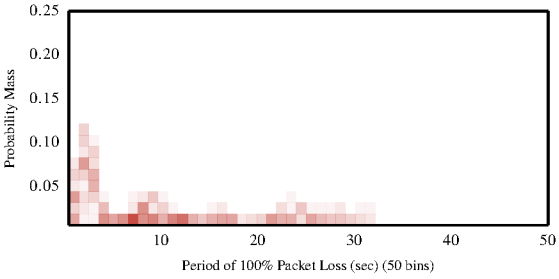
(a) $\tau = 1\delta = 2.0$ sec



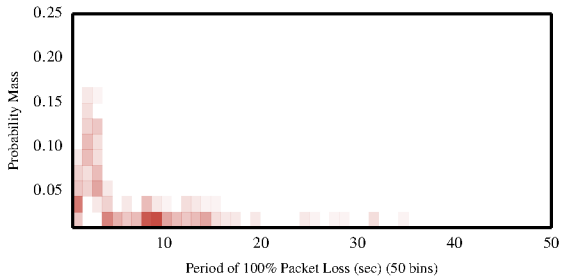
(b) $\tau = 1.0005\delta = 2.001$ sec



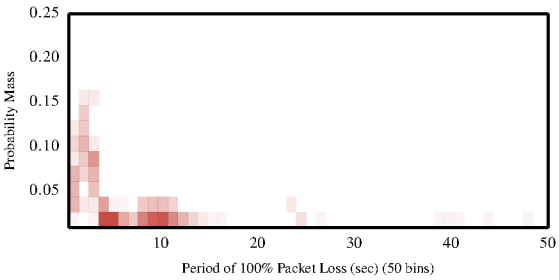
(c) $\tau = 1.0025\delta = 2.005$ sec



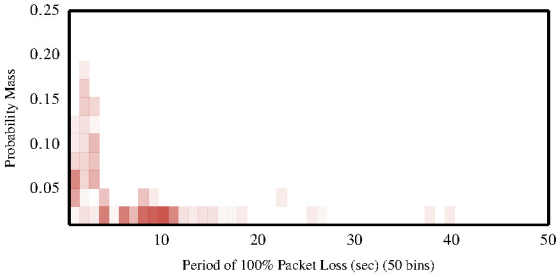
(d) $\tau = 1.005\delta = 2.01$ sec



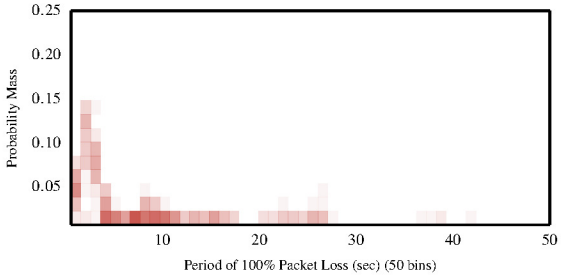
(e) $\tau = 1.025\delta = 2.05$ sec



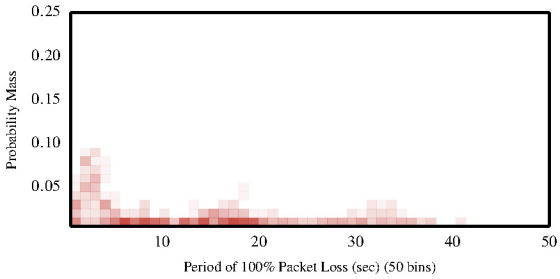
(f) $\tau = 1.05\delta = 2.1$ sec



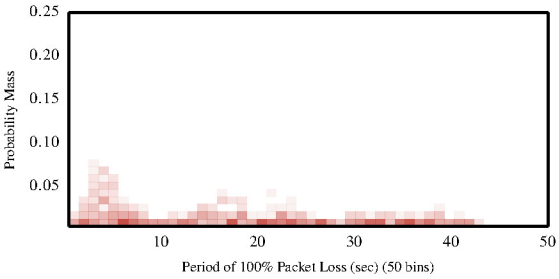
(g) $\tau = 1.1\delta = 2.2$ sec



(h) $\tau = 1.25\delta = 2.5$ sec

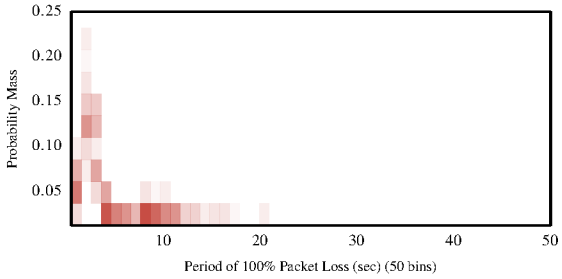


(i) $\tau = 1.5\delta = 3$ sec

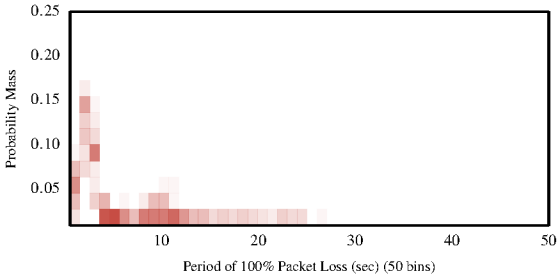


(j) $\tau = 2\delta = 4$ sec

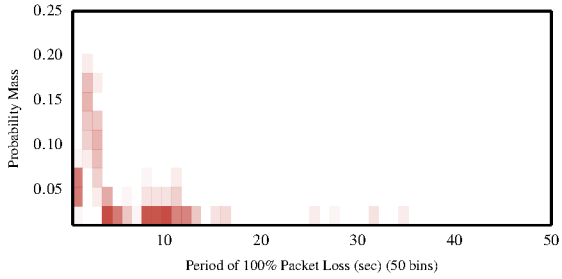
Figure B.2: Probability of various packet loss periods for various values of τ with (δ = 2 sec, replication 2).



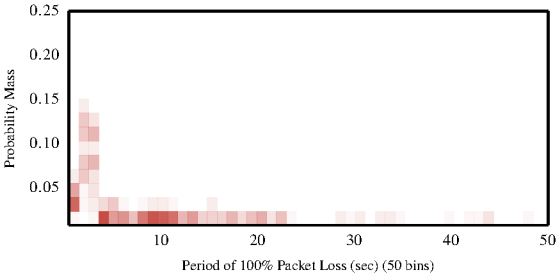
(a) $\tau = 1\delta = 2.0$ sec



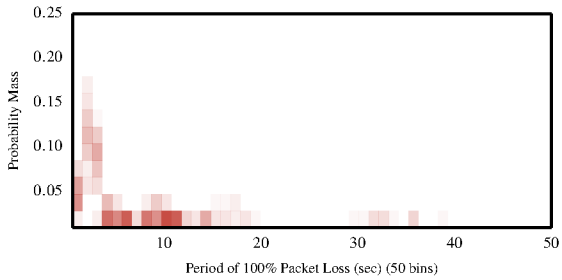
(b) $\tau = 1.0005\delta = 2.001$ sec



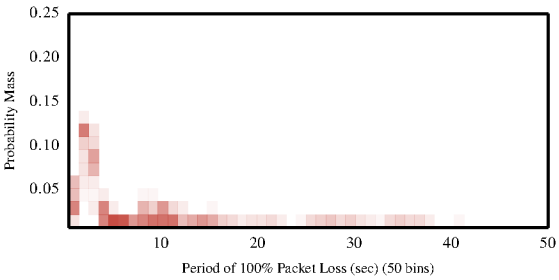
(c) $\tau = 1.0025\delta = 2.005$ sec



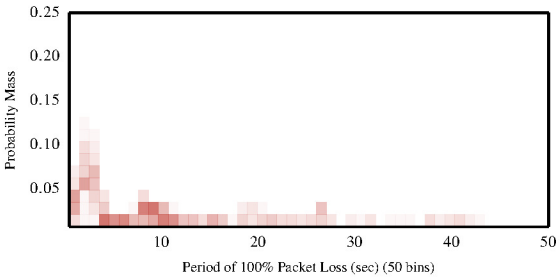
(d) $\tau = 1.005\delta = 2.01$ sec



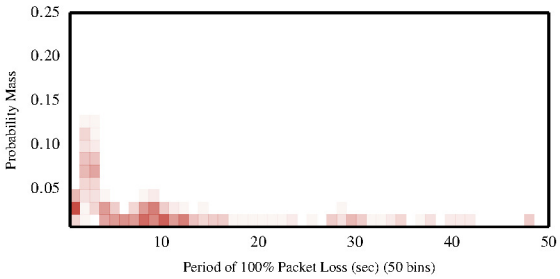
(e) $\tau = 1.025\delta = 2.05$ sec



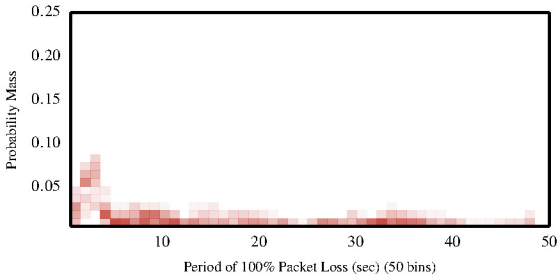
(f) $\tau = 1.05\delta = 2.1$ sec



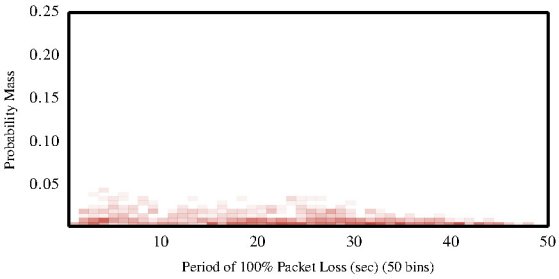
(g) $\tau = 1.1\delta = 2.2$ sec



(h) $\tau = 1.25\delta = 2.5$ sec

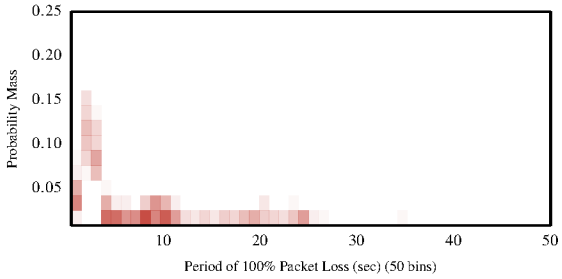


(i) $\tau = 1.5\delta = 3$ sec

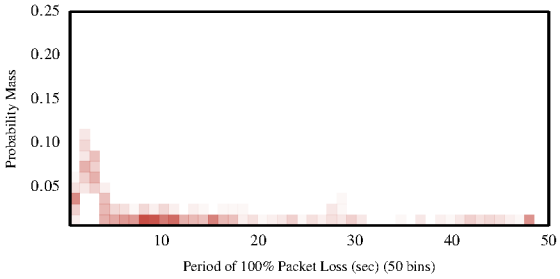


(j) $\tau = 2\delta = 4$ sec

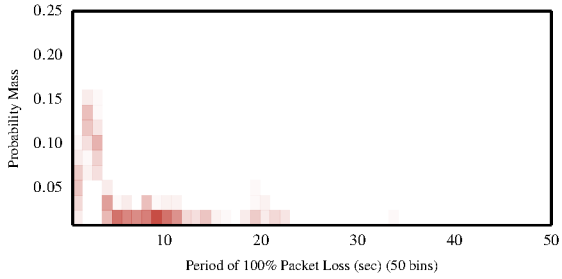
Figure B.3: Probability of various packet loss periods for various values of τ with ($\delta = 2$ sec, replication 3).



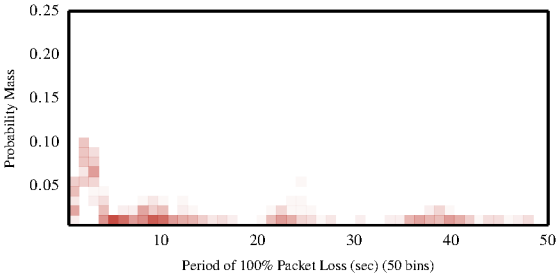
(a) $\tau = 1\delta = 2.0$ sec



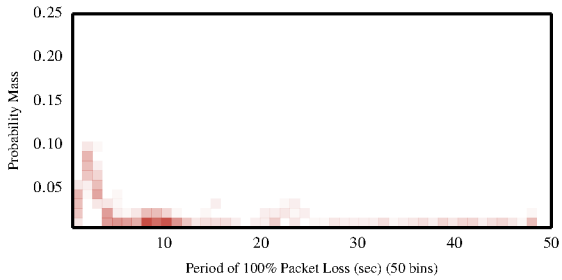
(b) $\tau = 1.0005\delta = 2.001$ sec



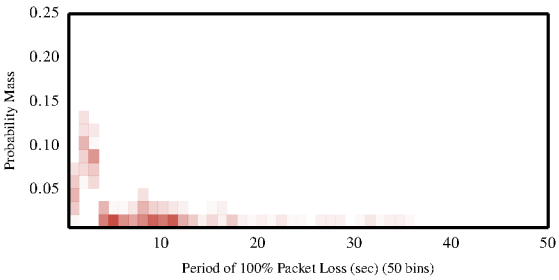
(c) $\tau = 1.0025\delta = 2.005$ sec



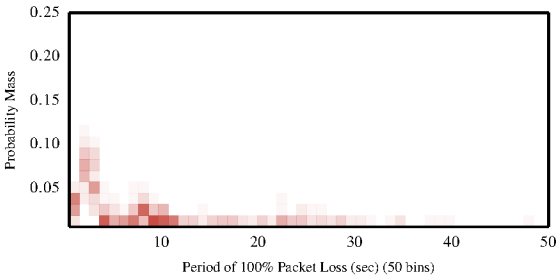
(d) $\tau = 1.005\delta = 2.01$ sec



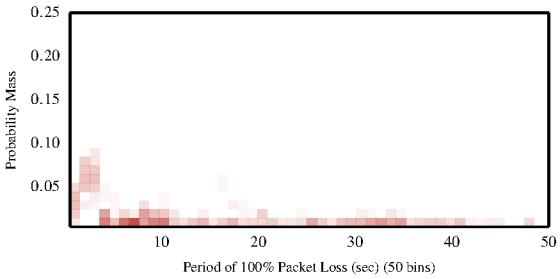
(e) $\tau = 1.025\delta = 2.05$ sec



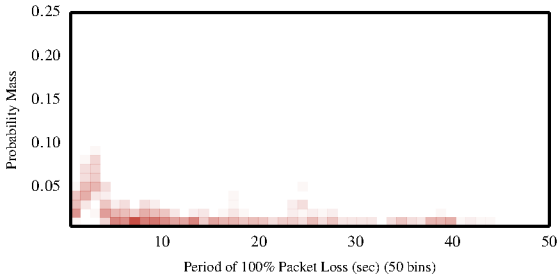
(f) $\tau = 1.05\delta = 2.1$ sec



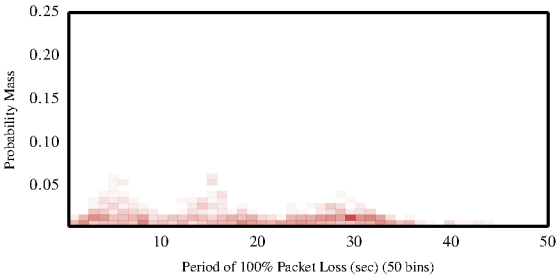
(g) $\tau = 1.1\delta = 2.2$ sec



(h) $\tau = 1.25\delta = 2.5$ sec

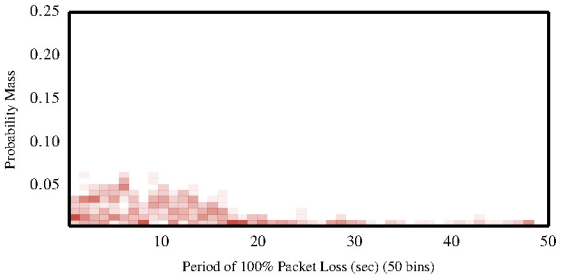


(i) $\tau = 1.5\delta = 3$ sec

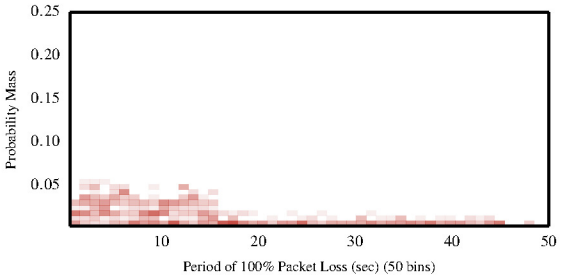


(j) $\tau = 2\delta = 4$ sec

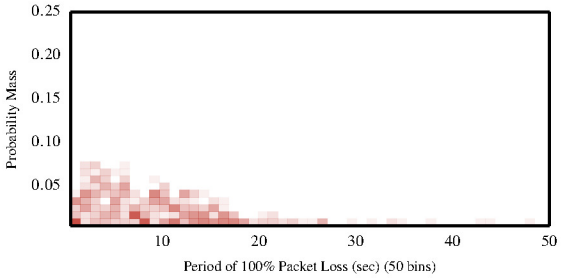
Figure B.4: Probability of various packet loss periods for various values of τ with ($\delta = 2$ sec, replication 4).



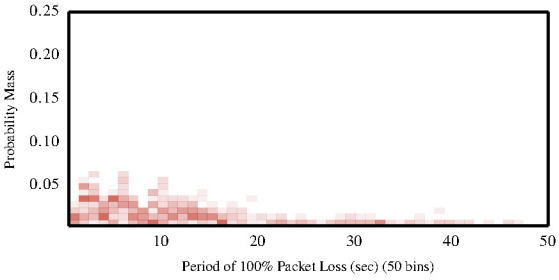
(a) $\tau = 1\delta = 4.0$ sec



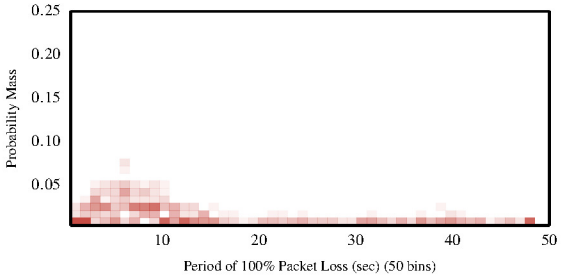
(b) $\tau = 1.1\delta = 4.4$ sec



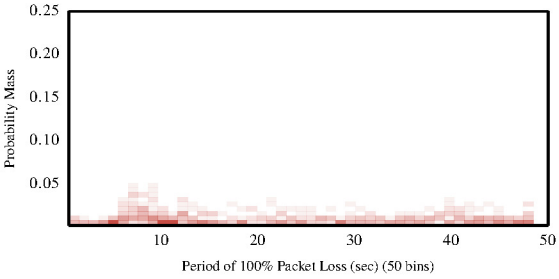
(c) $\tau = 1.125\delta = 4.5$ sec



(d) $\tau = 1.25\delta = 5$ sec

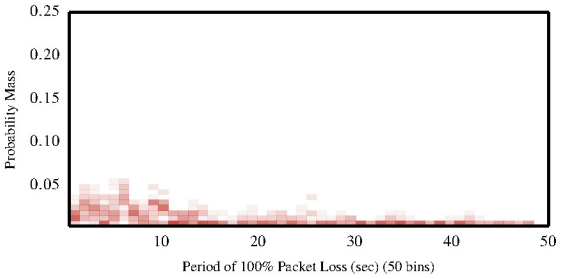


(e) $\tau = 1.5\delta = 6$ sec

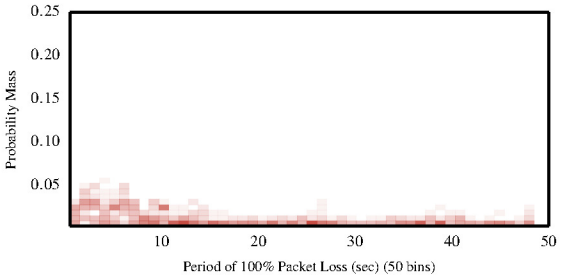


(f) $\tau = 2\delta = 8$ sec

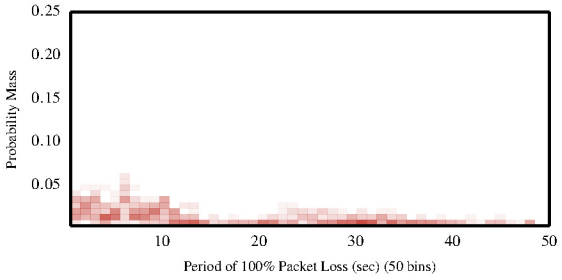
Figure B.5: Probability of various packet loss periods for various values of τ with ($\delta = 4$ sec, replication 1).



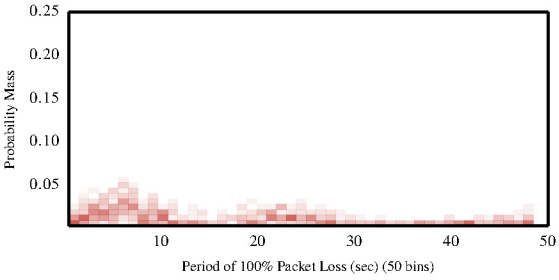
(a) $\tau = 1\delta = 4.0$ sec



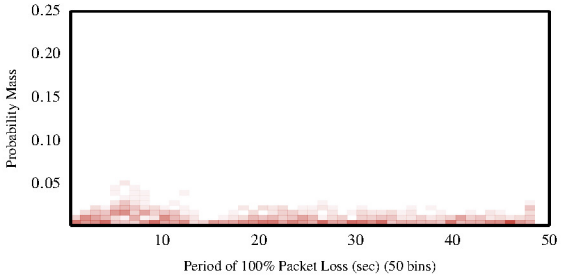
(b) $\tau = 1.1\delta = 4.4$ sec



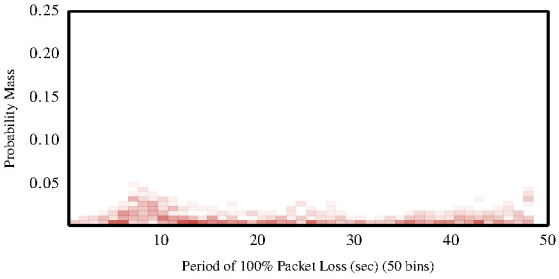
(c) $\tau = 1.125\delta = 4.5$ sec



(d) $\tau = 1.25\delta = 5$ sec

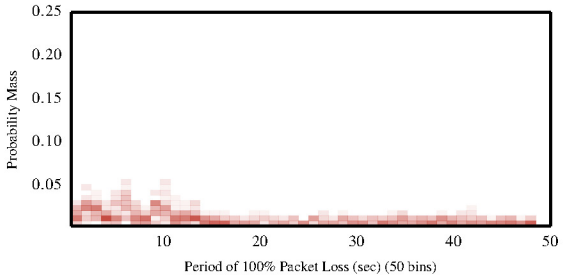


(e) $\tau = 1.5\delta = 6$ sec

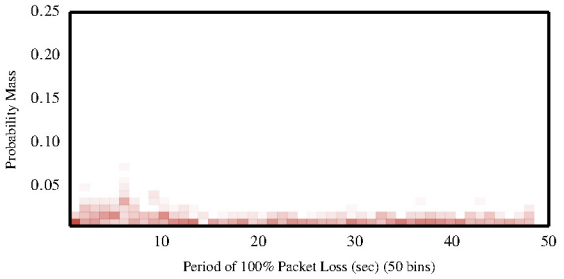


(f) $\tau = 2\delta = 8$ sec

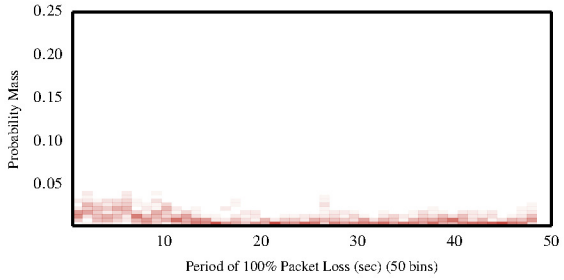
Figure B.6: Probability of various packet loss periods for various values of τ with ($\delta =$ 4 sec, replication 2).



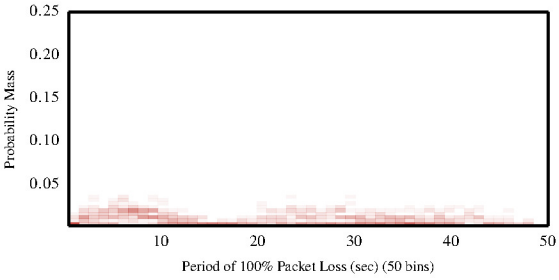
(a) $\tau = 1\delta = 4.0$ sec



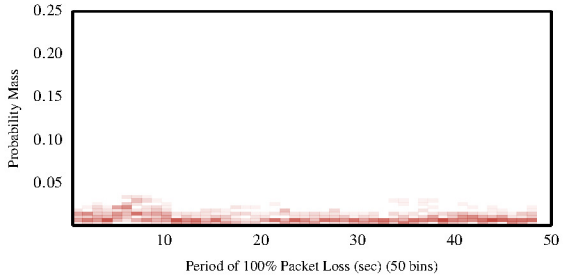
(b) $\tau = 1.1\delta = 4.4$ sec



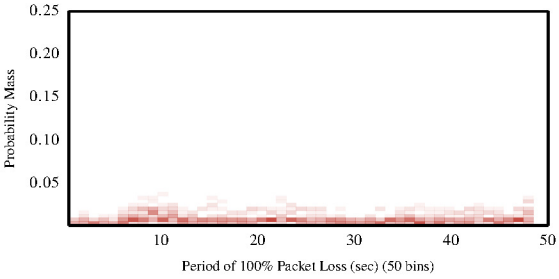
(c) $\tau = 1.125\delta = 4.5$ sec



(d) $\tau = 1.25\delta = 5$ sec

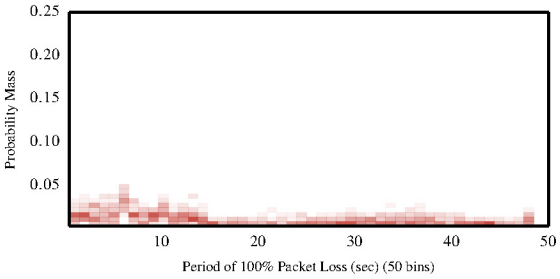


(e) $\tau = 1.5\delta = 6$ sec

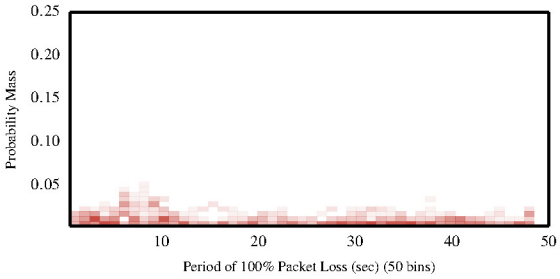


(f) $\tau = 2\delta = 8$ sec

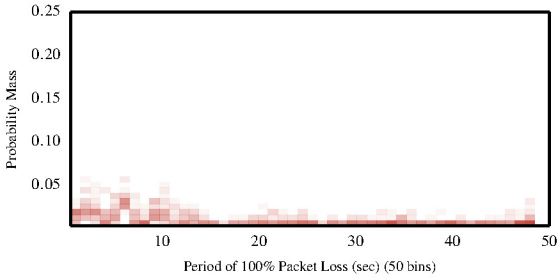
Figure B.7: Probability of various packet loss periods for various values of τ with ($\delta =$ 4 sec, replication 3).



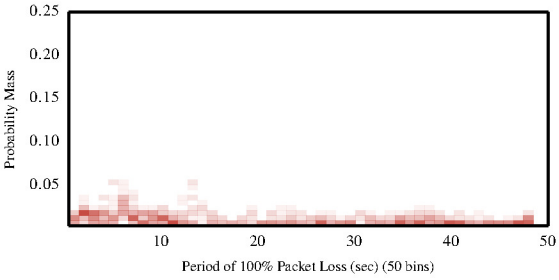
(a) $\tau = 1\delta = 4.0$ sec



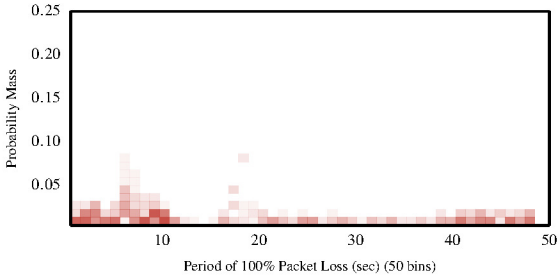
(b) $\tau = 1.1\delta = 4.4$ sec



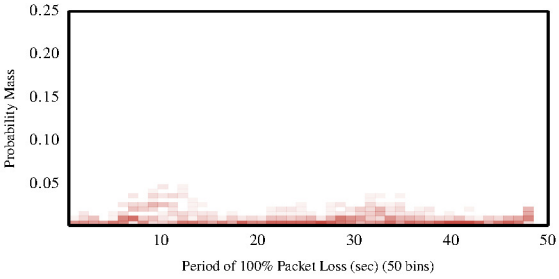
(c) $\tau = 1.125\delta = 4.5$ sec



(d) $\tau = 1.25\delta = 5$ sec

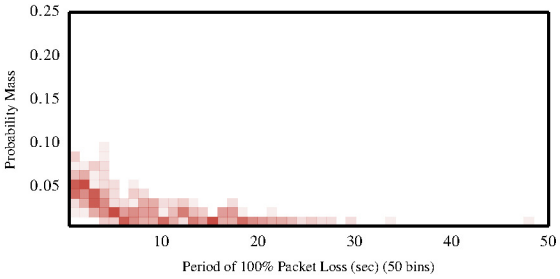


(e) $\tau = 1.5\delta = 6$ sec

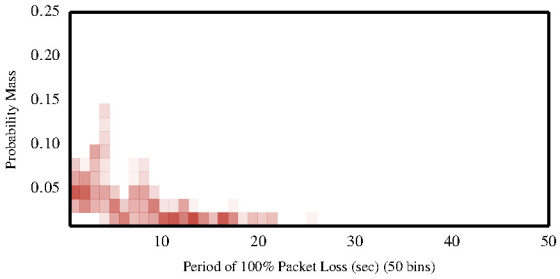


(f) $\tau = 2\delta = 8$ sec

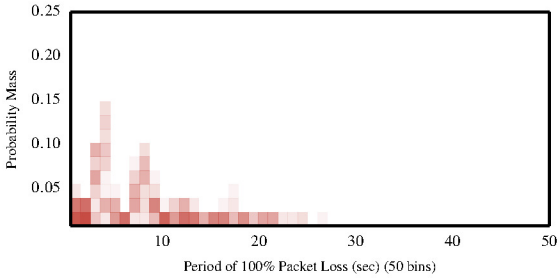
Figure B.8: Probability of various packet loss periods for various values of τ with ($\delta = 4$ sec, replication 4).



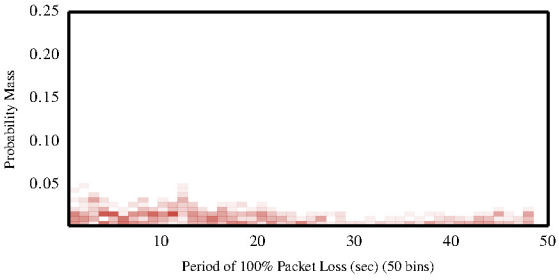
(a) $\tau = 1\delta = 8.0$ sec



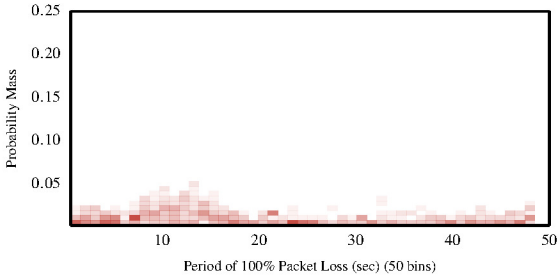
(b) $\tau = 1.1\delta = 8.8$ sec



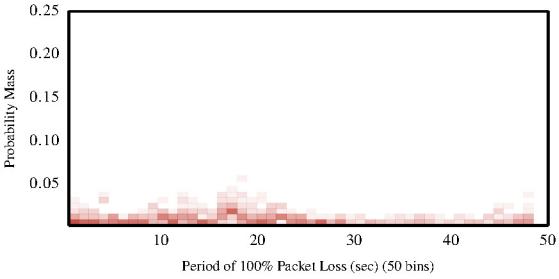
(c) $\tau = 1.125\delta = 9.0$ sec



(d) $\tau = 1.25\delta = 10.0$ sec

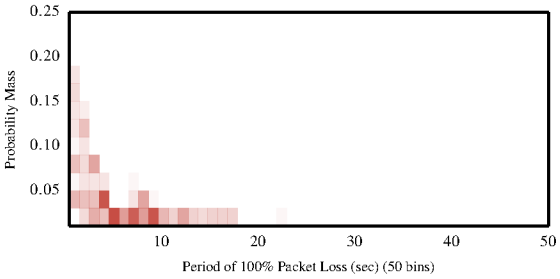


(e) $\tau = 1.5\delta = 12.0$ sec

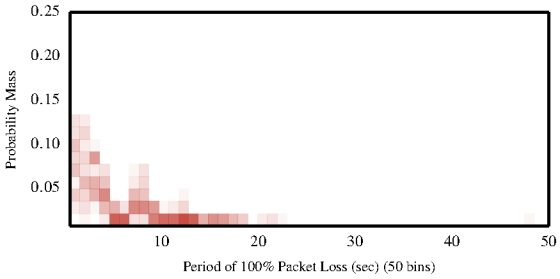


(f) $\tau = 2\delta = 16$ sec

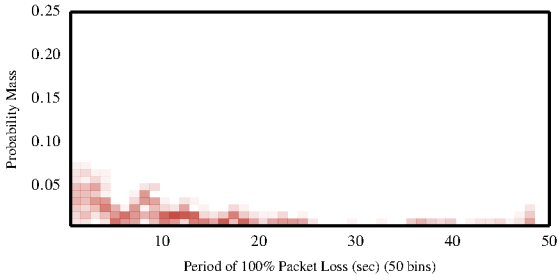
Figure B.9: Probability of various packet loss periods for various values of τ with (δ = 8 sec, replication 1).



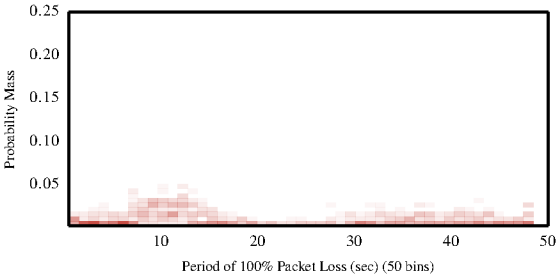
(a) $\tau = 1\delta = 8.0$ sec



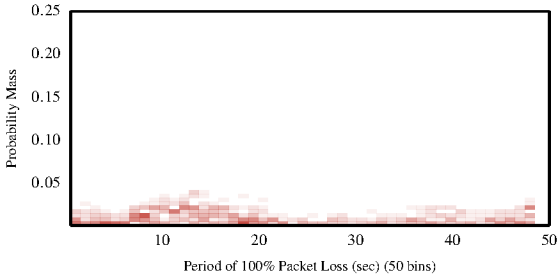
(b) $\tau = 1.1\delta = 8.8$ sec



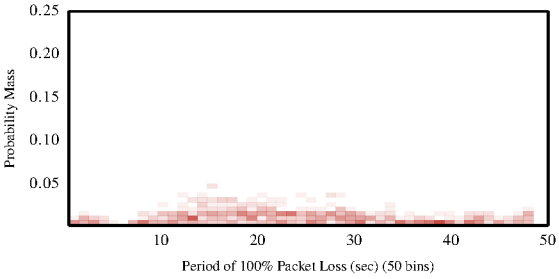
(c) $\tau = 1.125\delta = 9.0$ sec



(d) $\tau = 1.25\delta = 10.0$ sec

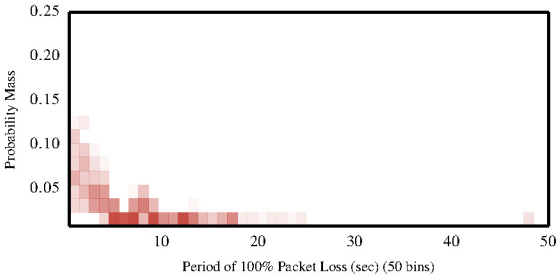


(e) $\tau = 1.5\delta = 12.0$ sec

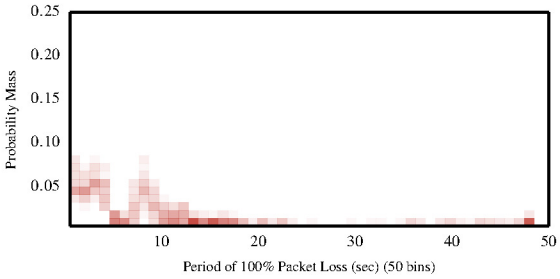


(f) $\tau = 2\delta = 16$ sec

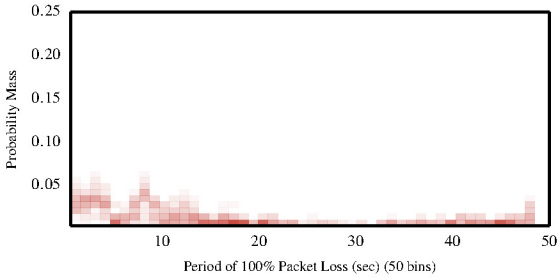
Figure B.10: Probability of various packet loss periods for various values of τ with (δ = 8 sec, replication 2).



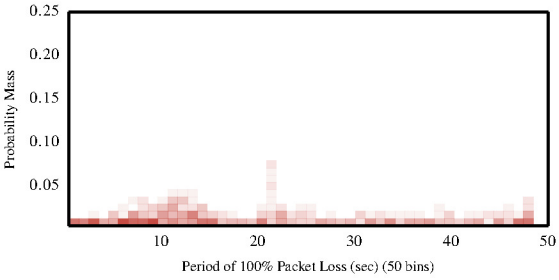
(a) $\tau = 1\delta = 8.0$ sec



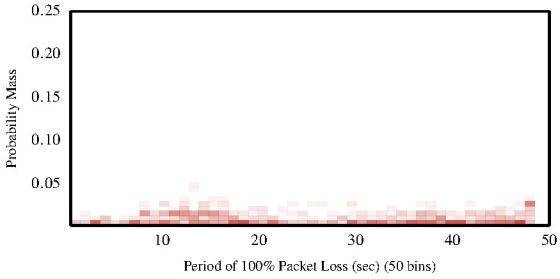
(b) $\tau = 1.1\delta = 8.8$ sec



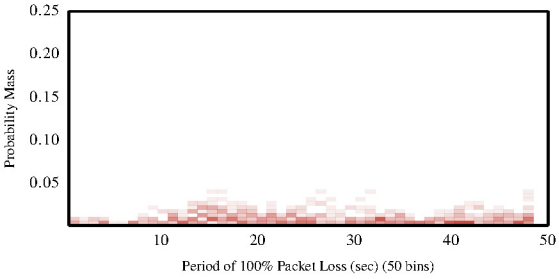
(c) $\tau = 1.125\delta = 9.0$ sec



(d) $\tau = 1.25\delta = 10.0$ sec

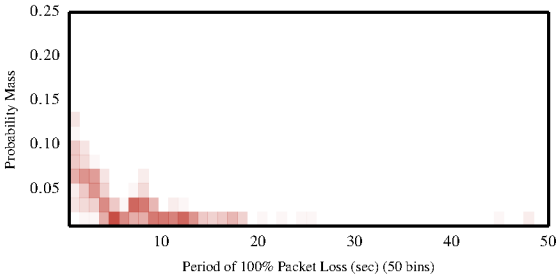


(e) $\tau = 1.5\delta = 12.0$ sec

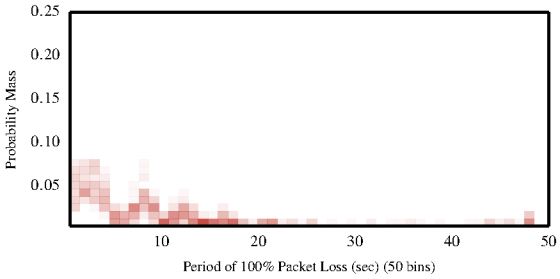


(f) $\tau = 2\delta = 16$ sec

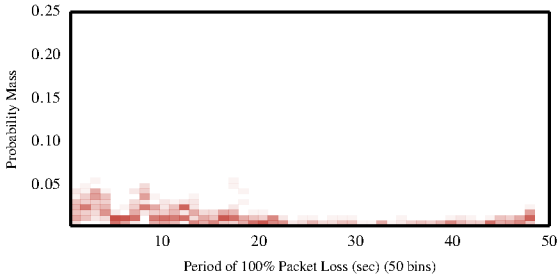
Figure B.11: Probability of various packet loss periods for various values of τ with (δ = 8 sec, replication 3).



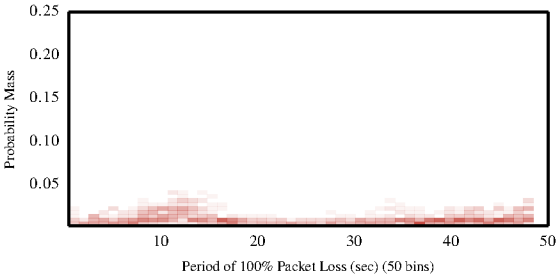
(a) $\tau = 1\delta = 8.0$ sec



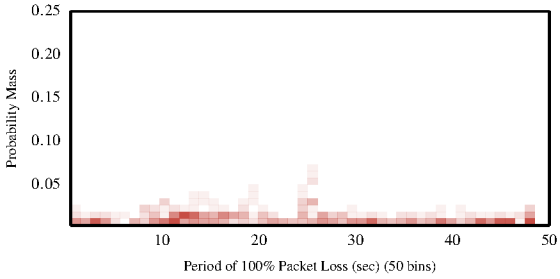
(b) $\tau = 1.1\delta = 8.8$ sec



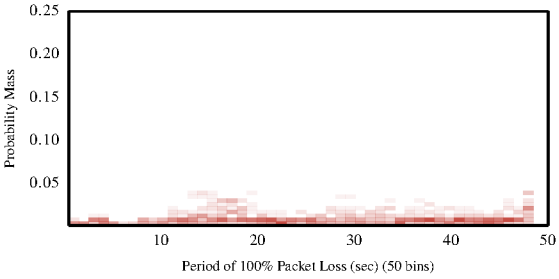
(c) $\tau = 1.125\delta = 9.0$ sec



(d) $\tau = 1.25\delta = 10.0$ sec



(e) $\tau = 1.5\delta = 12.0$ sec



(f) $\tau = 2\delta = 16$ sec

Figure B.12: Probability of various packet loss periods for various values of τ with (δ = 8 sec, replication 4).

Appendix C

Code

C.1 startExperiment.py

```
#!/usr/bin/python
#
# OLSR Experiment Startup Script
#
# Author: Joshua McCartney
# Date: 01/28/2014
#
# Requires paramiko and h5py

import paramiko, argparse, os, platform, time, datetime
from topologies import *
from util import *

args = parseArgs()

scriptDirectory = os.path.dirname(os.path.abspath(__file__))
localRun = False

# The command to execute on the remote machine or possible this
# machine.
command = 'python '+scriptDirectory+'/olsrExperiment.py -c '+
   scriptDirectory+'/'+args.config+' -o '+args.outFile

def startNode(host, command):
    global localRun
    # If we want to run the command locally in the forground, we
        must
    # check to see whether this node's name is the same as the
    # hostname.
    if host == platform.node() and not args.background:
```

```
            localRun = True
        else:
            try:
                client = paramiko.SSHClient()
                client.set_missing_host_key_policy(
                    paramiko.AutoAddPolicy())
                client.load_system_host_keys()
                client.connect(host)

                # Command to launch on host. Note that stdout and
                   stderr must
                # be redirected somewhere otherwise this will block.
                client.exec_command(command+' > /tmp/topologies.out
                   2> /tmp/topologies.err &')
            finally:
                client.close()

start = time.time()

nodes = {"node01": nodes['node01'],
         "node03": nodes['node03']}

for node in nodes:
    print "Starting", node
    startNode(node, command)

print "All nodes are now running"

# If this script was run on one of the machines in the nodes
   dict and
# the user did not specify to run in the background,
   olsrExperiment.py
# will be run in the foreground.
if localRun and not args.background:
    import subprocess
    subprocess.call(command.split())
    sec = datetime.timedelta(seconds=(time.time()-start))
    d = datetime.datetime(1,1,1) + sec

    # startNode('japan.cs.utep.edu', "echo 'Finished in %d days,
       %d hours, %d minutes and %d seconds. command: %s' | mail
       -s 'experiment' limaner2002@gmail.com"%(d.day-1, d.hour,
```

```
        d.minute, d.second, command))

    print "Finished"
```

C.2 olsrExperiment.py

```
#!/usr/bin/python
#
# OLSR Experiment Script
#
# Author: Joshua McCartney
# Date: 01/13/2014
#
#

import os, subprocess, traceback, sys, datetime, select, signal,
    tempfile, h5py
import re
from barrier import *
import util as ut
from topologies import *
import argparse
import Experiment as ex

args = ut.parseArgs()

glob = ut.Globals.Instance()

##############################
#
# Main experiment function (could modify this function to change
    the parameter derivations)
#
##############################
def main():
    #
    # Get the parameter configuration
    #
    cfg = open(args.config, 'r')
    exec(cfg.read())
```

```
cfg.close()

# glob.numTopologyChanges = numChanges
signal.signal(signal.SIGTERM, ut.signal_term_handler)

glob.initialize(args.outFile)

dt = h5py.special_dtype(vlen=str)

# Get the communicator used for synchronization
nodeSub = {"node01": nodes['node01'],
           "node03": nodes['node03']}
glob.comm = Communicator(nodes=nodeDict(nodeSub))
# Needed for importing and running ping later
scriptDirectory = sys.argv[0].rsplit('/', 1)[0]
#
# Run experiments for each value of big delta
#
for Delta in DeltaList:
    for tcMessageInterval in tcMessageIntervalList:
        for helloInterval in helloIntervalList:
            for tcValidity in tcValidityList:
                for helloValidity in helloValidityList:
                    for topologySequenceKey in
                       topologySequenceList:
                        experiment = ex.Experiment(Delta,
                                                   numChanges
                                                      ,
                                                   numRuns,
                                                   timeout,
                                                   pingInterval
                                                      ,
                                                   helloInterval
                                                      ,
                                                   tcMessageInterval
                                                      ,
                                                   helloValidity
                                                      ,
                                                   tcValidity
                                                      ,
                                                   scriptDirectory
                                                      ,
```

```
                                                        topologySequenceKey
                                                           ,
                                                           topologySequenceL
                                                           )
                                    for run in xrange(numRuns):
                                        experiment.run(run)
    # Exit
    return

##############################
#
# exec_olsr_experiment() function
#
# Arguments:
#
# Return value: none
#
##############################
def exec_olsr_experiment(topoChangeInt, helloInterval,
   tcMessageInterval, expRun,
                            helloValidity=20, tcValidity=300,
                               scriptDirectory='', topologySequence
                               =None):

    print "\nLaunching experiment", glob.experimentNumber, "with
        the following OLSR parameters:"
    print "Delta = ", topoChangeInt, "TC_INTERVAL = ",
       tcMessageInterval, "TOP_HOLD_TIME = ", tcValidity
    print "HELLO_MSG_INTERVAL = ", helloInterval, "
       HELLO_HOLD_TIME = ", helloValidity
    sys.stdout.flush()

    ut.startOLSR('%s/olsrd_template.conf'%scriptDirectory,
              '/tmp/olsrd_%03d.conf'%glob.experimentNumber,
              helloInterval,
              tcMessageInterval,
              helloValidity,
              tcValidity,
              scriptDirectory)

    # Wait for a random amount of time from 0-30 seconds
    time.sleep(30+random.randint(0,30))
```

```
    ut.syncNodes(glob.comm)
    print "Synched nodes"
    sys.stdout.flush()

    # Create the instance of the manet class that stores the
       topology
    # sequence and the number of topology changes.
    glob.manet = ut.network(topoChangeInt, maxChanges=glob.
       numTopologyChanges, topologies=topologySequence)

    ut.setupTopology(0)
    dt=numpy.dtype([('time', numpy.float32), ('label', numpy.
       str_, 16)])

    # hdf5 group for storing the time of the instant when a
       topology
    # change occurs.
    changePoints = ut.createDataset('change_%s'%glob.hostname,
                                    parent=expRun,
                                    shape=(glob.manet.maxChanges
                                       +1,),
                                    dtype=dt,
                                    attrs={'label' : glob.hostname,
                                           'numItems' : glob.manet.
                                              maxChanges+1}
                                    )
    changePoints.addPoint((ut.getEpochSeconds(), "topology1"))

#     # Create the threads for pinging
#     packetSenders = startPingSessions(topoChangeInt, expRun)

#     mainLoop(changePoints)

#     for sender in packetSenders:
#          sender.join()
#     mprint("Finished with this configuration")
#     ut.flushFirewall()
#     ut.syncNodes(glob.comm)

#     # Note that the experiment completed without having to be
   restarted
```

```
#     expRun.attrs['Completed'] = True
#     # Exit
#     return

##############################
#
# End of exec_olsr_experiment() function
#
##############################

# This is a subclass of the Thread that is responsible for
   calling
# ping and loggind its output
class PacketSender(threading.Thread):
    def __init__(self, command, path, experiment, chunkSize
       =1000):
        super(PacketSender, self).__init__()
        self.command = command
        self.path = path
        self.chunkSize = chunkSize
        self.experiment = experiment
        pathList = None
        self.pathData = None
        # Regular expressions for parsing ping's output
        self.pingParse = re.compile('\[(?P<sec>.+)\.(?P<usec>.+)
           \] (?P<size>.+) bytes from
           [0-9]+.[0-9]+.[0-9]+.[0-9]+: icmp_req=(?P<
           sequenceNumber>.+) ttl=(?P<ttl>.+) time=(?P<rtt>.+)
           ms')
        self.statParse = re.compile('(?P<transmitted>.+) packets
            transmitted, (?P<received>.+) received, (?P<loss>.+)
           \% packet loss, time (?P<time>.+)ms')

    # Required by the threading module. This is where the thread
    # actually does its work.
    def run(self):
        mutex.acquire()
        try:
            dt=numpy.dtype([('sec', numpy.uint64), ('usec',
               numpy.uint64), ('size', numpy.int32), ('
               sequenceNumber', numpy.int32), ('ttl', numpy.
               int32), ('rtt', numpy.float64)])
```

```
            self.pingData = ut.createDataset("pingData_%s"%(self
                .path), glob.logFile, shape=(self.chunkSize,),
                chunks=(self.chunkSize,), maxshape=(None,), dtype
                =dt)
            sys.stderr.write("pingData.type: %s"%type(self.
                pingData))

            self.experiment.attrs['cmd_%s'%self.path] = self.
                command
            self.pathList = self.experiment['paths']
            self.pathData = ut.createGroup(self.path, parent=
                self.pathList)
        finally:
            mutex.release()

        fstdout = tempfile.TemporaryFile(mode="w+")
        fstderr = tempfile.TemporaryFile(mode="w+")

        self.proc = subprocess.Popen(self.command.split(),
            stdout = fstdout, stderr=fstderr)
        self.proc.wait()
        fstdout.seek(0)
        fstderr.seek(0)
        self.collectOutput(fstdout, fstderr)

        fstdout.close()
        fstderr.close()

    def collectOutput(self, stdout, stderr):
        retcode = self.proc.poll()
        # retcode returns None if the process is still running
        while retcode is None:
            f = select.select([stdout, stderr], [], [])
            mutex.acquire()
            try:
                for fd in f[0]:
                    self.log(fd.readline())
            finally:
                mutex.release()
            retcode = self.proc.poll()

        mutex.acquire()
```

```
    try:
        line = stdout.readline()
        while len(line) > 0:
            self.log(line)
            line = stdout.readline()

        line = stderr.readline()
        while len(line) > 0:
            self.log(line)
            line = stderr.readline()

        self.pathData.attrs['dataRegion'] = self.pingData.
           createRegionReference()
    except:
        traceback.print_exc(file=sys.stderr)
        ut.cleanup()
        sys.exit(1)
    finally:
        mutex.release()

# Used for converting ping's timestamps to the format of
   datetime for
# use later during analysis.
def log(self, msg):
     if msg is None:
          return

     for line in msg.split('\n'):
         # Get the information for a single icmp packet
         if glob.saveMatch(self.pingParse, line) is not None
            :
             g = glob.m.groupdict()
             self.pingData.addPoint((g['sec'], g['usec'], g[
                'size'], g['sequenceNumber'], g['ttl'], g['
                rtt']))
         # Get the statistics for the entire ping session
         elif glob.saveMatch(self.statParse, line) is not
            None:
             g = glob.m.groupdict()
             self.pathData.attrs['transmitted'] = int(g['
                transmitted'])
```

```
                self.pathData.attrs['received'] = int(g['
                    received'])
                self.pathData.attrs['totalTime'] = int(g['time'
                    ])

def startPingSessions(topoChangeInt, expRun):
    packetSenders = []

    # Get information of the system
    processor = platform.processor()
    pingDir = sys.argv[0].rsplit('/', 1)[0]

    for nodeName in nodes.iterkeys():
         if nodeName != glob.hostname:
              pingTimeout = topoChangeInt*(glob.manet.maxChanges
                  +1)
              cmd = "sudo %s/build/%s/ping -D -i 1 -n -W
                  0.038016 -w %d %s"%(pingDir, processor,
                  pingTimeout, nodes[nodeName][1])
              sender = PacketSender(cmd, "%s->%s"%(glob.hostname
                  , nodeName), expRun)
              sender.start()
              packetSenders.append(sender)

    return packetSenders

def mainLoop(changePoints):
    try:
        j=1
        direction=1
        change=1
        # Changes topology sequence up, then back down, then
            back up,
        # etc until maxChanges have occurred
        while change < glob.manet.maxChanges:
            time.sleep(glob.manet.topoChangeInt)
            ut.setupTopology(j)
            # Add the instant when changing to topology j+1 to
                the log file
            changePoints.addPoint((ut.getEpochSeconds(), "
                topology%d"%(j+1)))
            j += direction
```

```
            if j >= len(glob.manet.topologies) or j < 0:
                direction *= -1
                j += 2*direction
            change += 1
    # KeyError occurs when the node is not setup to block any
       packets for the current topology
    except KeyError:
        mprint("Not blocking anything")

>>>>>>> Stashed changes

#
# this script merely calls our main() function
#
try:
     main()
except:                              # Catch *all* exceptions
    traceback.print_exc(file=sys.stderr)
finally:
    ut.cleanup()
```

C.3 topologies.py

```
## Stores the hostname of each node, and its corresponding MAC
   address
 # and IP address
# nodes={"master" : ["f8:1a:67:1b:6f:f2", "172.29.111.242", None
   ],
#        "node01" : ["f8:1a:67:1b:24:dd", "172.29.36.221", None
   ],
#        "node02" : ["f8:1a:67:1b:42:6b", "172.29.66.107", None
   ],
#        "node03" : ["f8:1a:67:0f:a3:b3", "172.29.163.179", None
   ],
#        "node04" : ["f8:1a:67:1b:41:ca", "172.29.65.202", None
   ],
#        "node05" : ["a0:f3:c1:1a:f9:10", "172.29.111.227", None
   ]}

nodes={"master" : ["f8:1a:67:1b:6f:f2", "172.29.111.242", None],
```

```
        "node01" : ["08:00:27:1c:76:66", "192.168.0.1", None],
        "node02" : ["f8:1a:67:1b:42:6b", "172.29.66.107", None],
        "node03" : ["08:00:27:98:b8:dd", "192.168.0.2", None],
        "node04" : ["f8:1a:67:1b:41:ca", "172.29.65.202", None],
        "node05" : ["a0:f3:c1:1a:f9:10", "172.29.111.227", None]}

## The following dictionaries describe how each topology should
   be
 # constructed based on MAC address. The key is the name of the
    current
 # host and the values are the hostnames of the other nodes that
     the
 # current host should block. The corresponding MAC address of
    each
 # node can be found in the "nodes" dictionary
topology1={"master" : ["node02", "node04"],
           "node01" : [],
           "node02" : ["node04", "node05", "master"],
           "node03" : ["node04", "node05"],
           "node04" : ["node02", "node03", "master"],
           "node05" : ["node02", "node03"]}

topology2={"master" : ["node01", "node02", "node04"],
           "node01" : ["master"],
           "node02" : ["node04", "node05", "master"],
           "node03" : ["node04", "node05"],
           "node04" : ["node02", "node03", "master"],
           "node05" : ["node02", "node03"]}

topology3={"master" : ["node01", "node02", "node04"],
           "node01" : ["node03", "node05", "master"],
           "node02" : ["node04", "node05", "master"],
           "node03" : ["node01", "node04", "node05"],
           "node04" : ["node02", "node03", "master"],
           "node05" : ["node01", "node02", "node03"]}

topology4={"master" : ["node01", "node02", "node04"],
           "node01" : ["node03", "node05", "master", "node04"],
           "node02" : ["node04", "node05", "master"],
           "node03" : ["node01", "node04", "node05"],
           "node04" : ["node01", "node02", "node03", "master"],
           "node05" : ["node01", "node02", "node03"]}
```

```
topology5={"master" : ["node01", "node02", "node04"],
           "node01" : ["node03", "node05", "master", "node02"],
           "node02" : ["node01", "node04", "node05", "master"],
           "node03" : ["node01", "node04", "node05"],
           "node04" : ["node02", "node03", "master"],
           "node05" : ["node01", "node02", "node03"]}

topology6={"master" : ["node01", "node02", "node04"],
           "node01" : ["node03", "node05", "master"],
           "node02" : ["node04", "node05", "master", "node03"],
           "node03" : ["node01", "node02", "node04", "node05"],
           "node04" : ["node02", "node03", "master"],
           "node05" : ["node01", "node02", "node03"]}

topology7={"master" : ["node01", "node02", "node04"],
           "node01" : ["node03", "node04", "node05", "master"],
           "node02" : ["node03", "node05", "master"],
           "node03" : ["node01", "node02", "node04", "node05"],
           "node04" : ["node01", "node03", "master"],
           "node05" : ["node01", "node02", "node03"]}

topology8={"master" : ["node01", "node02", "node04"],
           "node01" : ["node03", "node04", "node05", "master"],
           "node02" : ["node03", "node05", "master"],
           "node03" : ["node01", "node02", "node04"],
           "node04" : ["node01", "node03", "master"],
           "node05" : ["node01", "node02"]}

topology9={"master" : ["node01", "node02", "node04", "node05"],
           "node01" : ["node03", "node04", "node05", "master"],
           "node02" : ["node03", "node05", "master"],
           "node03" : ["node01", "node02", "node04"],
           "node04" : ["node01", "node03", "master"],
           "node05" : ["node01", "node02", "master"]}

topology10={"node01" : [],
            "node02" : ['node03', 'node04', 'node05', 'master'],
            "node03" : ['node02', 'node04', 'node05'],
            "node04" : ['node02', 'node03', 'node05', 'master'],
            "node05" : ['node02', 'node03', 'node04', 'master'],
            "master" : ['node02', 'node04', 'node05']}
```

```
topology11={"node01" : ['node02', 'node05'],
            "node02" : ['node01', 'node04', 'node05', 'master'],
            "node03" : [],
            "node04" : ['node02', 'node05'],
            "node05" : ['node01', 'node02', 'node04', 'master'],
            "master" : ['node02', 'node05']}

topology12={"node01" : ['node04', 'master'],
            "node02" : ['node04', 'master'],
            "node03" : ['node04', 'master'],
            "node04" : ['node01', 'node02', 'node03'],
            "node05" : ['master'],
            "master" : ['node01', 'node02', 'node03', 'node05']}

topology13={"node01" : ['node02', 'node05', 'master'],
            "node02" : ['node01', 'node03', 'node05', 'master'],
            "node03" : ['node02', 'node05'],
            "node04" : ['node05'],
            "node05" : ['node01', 'node02', 'node03', 'node04'],
            "master" : ['node01', 'node02']}

topology14={"node01" : ['node05'],
            "node02" : ['node04', 'node05', 'master'],
            "node03" : ['node04'],
            "node04" : ['node02', 'node03', 'node05', 'master'],
            "node05" : ['node01', 'node02', 'node04', 'master'],
            "master" : ['node02', 'node04', 'node05']}

topology15={"node01" : ['node05'],
            "node02" : ['node03'],
            "node03" : ['node02', 'node04', 'node05', 'master'],
            "node04" : ['node03', 'node05', 'master'],
            "node05" : ['node01', 'node03', 'node04', 'master'],
            "master" : ['node03', 'node04', 'node05']}

topology16={"node01" : ['node02', 'node05'],
            "node02" : ['node01', 'node03', 'node04', 'node05'],
            "node03" : ['node02', 'node04', 'node05', 'master'],
            "node04" : ['node02', 'node03', 'node05'],
            "node05" : ['node01', 'node02', 'node03', 'node04'],
            "master" : ['node03']}
```

```
topology17={"node01" : ['node02'],
            "node02" : ['node01', 'node03', 'node04', 'node05'],
            "node03" : ['node02', 'node04', 'master'],
            "node04" : ['node02', 'node03', 'node05'],
            "node05" : ['node02', 'node04', 'master'],
            "master" : ['node03', 'node05']}

topology18={"node01" : ['node03', 'node04', 'node05', 'master'],
            "node02" : [],
            "node03" : ['node01', 'node04', 'node05', 'master'],
            "node04" : ['node01', 'node03', 'node05', 'master'],
            "node05" : ['node01', 'node03', 'node04', 'master'],
            "master" : ['node01', 'node03', 'node04', 'node05']}

topology19={"node01" : ['node02'],
            "node02" : ['node01', 'node03', 'node04', 'master'],
            "node03" : ['node02', 'node04'],
            "node04" : ['node02', 'node03', 'master'],
            "node05" : [],
            "master" : ['node02', 'node04']}

topology20={"node01" : ['node03', 'node05', 'master'],
            "node02" : ['node03', 'node04', 'node05', 'master'],
            "node03" : ['node01', 'node02', 'node04'],
            "node04" : ['node02', 'node03', 'node05'],
            "node05" : ['node01', 'node02', 'node04', 'master'],
            "master" : ['node01', 'node02', 'node05']}

topology21={"node01" : ['node02', 'node03', 'node05', 'master'],
            "node02" : ['node01', 'node03', 'node05', 'master'],
            "node03" : ['node01', 'node02', 'node05'],
            "node04" : ['master'],
            "node05" : ['node01', 'node02', 'node03', 'master'],
            "master" : ['node01', 'node02', 'node04', 'node05']}

topology22={"node01" : ['node02', 'master'],
            "node02" : ['node01', 'node04', 'node05', 'master'],
            "node03" : ['node04'],
            "node04" : ['node02', 'node03', 'node05', 'master'],
            "node05" : ['node02', 'node04', 'master'],
            "master" : ['node01', 'node02', 'node04', 'node05']}
```

```
topology23={"node01" : ['node03', 'node04'],
            "node02" : ['master'],
            "node03" : ['node01', 'node04', 'master'],
            "node04" : ['node01', 'node03', 'node05', 'master'],
            "node05" : ['node04', 'master'],
            "master" : ['node02', 'node03', 'node04', 'node05']}

topology24={"node01" : ['node03', 'node04', 'node05', 'master'],
            "node02" : ['node03', 'node05'],
            "node03" : ['node01', 'node02', 'node05', 'master'],
            "node04" : ['node01', 'node05'],
            "node05" : ['node01', 'node02', 'node03', 'node04'],
            "master" : ['node01', 'node03']}

topology25={"node01" : ['node05', 'master'],
            "node02" : ['node04', 'node05', 'master'],
            "node03" : ['node04', 'master'],
            "node04" : ['node02', 'node03'],
            "node05" : ['node01', 'node02', 'master'],
            "master" : ['node01', 'node02', 'node03', 'node05']}

topology26={"node01" : ['node02', 'node04', 'node05', 'master'],
            "node02" : ['node01', 'node03', 'node04', 'node05'],
            "node03" : ['node02'],
            "node04" : ['node01', 'node02', 'node05', 'master'],
            "node05" : ['node01', 'node02', 'node04', 'master'],
            "master" : ['node01', 'node04', 'node05']}

topology27={"node01" : ['node03', 'node04', 'node05'],
            "node02" : ['node05'],
            "node03" : ['node01', 'master'],
            "node04" : ['node01', 'node05', 'master'],
            "node05" : ['node01', 'node02', 'node04', 'master'],
            "master" : ['node03', 'node04', 'node05']}

topology28={"node01" : ['node02', 'node05'],
            "node02" : ['node01', 'node03', 'node04', 'node05'],
            "node03" : ['node02', 'node04'],
            "node04" : ['node02', 'node03', 'node05', 'master'],
            "node05" : ['node01', 'node02', 'node04'],
            "master" : ['node04']}
```

```
topology29={"node01" : [],
            "node02" : ['node03', 'node04', 'node05', 'master'],
            "node03" : ['node02', 'node04', 'node05', 'master'],
            "node04" : ['node02', 'node03', 'node05', 'master'],
            "node05" : ['node02', 'node03', 'node04', 'master'],
            "master" : ['node02', 'node03', 'node04', 'node05']}

topology30={"node01" : ['node02', 'node03', 'node04', 'node05'],
            "node02" : ['node01', 'node03', 'node04'],
            "node03" : ['node01', 'node02', 'node04', 'node05'],
            "node04" : ['node01', 'node02', 'node03', 'node05'],
            "node05" : ['node01', 'node03', 'node04'],
            "master" : []}

topology31={"node01" : ['node02', 'node03', 'node05', 'master'],
            "node02" : ['node01', 'node05', 'master'],
            "node03" : ['node01', 'node05', 'master'],
            "node04" : [],
            "node05" : ['node01', 'node02', 'node03', 'master'],
            "master" : ['node01', 'node02', 'node03', 'node05']}

topology32={"node01" : ['node02', 'node03', 'node04', 'node05'],
            "node02" : ['node01', 'node03', 'node05'],
            "node03" : ['node01', 'node02', 'node04', 'master'],
            "node04" : ['node01', 'node03', 'node05', 'master'],
            "node05" : ['node01', 'node02', 'node04'],
            "master" : ['node03', 'node04']}

topology33={"node01" : ['master'],
            "node02" : ['node03', 'node04', 'master'],
            "node03" : ['node02', 'node04'],
            "node04" : ['node02', 'node03', 'node05', 'master'],
            "node05" : ['node04', 'master'],
            "master" : ['node01', 'node02', 'node04', 'node05']}

topology34={"node01" : ['node03', 'node04', 'node05', 'master'],
            "node02" : ['node03', 'node05'],
            "node03" : ['node01', 'node02', 'node05'],
            "node04" : ['node01'],
            "node05" : ['node01', 'node02', 'node03', 'master'],
```

```
            "master" : ['node01', 'node05']}

topology35={"node01" : ['node02', 'node03', 'node04', 'master'],
            "node02" : ['node01', 'node03', 'node04', 'master'],
            "node03" : ['node01', 'node02'],
            "node04" : ['node01', 'node02', 'master'],
            "node05" : ['master'],
            "master" : ['node01', 'node02', 'node04', 'node05']}

topology36={"node01" : ['node03', 'node05', 'master'],
            "node02" : [],
            "node03" : ['node01', 'node05', 'master'],
            "node04" : ['node05', 'master'],
            "node05" : ['node01', 'node03', 'node04'],
            "master" : ['node01', 'node03', 'node04']}
```

C.4 util.py

```
#!/usr/bin/python
#
# OLSR Utilities Script
#
# This script contains utility functions such as executing
   commands
# and handling output from the subprocesses. It also includes a
# singleton class for holding all global variables and another
   class
# for handling of HDF5 files.
#
# Author: Joshua McCartney
# Date: 01/13/2014
#
#

import datetime, subprocess, datetime, sys, platform, time, h5py
   , re, numpy, argparse, os.path
from topologies import *

class Singleton:
    """
```

```
A non-thread-safe helper class to ease implementing
   singletons.
This should be used as a decorator -- not a metaclass -- to
   the
class that should be a singleton.

The decorated class can define one `__init__` function that
takes only the `self` argument. Other than that, there are
no restrictions that apply to the decorated class.

To get the singleton instance, use the `Instance` method.
   Trying
to use `__call__` will result in a `TypeError` being raised.

Limitations: The decorated class cannot be inherited from.

"""

def __init__(self, decorated):
    self._decorated = decorated

def Instance(self, *args):
    """
    Returns the singleton instance. Upon its first call, it
       creates a
    new instance of the decorated class and calls its `
       __init__` method.
    On all subsequent calls, the already created instance is
        returned.

    """
    try:
        return self._instance
    except AttributeError:
        self._instance = self._decorated(*args)
        return self._instance

def __call__(self):
    raise TypeError('Singletons must be accessed through `
       Instance()`.')

def __instancecheck__(self, inst):
```

```
        return isinstance(inst, self._decorated)

# This class should be renamed to experimentSettings or
   something
# better. Maybe Globals?
@Singleton
class Globals:
     def __init__(self):
         self.logFile = None
         self.logFileName = None
         self.epoch = datetime.datetime.utcfromtimestamp(0)
         self.experimentNumber = 0
         self.hostname = platform.node()
         self.comm = None
         self.manet = None
         self.experiments = None
         self.numTopologyChanges = 20
         self.run = 0

     def initialize(self, logFileName):
         self.fileName = logFileName
         if os.path.isfile(logFileName):
             self.logFile = h5py.File(logFileName, mode='r+')
         else:
             self.logFile = h5py.File(logFileName, mode='w-')
         self.experiments = createGroup('experiments', parent=
            self.logFile)

     def saveMatch(self, expression, string):
         self.m = re.match(expression, string)
         return self.m

class Dataset:
    def __init__(self, h5Group):
        self.h5Group = h5Group
        if self.h5Group.chunks is not None:
            self.chunkSize = self.h5Group.chunks
        else:
            self.chunkSize = self.h5Group.shape

        if 'numPoints' in self.h5Group.attrs:
            self.regionStart = self.h5Group.attrs['numPoints']
```

```
        else:
            self.regionStart = 0

        self.chunkStart = self.regionStart
        self.attrs = self.h5Group.attrs
        self.index = self.regionStart

    def addPoint(self, point):
        while self.index >= self.h5Group.shape[0]:
            if len(self.h5Group.shape) > 1:
                self.h5Group.resize((self.index+self.chunkSize
                   [0], self.h5Group.shape[1]))
            else:
                self.h5Group.resize((self.index+self.chunkSize
                   [0],))
        self.h5Group[self.index] = point
        self.index += 1

    def createRegionReference(self):
        self.h5Group.attrs['numPoints'] = self.index
        sys.stderr.write("start: %d, end: %d\n"%(self.
           regionStart, self.index))
        return self.h5Group.regionref[self.regionStart:self.
           index]

# Responsible for parsing the command line arguments
def parseArgs():
    parser = argparse.ArgumentParser(description='Run a set of
       experiments with the given parameter configuration',
                                     formatter_class=argparse.
                                        RawDescriptionHelpFormatter
                                        )
    parser.epilog = '''\
Example:
        %s -c example.conf -o exampleOutput.hdf5
'''%parser.prog


    parser.add_argument('-c', '--config', required=True,
                        help='The name of the config file to use
                           ')
```

```
    parser.add_argument('-o', '--output-file', dest='outFile',
       required=True,
                        help='The name of the file that the
                           results will be logged to')
    parser.add_argument('-b', '--background', action='store_true
       ',
                        help='Run the local process in the
                           background')
    return parser.parse_args()

# Stores the topological change interval, the maximum number of
# topological changes and the list of topology dictionaries (
   defined
# above) to cycle through.
class network:
    def __init__(self, topoChangeInt, maxChanges=None,
       topologies=[topology1, topology2, topology3, topology4,
       topology5, topology6, topology7, topology8, topology9]):
        self.topoChangeInt = topoChangeInt
        self.topologies = topologies
        if maxChanges is None:
            self.maxChanges = len(topologies)
        else:
            self.maxChanges = maxChanges

def getBetween(string, first, second):
    start = string.find(first)+len(first)
    end = string.find(second, start)
    if start == -1:
        return None

    return string[start:end]

glob = Globals.Instance()
debug = False

# Handle for termination signal
def signal_term_handler(signal, frame):
    print 'got SIGTERM'
    cleanup()
    sys.exit(0)
```

```
def getEpochSeconds():
    time = datetime.datetime.now()
    delta = time-glob.epoch
    return delta.total_seconds()

# Runs the command specified in the parameter cmd using the
   subprocess
# module
def runCommand(cmd):
    if debug:
        subprocess.call(cmd)
    else:
        proc = subprocess.Popen(cmd, stdout = subprocess.PIPE,
           stderr = subprocess.PIPE)
        return proc.communicate()
    return (None, None)

# Not sure if still used
def mRead(stream):
    if stream is not None:
        if isinstance(stream, basestring):
            mprint(stream)
        else:
            mprint(stream.read())

# Writes msg to the logfile specified during invocation
def mprint(msg, timestamp=True, dataSet=None):
    if len(msg) > 0:
        print("%s: %s"%(datetime.datetime.now(), msg))
        sys.stdout.flush()

# Flushes all iptables rules
def flushFirewall():
    if debug:
        mprint("Before flushing")
        showFirewall()

    flush = "sudo iptables -t raw -F"
    runCommand(flush.split())

# Displays iptables rules for the raw table. Used only for
   debugging
```

```
def showFirewall():
    cmd = "sudo iptables -t raw -L"
    runCommand(cmd.split())

## Sets up the iptables rules according to the topology
   parameter.
 #
 # @param topology contains the list of MAC addresses that
    should be blocked
 #
 # @param hostname the name of the current host. This is used as
     a
 # dictionary key to locate the MAC addresses that this host
    should
 # block.
def setupFirewall(topology):
    for block in topology[glob.hostname]:
        cmd = "sudo iptables -t raw -A PREROUTING -m mac --mac-
           source %s -j DROP"%(nodes[block][0])
        (stdout, stderr) = runCommand(cmd.split())
        mRead(stdout)
        mRead(stderr)
    if debug:
        mprint("After setup")
        showFirewall()

## Synchronized the clocks on all of the nodes
def syncNodes(comm):
     mprint("Barrier")
     comm.barrier()

     if glob.hostname == "master":
          cmd = "sudo ntpdate japan.cs.utep.edu"
     else:
          currentDirectory = sys.argv[0].rsplit('/', 1)[0]
          cmd = "bash %s/syncClocks.sh"%currentDirectory
     (stdout, stderr) = runCommand(cmd.split())
     mRead(stdout)
     mRead(stderr)

## Starts olsr with the desired configuration
 #
```

```
 # @param templateFile This is path to the base configuration
 # file. The file should contain everything except for the
    message
 # intervals
 #
 # @param configFile The path to the configuration file that
    olsrd
 # will use. This function will create that file using the
    basefile
 # and any parameters that follow
 #
def startOLSR(templateFile, configFile, helloInterval,
   tcMessageInterval, helloValidity=20, tcValidity=300,
   scriptDirectory=''):
    template = open(templateFile, 'r')
    configPointer = open(configFile, 'w')
    configPointer.write(template.read()%(helloInterval,
       helloValidity, tcMessageInterval, tcValidity))
    configPointer.close()
    template.close()

    cmd = 'sudo killall olsrd'
    (stdout, stderr) = runCommand(cmd.split())
    mRead(stdout)
    mRead(stderr)


    cmd = 'bash %s/startOLSR.sh %s'%(scriptDirectory, configFile
       )
    (stdout, stderr) = runCommand(cmd.split())
    mRead(stdout)
    mRead(stderr)

## Used to show the iptables rules for the raw table. Only used
   for
## debugging
def showFirewall():
    cmd = "sudo iptables -t raw -L"
    runCommand(cmd.split())

## Used to disable the ethernet device. Not currently used
def disableEthernet():
```

```
    cmd = 'sudo ifconfig eth0 down'
    runCommand(cmd.split())

## Used to re-enable the ethernet device. Not currently used
def enableEthernet():
    cmd = 'sudo ifconfig eth0 up'
    runCommand(cmd.split())
    cmd = 'sudo route add default gw 129.108.4.1'
    runCommand(cmd.split())
    cmd = 'sudo route del -net 129.108.0.0 netmask 255.255.0.0
       gw 0.0.0.0'
    runCommand(cmd.split())

# This should be run whenever this script exits. It flushes the
# firewall, closes the log file, and closes all TCP connections
   used
# in the barrier
def cleanup():

    mprint("Finished, clearing firewall")
    flushFirewall()
    if glob.comm is not None:
        glob.comm.closeConnections()
    time.sleep(3)

def setupTopology(topologyNumber):
    flushFirewall()
    setupFirewall(glob.manet.topologies[topologyNumber])

def addPoint(dataSet, point):
    if 'numPoints' in dataSet.attrs:
        index = dataSet.attrs['numPoints']
    else:
        index = 0

    if isinstance(point, numpy.ndarray):
        offset = len(point)
    else:
        offset = 0

    while index >= dataSet.shape[0] - offset:
        chunkSize = dataSet.chunks[0]
```

```
            if len(dataSet.shape) > 1:
                dataSet.resize((index+chunkSize, dataSet.shape[1]))
            else:
                dataSet.resize((index+chunkSize,))

        if isinstance(point, numpy.ndarray):
            for i in xrange(len(point)):
                dataSet[index+i] = point[i]
            dataSet.attrs['numPoints'] = index+len(point)
        else:
            dataSet[index] = point
            dataSet.attrs['numPoints'] = index+1

def createDataset(name, parent, attrs=[], **kwargs):
    return createGroup(name, parent, attrs, dataset=True, **
       kwargs)

def createGroup(name, parent, attrs=[], dataset=False, **kwargs)
   :
    if name in parent:
        group = parent[name]
    else:
        if dataset:
            group = parent.create_dataset(name=name, **kwargs)
        else:
            group = parent.create_group(name)

        for attr in attrs:
            group.attrs[attr] = attrs[attr]

    if dataset:
        return Dataset(group)
    else:
        return group
```

C.5 barrier.py

```
import socket, platform, copy, itertools, time, datetime, sys,
   traceback, select
```

```
sleepTime = 1
# timeout = 5
#logFile = sys.stdout
logFile = open('/tmp/barrier', 'w')
def mprint(msg):
    logFile.write('%s: %s\n'%(datetime.datetime.now(), msg))
    logFile.flush()

def eprint(msg):
    sys.stderr.write('%s: %s\n'%(datetime.datetime.now(), msg))

class nodeDict(dict):
    def __init__(self, *args, **kwargs):
        super(nodeDict, self).__init__(*args, **kwargs)

    def findKey(self, val):
        for k, v in self.iteritems():
            if v[1] == val:
                return k
        mprint(val)
        return None

        # This class is responsible for creating a TCP
           connection from
        # every node to the root node for synchronization
           purposes.
class Communicator():
    def __init__(self, nodes=nodeDict(), myName=platform.node())
       :
        self.nodes = nodes
        self.myName = myName
        self.port = 57836
        self.barrierNum = 0
        self.sockets = []
        self.recvSize = 0
        self.createConnections()
        mprint(self.nodes)

        # Returns a TCP socket that connects this machine to the
        # machine in the variable 'node'
    def createSocket(self, node):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        connected = False
        while not connected:
            try:
                s.connect((node, self.port))
                connected = True
            except socket.timeout:
                eprint("Socket.timeout")
                pass
            except socket.error as e:
                if e.errno == 113 or e.errno == 111 or e.errno
                   == 103 or e.errno == 110:# or e.errno ==
                   socket.errno.EPIPE:
                    eprint("%s"%str(e.args))
                    time.sleep(sleepTime)
                else:
                    raise
        return s

        # Creates a TCP socket that listens for connections.
           Only for
        # the root node.
    def listen(self, numNodes=5):
        mprint('Listening')
        self.socket = socket.socket(socket.AF_INET, socket.
           SOCK_STREAM)
        self.socket.bind(('', self.port))
        self.socket.listen(numNodes)

        # Sets up all connections from the root node and every
           other
        # node
    def createConnections(self):
        root = self.nodes.iterkeys().next()
        if self.myName == root:
            nodesTemp = self.nodes.keys()
            nodesTemp.remove(self.myName)
            listening = False
            while listening == False:
                try:
                    self.listen()
                    listening = True
```

```
                except socket.error as e:
                    if e.errno == socket.errno.EADDRINUSE:
                        eprint("%s"%str(e.args))
                        time.sleep(10)
                    else:
                        raise
            # Accept connections from all other nodes
            while nodesTemp:
                connection, addr = self.socket.accept()
                nodeName = self.nodes.findKey(addr[0])
                if nodeName in nodesTemp:
                    if self.recvSize == 0:
                        self.recvSize = connection.getsockopt(
                           socket.SOL_SOCKET, socket.SO_RCVBUF)
                    self.sockets.append(connection)
                    nodesTemp.remove(nodeName)
                    mprint("Accepted connection from %s"%str(
                       addr))
                else:
                    mprint("Already connected to %s"%str(addr))
        else:
            time.sleep(sleepTime)
            self.nodes[root][2] = self.createSocket(root)
            self.recvSize = self.nodes[root][2].getsockopt(
               socket.SOL_SOCKET, socket.SO_RCVBUF)
            # self.sockets.append(self.createSocket(root))
            mprint("Connected to %s"%root)

    def barrier(self):
        root = self.nodes.iterkeys().next()
        synched = False
        index = -1
        if self.myName == root:
            # Create a list of nodes that the root needs to
            # synchronize with
            nodesTemp = self.nodes.keys()
            # Remove myself from the list
            nodesTemp.remove(self.myName)
            # Listen for connections until all nodes have been
               heard
            # from
            while nodesTemp:
```

```
            try:
                # Wait until one of the sockets has data
                   ready for
                # reading
                f = select.select(self.sockets, [], [])
                for soc in f[0]:
                    bufsize = soc.getsockopt(socket.
                       SOL_SOCKET, socket.SO_RCVBUF)
                    data = soc.recv(self.recvSize)
                    data = filter(lambda x: len(x) > 0, data
                       .split(":"))
                    mprint("Received %s"%data)
                    for datum in data:
                        datum = datum.split(',')
                        node = datum[0]
                        if node in nodesTemp and int(datum
                           [1]) == self.barrierNum:
                            nodesTemp.remove(node)
            except socket.timeout:
                eprint("Timeout occurred!")
                time.sleep(sleepTime)
                pass
            except socket.error as e:
                if e.errno == 110:
                    eprint("Error: %s"%str(e.args))
                    socketIndex = self.sockets.index(soc)
                    soc.close()
                    connection, addr = self.socket.accept()
                    self.sockets[socketIndex] = connection
                    mprint("Accepted connection from %s"%str
                       (addr))
                else:
                    raise
        # Broadcast to all nodes that every node has reached
            the
        # barrier
        for soc in self.sockets:
            soc.send("True,%d:"%self.barrierNum)
    # Regular node
    else:
        while not synched:
            try:
```

```
                    self.nodes[root][2].send("%s,%s:"%(self.
                        myName, self.barrierNum))
                    data = self.nodes[root][2].recv(self.
                        recvSize)
                    data = filter(lambda x: len(x) > 0, data.
                        split(":"))
                    mprint("Received %s"%str(data))
                    for datum in data:
                        datum = datum.split(',')
                        if bool(datum[0]) and int(datum[1]) ==
                            self.barrierNum:
                            synched = bool(data)
                            break

                except socket.error as e:
                    if e.errno == 113 or e.errno == 111 or e.
                        errno == 110 or e.errno == socket.errno.
                        ECONNRESET:
                        eprint("%s"%str(e.args))
                        time.sleep(sleepTime)
                        pass
                    elif e.errno == socket.errno.EPIPE:
                        eprint("Broken pipe, recreating
                            connection")
                        self.nodes[root][2].close()
                        self.nodes[root][2] = self.createSocket(
                            root)
                        pass
                    else:
                        raise

        self.barrierNum += 1

    def closeConnections(self):
        for soc in self.sockets:
            try:
                soc.shutdown(socket.SHUT_RD)
                soc.close()
            except socket.error as e:
                        if e.errno == 107:
                            eprint("Socket for %s was already
                                closed!\n\n"%node)
```

```
                            traceback.print_exc(file=sys.stderr)
                            eprint("Continuing\n")
                            pass
                        else:
                            raise

# Used for testing purposes
if __name__ == "__main__":
    try:
        import random
        nodes=nodeDict()

        for line in sys.argv[1][:-1].split(';'):
            line = line.strip().split(',')
            nodes[line[0]] = line[1:]
        print nodes

        comm = Communicator(nodes=nodes)
        root = nodes.iterkeys().next()
        for i in xrange(0,30):
            if platform.node() != root:
                sleepInterval = random.randint(1,5)
            else:
                sleepInterval = random.randint(1,60)
            mprint("Sleeping for %ds"%sleepInterval)
            time.sleep(sleepInterval)
            mprint("Barrier %d"%i)
            comm.barrier()
            mprint("Finished barrier %d."%i)
        mprint("Closing connections")
        comm.closeConnections()
    except:
        print "%s encountered an error"%platform.node()
        traceback.print_exc(file=sys.stderr)
```

C.6 Example Parameter Configuration File

```
## OLSR Parameters
# An experiment set is defined by the smallest topology change
# interval (in seconds)
```

```
delta= 120
# The next three parameters must be a list
DeltaList= [delta]
tcMessageIntervalList= [5]
helloIntervalList= [8]
tcValidityList = [300]
helloValidityList= [x*y for x in helloIntervalList for y in
   [1.125, 1.25, 1.5, 2, 1.0, 1.1, 4]]
# The number of runs per experiment.
numRuns = 1
numChanges = 160

# ping parameters
# Set the interval (in seconds) of each ping
pingInterval= 1
# Time to wait (in seconds) for a response
timeout= 0.038016

#topology parameters
# The sequence of topology changes to use for each of the four
   runs.
topologySequenceList = [[topology1, topology2, topology3,
   topology4, topology5, topology6, topology7, topology8,
   topology9]]
```

C.7 sync.sh

```
#!/bin/bash

#MASTER=129.108.4.100
#MASTER=129.108.4.99
MASTER=10.0.1.27
USER=josh
WORKSPACE_DIR=$HOME/workspace
SCRIPTS_DIR=$WORKSPACE_DIR/Scripts/OLSRScripts/*
HOME=/home/josh

rsync --exclude '*~' --delete-before --copy-links -avh --rsync-
   path="mkdir -p $HOME/workspace/ && rsync" $SCRIPTS_DIR
   $USER@$MASTER:$HOME/workspace/scripts
```

```
# ssh $MASTER "sudo ntpdate 129.108.4.88"
ssh $USER@$MASTER "bash $HOME/workspace/scripts/syncNodes.sh"
```

C.8 syncNodes.sh

```
#!/bin/bash

WORKSPACE_DIR=$HOME/workspace
SCRIPTS_DIR=$WORKSPACE_DIR/scripts

for i in {3..3}
do
    echo "Synching to node0$i"
    rsync --exclude '*~' --delete-before -avh $SCRIPTS_DIR
       node0$i:"$WORKSPACE_DIR"
    # ssh node0$i "bash $SCRIPTS_DIR/syncClocks.sh"
done
```

Chapter 7

Curriculum Vita

I received my Bachelor's in Computer Science at the University of Texas at El Paso in 2011. During this time, I worked in the Interactive Systems Group under Dr. Nigel Ward and Dr. David Novick from Fall 2008 to May 2010. I created a tool that helped significantly reduce the time to develop a rule for back-channeling in human-computer interaction. This will potentially allow someone create a rule for back-channels with any language without requiring an understanding of the language. This was my first introduction to research and I was able to publish a paper on the work I did there.

In May 2010, I switched to the HiPerSys research lab under Dr. Patricia Teller and Dr. Sarala Arunagiri. I started my first project where I compared execution time and power consumption performance of CUDA and PGI compiled codes that run on General Purpose Graphical Processing Units (GPGPUs). The PGI compilers aim to reduce the amount of work required to develop efficient code that runs on GPGPUs. Next I carried out a study on the effect of node power capping on two applications of interest to the U.S. Army for fielded applications.

In summer of 2010, I attended the Army Higher Performance Computing Research Center (AHPCRC) Summer Institute at Stanford University in the Department of Materials Engineering. Here I extended modeling dislocations to semiconductor materials, such as silicon, using dislocation dynamics. This is the first step to modeling many dislocations on a large scale to help study the deterioration of semiconductors during use.

In the summer of 2011, I interned at the U.S. Army Research Laboratory (ARL) in the Computational and Information Sciences Division at Aberdeen Proving Ground, Maryland. Here I implemented spatial data structures, namely quad-tree, k-d tree, and Binary Space Partition (BSP) tree, and evaluated them in terms of their respective execution performance for real-time ray-tracing on GPGPUs. This ray-tracing is of interest to the army because it can be used for real-time threat analysis of an area of interest and can help determine a path that minimizes exposure to potential ballistic threats.

In spring of 2012 I started my Graduate Studies at the University of Texas at El Paso under the guidance of Dr. Patricia Teller. I conducted a preliminary study in mobile ad-hoc network (MANET) parameter settings as my Master's thesis. The goal was to find the settings that minimize the amount overhead required to keep all network link information up-to-date. A fully automated physical test-bed was built to help facilitate this research and a publication is currently in the review process.

7.1 Publications

Joshua McCartney, Patricia J. Teller, Sarala Arunagiri, "Evaluation of Core Performance when the Node is Power Capped Using Intel® Data Center Manager," icppw, pp.246-253, 2012 41st International Conference on Parallel Processing Workshops, 2012.

Nigel G. Ward, Joshua L. McCartney, "Visualizations Supporting the Discovery of Prosodic Contours Related to Turn-Taking" Interdisciplinary Workshop on Feedback Behaviors in Dialog, 2012.