Evaluation of OSPF MANET Extensions

Boeing Technical Report: D950-10897-1[1]

Thomas R. Henderson
Phillip A. Spagnolo
Guangyu Pei

The Boeing Company
P.O.Box 3707, MC 7L-49
Seattle, WA 98124-7707

July 21, 2005

[1] This research was funded by the Office of Naval Research under contract N00014-04-C-0003. The simulator developed for this project is an extension of two open-source software projects: the Georgia Tech Network Simulator and the Quagga Routing Suite.

This page intentionally left blank.

Executive Summary

This technical report summarizes the results of a detailed comparative study of proposed MANET extensions to the Open Shortest Path First (OSPF), version 3 routing protocol for IPv6. In 2004, the OSPF Working Group at the Internet Engineering Task Force (IETF) chartered a design team to study alternatives for improving OSPF performance in such networks. During the course of the design team activities, the team has actively considered two proposals:

- A proposal known as **Overlapping Relays**, contributed by Madhavi Chandra (editor) from a team of authors at Cisco Systems, and
- A proposal known as **MANET Designated Routers** (MDRs), contributed by Richard Ogier.

We have developed an experimental implementation of each proposal and conducted comparative simulation studies, the key results of which are summarized herein:

- When compared with legacy OSPF Point-to-Multipoint interface performance, for a given number of nodes, both optimized flooding algorithms substantially reduce the amount of overhead due to OSPF operation in these networks, without compromising the routing/forwarding performance. Although results are scenario dependent, for the 50-node mobile scenario that we studied, the core optimized flooding algorithms reduced overhead by a factor of three. For small networks with similar levels of mobility (approximately one adjacency change every three seconds), there is improvement but it is not as significant since legacy OSPF can still be used.
- In comparative testing, both implementations achieved comparable levels of overhead reduction, although they employed different strategies. Briefly, Overlapping Relays uses non-active Multipoint Relays (MPRs) to reflood messages if acknowledgments are not heard from the target node, while MANET Designated Routers uses backup relays to reflood messages if the original flood was not overheard from the primary relays, and relies more on retransmissions along adjacencies.
- While flooding optimizations are promising, they also indicate a major limitation, in that the scaling behavior is still faster than the square of the number of nodes. Ogier's MDR proposal takes an additional step to leverage the MDRs to reduce the number of adjacencies and the advertised topology. The approaches outlined by Ogier's draft yielded significant reductions in overhead when compared even with the MANET-optimized flooding approaches of both Overlapping Relays and full-topology MANET Designated Routers, both in terms of absolute overhead reduction for a given number of nodes, and the scaling trend. The resulting forwarding performance is not substantially affected by such reductions although our results suggest that there is a tradeoff between how aggressive one tries to reduce the overhead and how much the forwarding performance is affected.
- A natural question is whether the above improvements shown in the MDR case can be realized also by Overlapping Relays. The MDR flooding backbone is shared and provides a framework for reducing superfluous adjacencies. However, Overlapping Relays use a source-dependent (i.e., not shared) flooding backbone, and we hypothesize that a similar strategy of reducing adjacencies to those between nodes and their relays will result in many more adjacencies. We also found that a recent proposal by Cisco to reduce adjacencies without leveraging the underlying flooding backbone resulted in either increased amount of overhead or a poorly synchronized network. It remains to be seen whether such a strategy for topology reduction in the Overlapping Relays framework can approach the performance demonstrated by MDRs.

Based on our evaluation in the context of OSPF extensions, we have concluded that the MANET Designated Routers (MDR) proposals for relay set selection and topology reduction are superior to those of the Overlapping Relays proposal. While both approaches yield comparable performance when just looking at optimized flooding algorithms in isolation, the MDR strategy for topology reduction leverages its flooding relay set to achieve simple heuristics for topology reduction that guarantee connectivity of the subgraph of adjacencies and that result in forwarding paths close to optimal. Our initial look at the performance of a similarly-proposed technique for Overlapping Relays that did not leverage any underlying relay set for adjacency formation did not dispel our intuition that it would be difficult to construct an algorithm that performed similarly well. There may be aspects of the Overlapping Relays proposal (such as dissemination of two-hop neighborhood information, or differential Hello mechanism) that might prove to be preferable to comparably proposed mechanisms in the MDR proposal, allowing a blend of the two proposals to perhaps be defined in the future, but we did not investigate such a hybrid design. Our recommendation is that the IETF focus on the MANET Designated Routers proposal as the core of its OSPF MANET extension design.

Table of Contents

Chapter 1

Introduction

This technical report summarizes the results of a detailed comparative study of proposed mobile ad hoc network (MANET) [CM99] extensions to the Open Shortest Path First (OSPF), version 3 [CFM99] routing protocol for IPv6. Previous experience with the OSPF protocol (e.g., [HSK03]) has shown the OSPF standard does not have suitable mechanisms for operating with high performance over wireless networks characterized by a broadcast-based transmission medium, in which the physical layer connectivity is not fully meshed.[1]

In 2004, the OSPF Working Group at the Internet Engineering Task Force (IETF) [2] chartered a design team to study alternatives for improving OSPF performance in such networks. During the course of the design team activities, the team has actively considered two proposals:

- A proposal known as **Overlapping Relays**, contributed by Madhavi Chandra (editor) from a team of authors at Cisco Systems [Cha05], and
- A proposal known as **MANET Designated Routers**, contributed by Richard Ogier [OS05].

We have developed an experimental implementation of both proposals and have conducted comparative simulation studies, the results of which are summarized herein. Our implementation is an extension of the `quagga`[3] OSPFv3 routing daemon. To support both experimental and simulation studies, we have developed a wrapper for the `quagga` routing daemon for the Georgia Tech Network Simulator [Ril03]; a packet-level, discrete-event simulator that supports detailed per-packet tracing, and simplified 802.11 or TDMA channel models.

This main body of the report is organized as follows:

- Chapter 2 introduces the technical problems introduced by OSPF operation in wireless, multi-hop, broadcast-based environments, and provides a high-level overview and comparison of the Chandra and Ogier drafts;
- Chapter 3 describes our experimental and simulation methodology, including the channel, radio, mobility, and traffic models used;
- Chapter 4 provides comparative performance results of the implementations of the two drafts.
- Chapter 5 provides a summary of the evaluation.

We also include five appendices to this report:

- Appendix A provides a software user's guide for those users who may want to repeat the results reported herein or who may want to conduct their own implementation- or simulation-based study,
- Appendix B provides background information on the modifications we made to `quagga` and *GTNets*,
- Appendix C provides additional detail on the implementation and validation of Overlapping Relays, and
- Appendix D provides additional detail on the implementation and validation of MANET Designated Routers.
- Appendix E looks at sensitivity analysis of the various configuration parameters of Overlapping Relays.

[1] IEEE 802.11 is an example of such a physical layer.

[2] http://www.ietf.org

[3] http://www.quagga.net

Chapter 2

Background

To briefly summarize the OSPF MANET problem statement, found in more detail in [Bak03], it has been repeatedly observed in experiments that OSPF does not have a suitable interface type for a wireless broadcast environment (which is characterized by a multicast-capable transmission medium that is not necessarily a full mesh, and which commonly has time-varying link availability and performance). One class of such networks has been called mobile ad hoc networks (MANETs), although there are other examples that are less ad hoc in nature. While a number of MANET routing protocols have been developed, their operation in large network deployments typically leads to the need to redistribute routing information between traditional IGPs and the MANET protocols, and hence a single, enhanced routing protocol would be operationally attractive.

The work on adapting OSPF to the MANET environment has its genesis in the MANET Working Group at the IETF, where four experimental protocols have been recently defined. One of the recent experimental protocols, Optimized Link State Routing (OLSR) [CJ03], resembles the basic operation of OSPF within a single area. In 2003, Boeing developed an adaptation of OLSR to fit it within the OSPF framework, and called it "Wireless OSPF" [HSK03]. This work showed that a periodic, unacknowledged flooding technique based on the principles of OLSR could, in some mobile network scenarios, reduce protocol overhead by an order of magnitude. Additional quantitative results can be found in [HP04].

In 2004, the OSPF Working Group agreed to commence work on a MANET extension for OSPFv3 (for IPv6), and formed a design team to come up with a recommended approach to the extension. The Working Group decided to focus on protocols that used reliable, acknowledged flooding, rather than periodic, unacknowledged flooding found in OLSR and Boeing's Wireless OSPF proposal, because reliable flooding is more in keeping with the design framework of OSPF. Two candidate proposals have emerged:

- **Overlapping Relays**, which is again similar to OLSR flooding but with reliability mechanisms [Cha05], and

- **MANET Designated Routers**, which uses an optimized, shared backbone for flooding and which contains further optimizations for topology control in dense networks [OS05].

Table 2.1 gives a high-level side by side comparison on the proposed designs. Below, we provide a brief description of the proposals.

Proposed Design	**Overlapping Relays**	**MANET Designated Routers**
Basis of design	Derived from Point-MultiPoint	Derived from Point-MultiPoint
Packet format extensions	Uses Link Local Signaling (LLS)	Uses Link Local Signaling (LLS)
Advertisement of 2-hop neighbors	Router-LSAs	Hello extensions
Optimized flooding	MPRs (source-dependent CDS)	MDRs (source-independent CDS)
Flooding backup	“Non-Active” relays	“Backup” MDRs
LSA retransmissions	along OSPF adjacencies	along OSPF adjacencies
Acknowledgements	multicast Acks	multicast Acks
Topology	Fully connected	Partially connected
Database exchange	Standard OSPF DB Exchange	Subset of LSA headers in the DD packet
Advertised links in router-LSA	All adjacent neighbors	Subset of adjacent and synchronized neighbors
Hello overhead reduction	“Incremental” Hellos	”Differential” Hellos

Table 2.1: Overview of ORs and MDRs.

Overlapping Relays (ORs) and MANET Designated Routers (MDRs) define a new OSPF MANET interface for wireless multihop networks. Because wireless multihop networks are multiaccess but not all nodes are biconnected, the interface type is derived from the OSPF Point-to-MultiPoint interface. Therefore, router-LSAs are built of a collection of point-to-point links to neighboring routers, and Network LSAs are not used.

ORs and MDRs utilize Link Local Signaling (LLS) to send arbitrary information between OSPF routers while maintaining backward compatability with legacy OSPF packet types. LLS is proposed in [NY05] for OSPFv2, and ORs and MDRs extend it to OSPFv3.

2.1 Overview of Overlapping Relays

Overlapping Relays (OR)s is a proposal by Cisco to extend the source dependent, efficient flooding techinque devloped for OLSR, [CJ03], to function in the reliable, acknowledged flooding evironment of OSPF.

In order to support efficient flooding, routers must be aware of their 2-hop neighborhoods. ORs advertise their 2-hop information in router-LSAs. A neighbor can identify its neighbor’s adjacent neighbors by looking at the point-to-point links in the neighbor’s router-LSA. For example, in Figure 2.2 node 3 can tell node 0 is adjacent with 1 and 2 by looking at its router-LSA. If node 3 looks at all of its neighbors’ router-LSAs then it can make a knowledgeable decsion on which nodes should flood its Link State Advertisements (LSA)s.

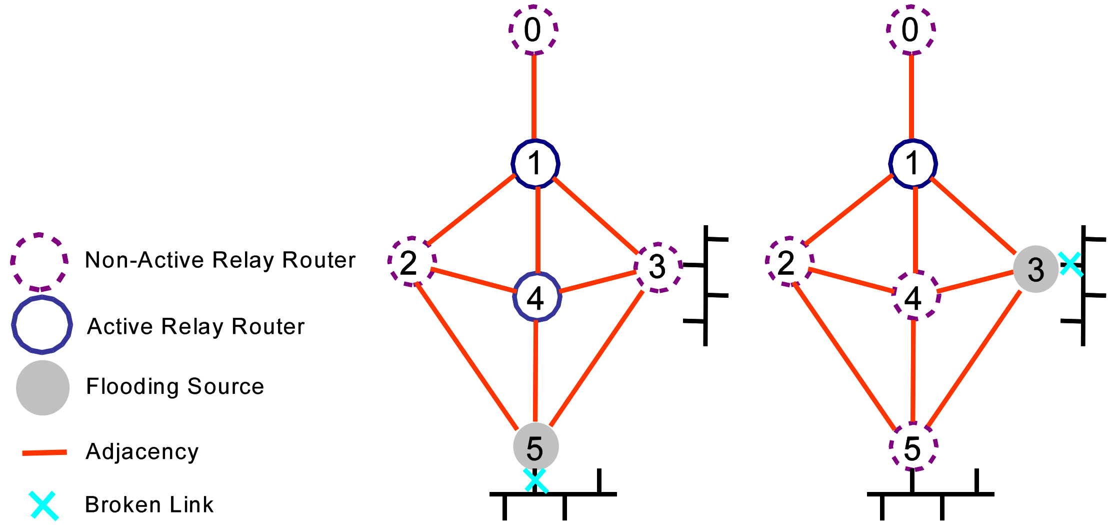
In legacy OSPF, such a decision is not made, and a flooded LSA is reflooded by all neighbors, but with ORs each router selects a subset of its neighbor to flood its LSAs. The router tries to pick the smallest subset of neighbors that can reach all of its two-hop neighbors. Each selected router becomes an Active Relay and each unselected router becomes a Non-Active Relay. When a router floods link state information, then each of its Active Relays will flood an LSA they haven’t previously flooded. For example, in Figure 2.1 (b) node 5 (the source) selects node 4 as an Active Relay so it floods node 5’s LSAs. Nodes 2 and 3 would normally flood, but they become Non-Active Relays. Node 4 then selects node 1 as an Active Relay, so the whole network is covered. In contrast, Figure 2.1 (c) node 3 (the source) selects node 1, and the whole network is covered. The different flooding sets make OR flooding Source Dependent (SD).

ORs extend MPRs with the use of intelligent acknowledgements (ack)s and Non-Active Relays. Each router that would have or did flood an LSA expects one acknowledgement of that LSA. To accomplish this, each LSA is acknowleded with a multicast ack. This acknowledges the LSA to neighbors who flooded the LSA and to those who suppressed flooding the LSA.

Non-Active Relays are not part of the flooding set; however, a Non-Active Relay will flood any link state information for which it would have normally flooded if it does not receive an acknowledgement. This flood occurs after the acknowledgement interval has expired and before the retransmission timer would expire.

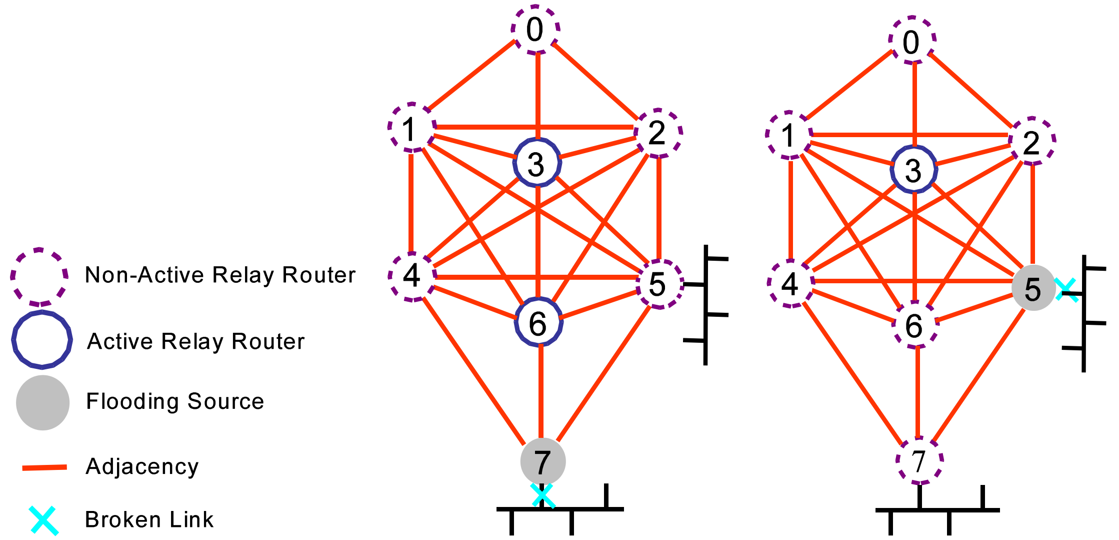
ORs become adjacent with all neighboring routers and they advertise all adjacent links in their router-LSAs. Figure 2.2 depicts the number of adjacencies that would be formed in a dense network. For instance node 3, would advertise six point-to-point links in its router-LSA. Recently, Cisco has explored ways to reduce the number of adjacencies when using ORs. Their technique is know as Smart Peering, and it has the option to include unsynchronzied links in the router-LSA. These figures do not show the reduction, but see Section 4.5 for more details.

ORs use incremental hellos to reduce the overhead associated with sending hellos. The goal of the incremental hello is to only send the change in state between hellos. Incremental hellos include a sequence number in each Hello. The sequence number is associated with the state of the neighbors, and it is incremented whenever any neighbor's state changes. A neighbor can verify that it has the neighbor's total state by verifying it has not missed a sequence number. If a sequence number is missed, then the full state of the neighbor is requested.



(a) Key (b) Flooding Source Node 5 (c) Flooding Source Node 3

Figure 2.1: Overview of Overlapping Relays.



(a) Key (b) Flooding Source Node 7 (c) Flooding Source Node 5

Figure 2.2: Overview of Overlapping Relays.

2.2 Overview of MANET Designated Routers

OSPF-MDRs optimize OSPF to support mobile ad hoc networks by using a Source Independent Connected Dominating Set (SI-CDS) to reduce flooding overhead and to reduce the number of adjacencies formed in dense MANETs.

MDRs use Hellos to obtain 2-hop neighbor information. The Hellos contain a list of 1-way (heard) and 2-way (reported) neighbors. For example, in Figure 2.4 node 0's hello would not contain a 1-way neighbor list and it would have nodes 1, 2, and 3 in its 2-way list. Using this information from all neighbors, a router can decide if it should flood LSAs.

MDRs optimize the flooding of LSAs (Link State Advertisement)s by forming an SI-CDS. The SI-CDS consists of MANET Designated Routers (MDRs) and Backup MDRs. The MDRs by themselves form a SI-CDS, and the MDRs and Backup MDRs together form a biconnected SI-CDS. The purpose of (Backup) MDRs in a MANET is similar to the purpose of the (Backup) DR in an OSPF broadcast network: to reduce the number of routers that must flood each LSA and to reduce the number of adjacencies. Based on 2-hop information, each router decides for itself if it will be an (B)MDR. Each MDR floods all received LSAs that have not previously been flooded. The flooding nodes do not change based on the source of the flood. In Figure 2.3 (b) and (c) for example, the same flooding set is used when either node 5 or 3 is the source.

Each router that would have or did flood an LSA expects an acknowledgement of that LSA. To accomplish this, each LSA is acknowleded with a multicast ack. This acknowledges the LSA to neighbors who flooded the LSA and to those who suppressed flooding the LSA.

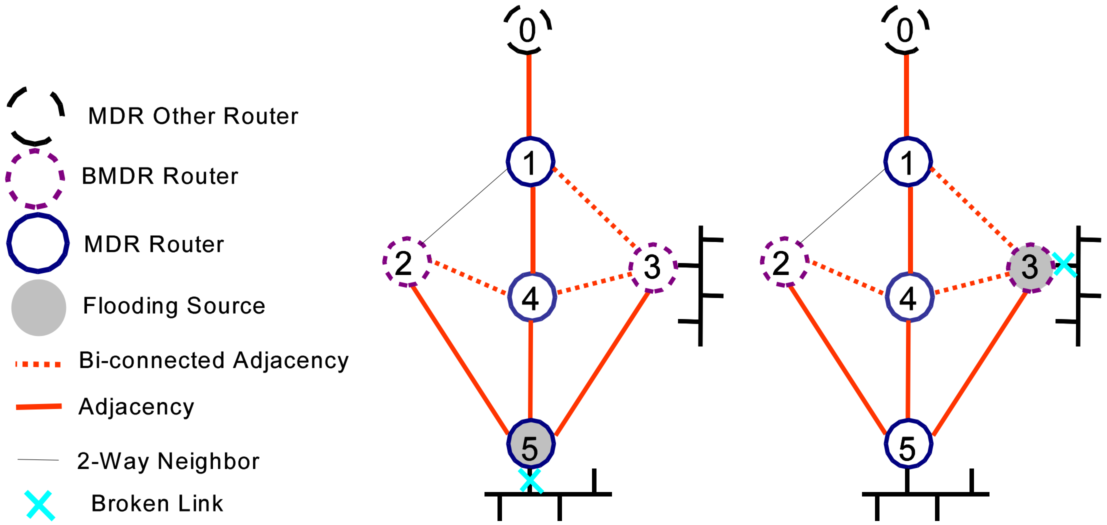
BMDRs provide redundant flooding coverage. The BMDR will only flood an LSA when it did not hear an MDR flood the LSA and it did not receive acks from its adjacent neighbors. MDR Other routers do not flood LSAs, but they may retransmit the LSA if it is not acknowledged.

Rather than have each router form adjacencies with all of its neighbors, each (Backup) MDR forms adjacencies with a subset of its (Backup) MDR neighbors to form a biconnected backbone, and each MDR Other forms adjacencies with two selected (Backup) MDR neighbors called "parents", thus providing a biconnected subgraph of adjacencies. The parent selection is persistent, so a router updates its parents only when necessary. The persistence of the (Backup) MDRs, combined with the persistence of the parent selection, maximizes the lifetime of the adjacencies. Figure 2.4 depicts the number of adjacencies that would be formed in a dense network. For instance node 3, would form adjacencies with three of its six neighbors.

A database exchange optimization is used in which a router (either the master or slave) does not include in its Database Description (DD) packets an LSA header if it knows the neighbor has the same or newer instance of the LSA. The already synchronized LSAs are determined based on DD packets received from the neighbor. Not including the LSA header is done by removing the LSA from the summary list when the same or newer LSA is received in a DD packet.

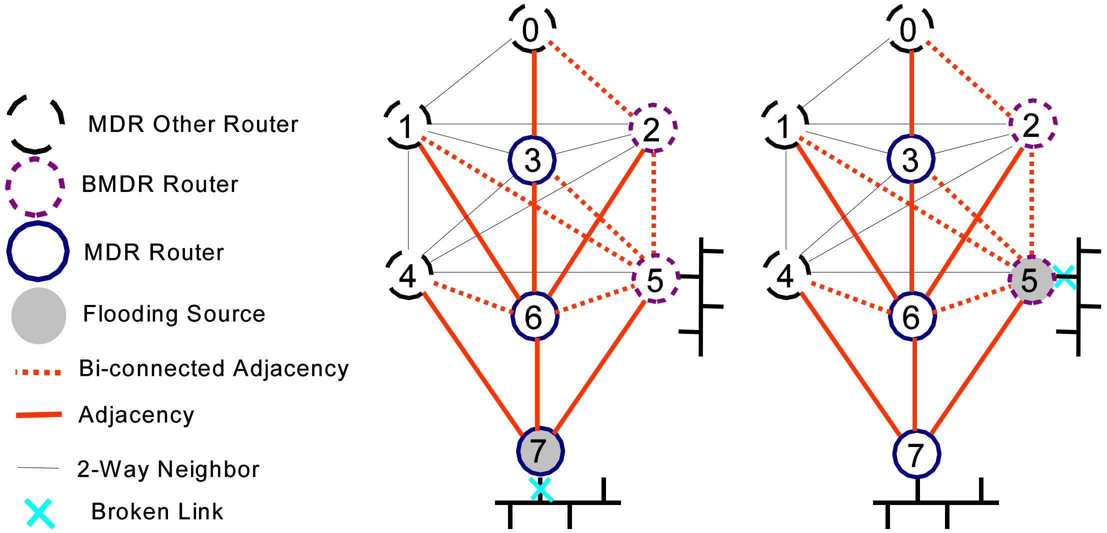
Router-LSAs minimally contain the adjacent neighbors. For example, Node 1 in Figure 2.4 would have two point-to-point links in its router-LSA. In addition, they can contain 2-way neighbors that are routable. The router is configured to either use full LSAs (all routable neighbors), (B)MDR full LSAs (min LSAs on other routers and full LSAs on (B)MDRs), near min-hop LSAs (all routers include routable neighbors that provide near min hops), or min LSAs.

MDRs employ Differential Hellos to reduce overhead due to full state hellos. MDRs send full state hellos every `2HopRefresh` hellos and Differential Hellos at all other times. Differential Hellos include a sequence number in each hello. The sequence number is incremented for each hello that is sent. When state changes occur, the change is sent in the next `HelloRepeatCount` hellos. If a node receives a hello which has a sequence number `HelloRepeatCount` greater than the last hello then it knows some state was lost. If state is lost, the router waits to the next full state hello to obtain full neighbor state information.



(a) Key (b) Flooding Source Node 5 (c) Flooding Source Node 3

Figure 2.3: Overview of MANET Designated Routers



(a) Key (b) Flooding Source Node 7 (c) Flooding Source Node 5

Figure 2.4: Overview of MANET Designated Routers

Chapter 3

Methodology

This section describes our overall approach to simulation evaluation of MANET extensions to OSPF. One problem in MANET routing simulations is that there exists an indirect mapping between configurable parameters in a simulation and the parameters that are most relevant to performance. For example, we first argue that the following parameters are most relevant to MANET routing protocol performance. For the purpose of discussion, we will define these as "intrinsic" performance parameters; parameters that are most relevant to the routing protocol regardless of the underlying protocol layers that provide them:

- **number of nodes.** The number of nodes is important when considering scalability and network diameter issues.
- **average neighbor density.** This can be defined as a ratio of the average number of neighbors (where a neighbor is a node physically within transmission range– see below) divided by the number of nodes in the simulation. This can be averaged across all nodes in the simulation. Such a definition yields a number in the range of roughly (0,1) (more precisely, [0, (N-1)/N]). This statistic, combined with the number of nodes, indirectly implies the average diameter of the network, which is relevant to routing.
- **average neighbor change rate.** This can be defined as the average number of changes to the neighbor set, per unit time, and also can be averaged across all nodes.
- **packet loss rate.** Statistics on the loss (or even reordering) of routing protocol packets are highly relevant. These statistics are most frequently a function of the distance between nodes, but also may be a function of whether the packet is unicast or multicast, the packet size, or the overall number of transmissions or interference in the network. Also, losses are often correlated (with fade events, for example), so simple representation as uniform losses is often not sufficient.

In an RF environment, the definition of a "neighbor" is imprecise. Is it a node that can receive 100% of transmissions? 80%? more than 0%? Neighbors at a routing protocol layer (e.g., OSPF neighbors) can also be different than actual physical neighbors, because OSPF will tend to hold onto neighbors for some period of time even after the nodes are out of range. Therefore, we also suggest that the term "neighbor" should be clearly defined in any simulation description, and should be linked to the physical relationship (not protocol relationship) between nodes (i.e., a physical relationship that directly impacts the observed packet loss rate).

However, when it comes to the actual knobs available for configuring the simulation, one can only make changes to many of the above parameters indirectly. Moreover, the configuration varies from simulator to simulator, such that a simulation run with particular configuration (e.g., 500 meter square grid with random waypoint mobility and 802.11 radios) could yield different results on different simulators, because the knobs translate to different intrinsic performance parameters depending on the models employed. Some typical "actual" parameters available for tuning are:

- **number of nodes.** This is one parameter that directly translates to an intrinsic parameter.
- **scenario topology.** Size and shape of the scenario, and possibly terrain and interference and other more sophisticated effects. This parameter, along with number of nodes, affects the neighbor density.
- **mobility model.** Node movement model, velocities of nodes. This parameter affects the neighbor change rate.

- **radio/MAC model.** This affects the neighbor density and loss rates, and can also affect the rate of change depending on certain parameters (e.g., for a fixed mobility model and scenario, different radio ranges will cause different rates of change of the topology).

Finally, the parameters lead to output performance statistics that are of most interest to routing protocols:

- **data delivery ratio.** How successful the network is in delivering packets without loss.
- **routing stretch.** How close to optimal are the paths that the packets traverse.
- **load distribution.** How well the approach makes use of available capacity in the network.
- **routing overhead.** Number of bits or packets consumed by the routing protocol itself.
- **processing.** Processing (CPU or algorithmic) requirements to operate the protocol.
- **convergence time.** Convergence of the routing tables, or convergence of other algorithms such as relay node set selection.
- **stability.** Stability of routing tables, neighbor sets, or relay node sets.

Figure 3.1 graphically summarizes the above discussion.

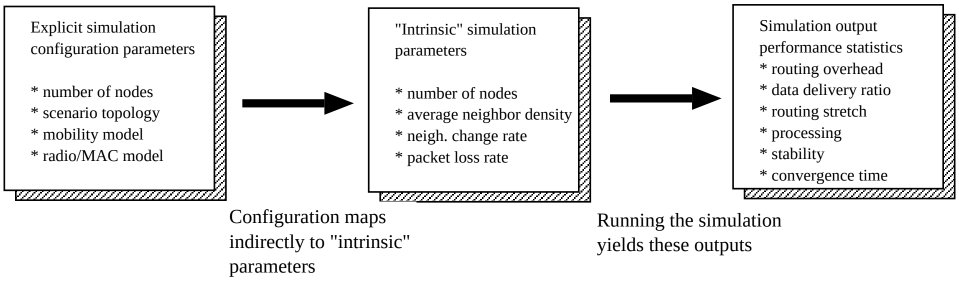

Explicit simulation configuration parameters
* number of nodes
* scenario topology
* mobility model
* radio/MAC model
"Intrinsic" simulation parameters
* number of nodes
* average neighbor density
* neigh. change rate
* packet loss rate
Simulation output performance statistics
* routing overhead
* data delivery ratio
* routing stretch
* processing
* stability
* convergence time
Configuration maps indirectly to "intrinsic" parameters
Running the simulation yields these outputs

Figure 3.1: Relationship of simulation configuration to output statistics.

3.1 Methodology Used

Ideally, it would be nice to express simulation runs purely in terms of intrinsic parameters; i.e., "invariant" parameters that might hold across simulation platforms. For example, if one specified the number of nodes, average neighbor density, etc., and were able to configure the underlying simulator to produce exactly those parameters, then one should be able to reproduce results across different simulation platforms.

Since it is difficult to parameterize the packet loss statistics (especially higher-order moments of the distribution), we have adopted a hybrid approach that combines mostly “intrinsic” parameters, with the use of radio/MAC model as a surrogate for the packet loss model:

- **number of nodes.**
- **average neighbor density.** We measure this by polling nodes every polling_interval seconds, and sampling the number of neighbors that each node has, and divide this by the total number of nodes, and average across all nodes.
- **average neighbor change rate.** We measure this by polling nodes every polling_interval seconds, and determining the count of nodes that either moved into or outside of range of the node, and then dividing this count over the time measured, and finally averaging across all nodes.
- **radio/MAC model.** These are a number of discrete models that we plug in. For example, three possible models are the default *GTNets* 802.11b model, a simplified contention-free radio/MAC model similar to 802.11b, and a TDMA-based radio/MAC model.

The radio model also governs how we define a physical neighbor. We define a physical neighbor as any node that is within the “radio range” of the given node, based on the radio/MAC model that is in use. We should be clear about the packet loss properties of nodes just within the radio range, such as the packet loss probability, and whether special packets (such as multicast, or large data packets) have higher or lower probabilities.

Again, our motivation here is that if we are able to describe our simulation runs in terms of the above parameters, someone else might be able to repeat them on a different platform (assuming the radio models were similar and accurate).

For example, consider the file `random_waypoint_manet.cc`. The number of nodes is configured as a run time command-line argument, and eventually is set within the simulator by the setting of the WirelessGrid topology:

```
  WirelessGridRectangular g(ca.mac_protocol,
                             l,
                             Constant(ca.nNodes), //number of nodes
                             Uniform(0.0, ca.grid_length),
                             firstIP);
```

which also sets the distribution of nodes within the grid, the size of the grid, and the mac protocol. The radio range is set separately, also by command-line argument, but eventually (within the program) by:

```
    n->SetRadioRange(ca.radio_range);
```

Mobility is added by the following line in this script:

```
  g.AddMobility(RandomWaypoint(g, Constant(ca.pause_time),
                                  Uniform(0,ca.velocity)));
```

Finally, the rate at which nodes are each checked for changes to their neighbor relationships is set as follows:

```
  ospf6d.LogNeighbors(0.5);  // Log physical layer neighbors every 500 ms
```

As a result, statistics will be compiled and printed out to the `.stat` file as shown in Figure 3.2.

```
Neighbor statistics (at physical layer):
--------------------------------------------------
Number of nodes: 20
Neighbor threshold distance (m): 250.00
Average neighbor density (neighbors/# of nodes):  0.58
Average neighbor changes/time (changes/sec/node): 0.10
```

Figure 3.2: Sample neighbor statistics outputted to the .stat file.

Summary

Therefore, we can explore the simulation space as follows. For each radio/MAC model, define what it means to be a neighbor, in the physical sense. This then leaves three independent parameters to vary:

- **number of nodes**
- **neighbor density**
- **neighbor change rate**

In general, for a parameter of interest (such as overhead), this would require a four-dimensional plot to observe how the overhead changes as a function of all three above parameters. Instead, to break this down into more manageable two-dimensional plots, we suggest to plot a performance parameter (such as overhead) against the number of nodes, for a given density and change rate. What this implies is that we should select a few representative values for these parameters (e.g., the four quadrants shown in Figure 3.3) and then define these, such as "low density, high mobility" scenarios, and then plot out the statistic of interest as a function of the number of nodes. The question of how to parameterize what constitutes meaningful network density and change rates (e.g., if constructing four scenarios, where to put the quadrant boundaries in Figure 3.3), should be drawn from experimental experience or projected network scenarios. Since parameters of neighbor change rate and neighbor density are somewhat a function of the number of nodes, a fixed value of the number of nodes (e.g., 20) should be used when trying to initially characterize the scenario.

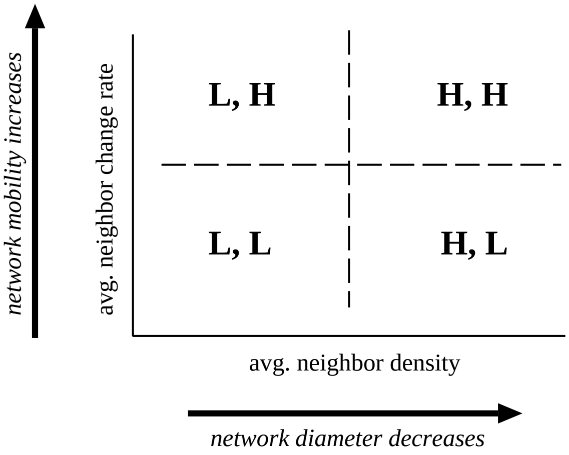

network mobility increases
avg. neighbor change rate
L, H
H, H
L, L
H, L
avg. neighbor density
network diameter decreases

Figure 3.3: Simulation scenarios defined by the pair of "neighbor density" and "neighbor change rate"; in this example, four scenario combinations of "low" and "high" density or change rate are depicted.

Such a process yields the following general steps to obtain meaningful simulation results:

1. Select and properly characterize the radio/MAC model of interest
2. Define what a physical neighbor is, in the context of the radio/MAC in use
3. Define some set of parameter pairs (average neighbor density and neighbor change rate) that characterize the mobility scenarios of interest
4. Determine how the scenario topology and mobility models need to be tweaked to yield the appropriate parameter pairs
5. For each parameter pair, define the set of "number of nodes" that are of interest. The combinations of these 3-tuples (number of nodes, neighbor density, neighbor change rate) consitute the simulation configurations.
6. For each simulation configuration, run the simulation for long enough time and/or with enough random seeds to obtain low variances in the output statistics of interest.

3.2 Radio and MAC Models, and Neighbor Definitions

3.2.1 802.11b Model

The radio model of *GTNets* is a threshold based model (illustrated in Figure 3.4). It defines two thresholds, namely, the carrier sensing threshold (`csThreshold = pow(10, -7.8); // -78 dBm`) and recieving threshold (`rxThreshold = pow(10, -6.4); // -64 dBm`). In the case of interference, the SNR threshold required for receiving the packet correctly is defined by (`snrThreshold = 10; // 10 dB`). When a transmitter starts a transmission, it computes the RX power for all nodes within the radio range which is set by user. The following scenarios can occur (assuming the receiver is not transmitting; otherwise, nothing will happen):

- If the RX power is less than `csThreshold`, the model does nothing.
- If the RX power is between the `csThreshold` and `rxThreshold` and the receiver is idle, the model will schedule a link event for receiver and the packet is marked with an error.

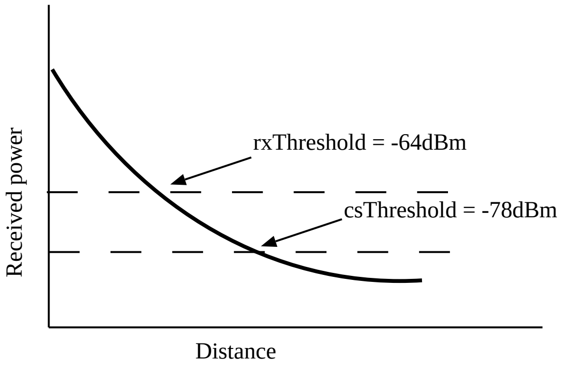


Figure 3.4: GTNets Radio Model

- If the RX power is between the `csThreshold` and `rxThreshold` and the receiver is busy, the model will compare SNR threshold with the ratio between the previous RX power at the receiver and the calculated RX power. If the ratio is less than the SNR threshold, the previous packet is marked as an errored packet.
- If the RX power is greater than `rxThreshold` and the receiver is idle, it will schedule an RX event and mark the packet with no error.
- If the RX power is greater than `rxThreshold` and the link is busy, the model will compare the SNR threshold with the ratio between the previous RX power at the receiver and the calculated RX power. If the ratio is less than the SNR threshold, the previous packet is marked as an errored packet.

It is clear that this model is a on-off radio model. If there are no collisions, a node can receive a packet iff the received power is greater than `rxThreshold`. However, field measurements[1] indicate otherwise. In order to model such behavior, bit error ratio (BER)-based models are required. Due to time limitations, we developed the following approximation.

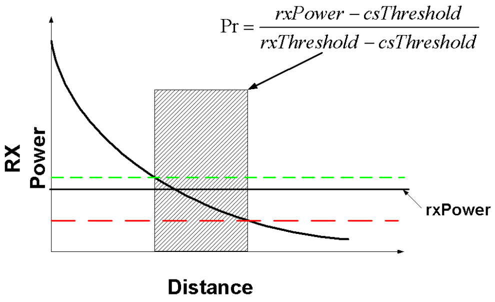


Figure 3.5: Modified GTNets Radio Model

As we discuss in section B.5, each node has to set transmission power properly to synchronize the view between the channel and upper layer. The next question is what power value the upper layer should use. By default, *GTNets* uses `rxThreshold` (after the fix described in section B.5). In order to approximate a radio that can receive packets within a grey area, we set the TX power at the transmitter such that the received power falls between `csThreshold` and `rxThreshold`. Our model is illustrated in Figure 3.5. Instead of marking a packet solely based on a single `rxThreshold` as in the original *GTNets* we use the following formula to calculate the error-free probability:

$P_r = \frac{RxPower - csThreshold}{rxThreshold - csThreshold}$

It has the following properties: (1) if $RxPower = csThreshold$, $P_r = 0$; and (2) if $RxPower = rxThreshold$, $P_r = 1$. The packet is marked based on this probability.

[1] "Link-level Measurements from an 802.11b Mesh Network", D. Aguayo, J. Bicket, S. Biswas, G. Judd and R. Morris, In Proceedings of Sigcomm 2004.

We use a parameter α to control the received power at the specified range. Figure 3.6 shows the relation among the `csThreshold`, `rxThreshold` and `rxPower`. If $\alpha = 1$, we fall back to the on-off radio model. Since the radio range is determined by the TX power, two-way path loss channel model and the above formula, there is no simple relationship between α and radio range. However, the smaller α the shorter range. Note that, *GTNets* will not include any nodes beyond Radio Range in `node->BuildRadioInterfaceList()` which is before all power calculations. Thus, the specified range is the absolute upper limit. The above simiplified SNR based model only creates a grey area below this upper limit.

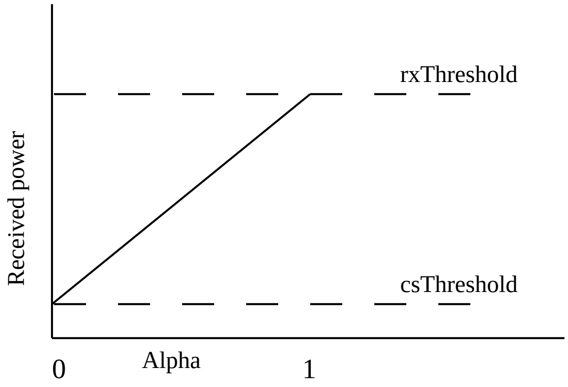


Figure 3.6: Definition of α

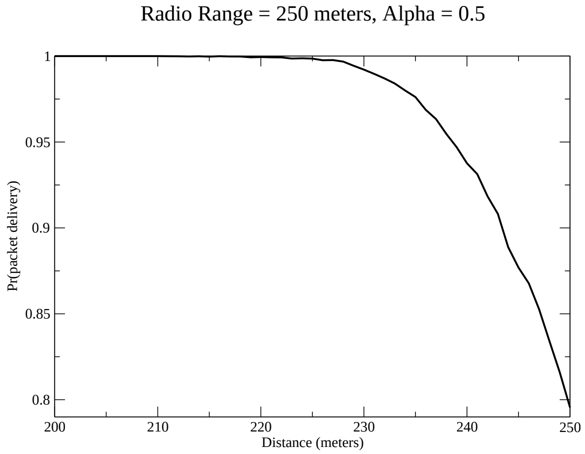


Figure 3.7: Sample of radio range plot

Figure 3.7 shows an example of a 2-node radio network. The radio range is set to be 250 meters. The packet delivery probability is measured for a total of 19996 packets generated by the CBR traffic source. The delivery probablity is 1 until the distance reaches 210 meters. After that, it gradually drops to 80% at the 250 meter range, beyond which the delivery probability is 0 due to the aformentioned reason.

In our simulations with this radio model, we declare a node to be a neighbor at the physical layer if it is within this radio range parameter (within 250 meters by default).

3.2.2 Simplified 802.11b Model

The existing 802.11 model (found in `SRC/l2proto802.11.{cc,h}`) implements a fairly full 802.11 state machine, including IFS spacing of transmissions, beaconing, and backoff. It also implements various timer events, including NAV, CTS, DATA, ACK, and backoff timeouts.

A simplified MAC that uses essentially the same physical layer, but that strips out the MAC state machine, is available (found in `SRC/l2protosimple.{cc,h}`). This code also simplifies the wireless LAN `Transmit()` method, requiring only one pass through the node list per transmission, rather than (the worst case scenario of) two. The simple MAC assumes no transmission collision possibility, and does not reliably retransmit either unicast or multicast frames. When a packet is available to send, the transmitter decides (probabilistically) whether the packet can reach the intended destination, and if so, schedules a successful receive event there.

The run time savings using this model is a function of the configuration and density of the network. We have observed run time improvements for up to a factor of three between using the legacy 802.11 MAC and the simplified MAC. For example, the `ietf/random_waypoint_mobility.cc` script, configured for 40 nodes, yields approximately 25% run-time improvement when using the simplified MAC.

Note that this simplified MAC will give slightly different results compared to a full 802.11 model. Most notably, unicast frames do not have the benefit of retransmissions anymore, causing a larger percentage of simulated user data to be lost. The multicast behavior (no collisions) also is different for OSPF as well. Therefore, it is not appropriate to use this model to compare results with 802.11, or in case you want to see the effects of collisions or link layer retransmissions.

How to enable: Select `mac_protocol=2` as a run-time option in files such as
`ietf/random_waypoint_manet/random_waypoint_manet-{opt,dbg}`, e.g.,
`random_waypoint_manet-dbg mac_protocol=2`. This option for `mac_protocol=2` may not be available for all scripts; using this on other scripts requires modifying them along the lines of the example
`random_waypoint_manet.cc` script mentioned above.

3.2.3 TDMA Model

The Time Division Multiple Access (TDMA) model implemented in GTNetS is based on the Unifying Slot Assignment Protocol (USAP) TDMA frame architecture. Figure 3.8 shows the TDMA slot allocation structure. Each TDMA frame has three types of slots: bootstrap slots, broadcast slots, and reservation slots. The bootstrap slots are used for nodes to exchange TDMA control information so that each node can keep synchronized with neighboring nodes, make reservations for the reservation slots, and resolve any slot assignment conflicts due to dynamics such as mobility. The broadcast slots are used to support datagram services and any other control traffic shared by nodes. If the reservation slots are not allocated, they serve as standby broadcast slots. In our initial implementation, each node has its own data slot reserved in the reservation slots.

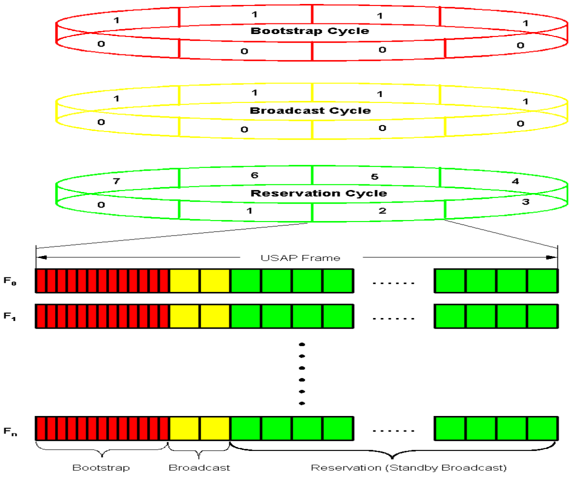


Figure 3.8: TDMA Frame Architecture.

One distinct feature of the USAP protocol is that each type of slot has its own cycle as shown in Figure A-7. In this example, the cycle for bootstrap and broadcast slots is four TDMA frames followed by one frame for reservation slots. For a small network size (less than 52 nodes), it is sufficient to assign one permanent bootstrap slot per node. This is what we implemented in the simulation model. Although our TDMA model can support any network size, the performance will degrade as the network size increases above 50 nodes unless the following advanced features are implemented:

- **Adaptive bootstrap cycles.** As the network size grows, we must spatially reuse the bootstrap slots. Otherwise, either the control slots will consume too much bandwidth or the cycle length will be too long to exchange critical USAP control information in a timely manner.
- **Data aggregation.** Figure 3.9 shows the histogram of the packet size received at the MAC for a 20-node network. Currently, the default maximum transfer unit (MTU) of 1500 bytes is configured for the OSPF interface. In this example, the median packet size of a total of 223,447 packets is 76 bytes. If we set TDMA slot size according to the default 1500 bytes (as currently done in the simulator), much of the bandwidth is wasted since only one layer 3 packet is sent per MAC frame. For small network size (less than 50 nodes) with a high speed data channel (e.g., 11 Mb/s), this might not be a problem. However, when the network size increases or channel data rate reduces (e.g., 2 Mb/s), such inefficiency will cause buffer overflow and long delays. As a result, OSPF will not converge and the network will no longer be usable. In order to improve the efficiency, data aggregation is needed, although USAP itself does not specify how the aggregation should be done. One might propose a solution that sets the MTU to a smaller size and forces the network layer to fragment packets. However, the simulation results show that network performance degrades in this case. For example, with MTU equal to 300 bytes and a 2 Mb/s 802.11 MAC protocol, the packet delivery ratio is only 0.53 for a 40-node network, compared with 0.8 for an 11 Mb/s network with MTU of 1500 bytes.
- **Heuristic reservation slot assignment algorithms.** Under the heavy load conditions, a more efficient slot assignment rather than the default reservation scheme is needed.
- **Reliability for unicast traffic.** Currently, there is no retransmission for unicast data if there is an error. This will affect the UDP packet delivery ratio compared with the 802.11 MAC when . < 1.

In summary, the TDMA model implemented in GTNetS is a generic TDMA architecture. It can be mapped to a particular TDMA radio if detailed information is provided. The scheduling part can be directly ported if the implementation code of such a TDMA protocol is available.

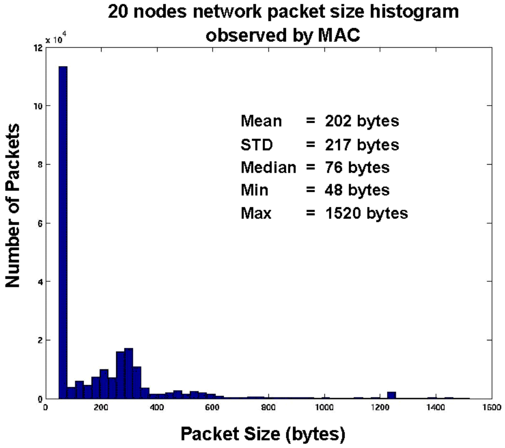


Figure 3.9: Profile of packet size distribution for a 20 node simulation, as observed at the MAC.

3.2.4 Link Matrix

LinkMatrix is new class defined in *GTNets*, based upon the Ethernet class. LinkMatrix allows a user to create a network with deterministically defined connectivity. By default, nodes connected by LinkMatrix will be fully connected (just like Ethernet) in a single-hop, broadcast-based environment. But, a user may call the function `LinkMatrix::ScheduleMatrixChange()` to specify the packet error rate between any two nodes on the

LinkMatrix. The error rate can be configured to change as many times as desired, to script how the connectivity between any two nodes might change.

For example, `ScheduleMatrixChange()` is called as follows `ScheduleMatrixChange(1900, 0, 1, .5)`. Then, at any time greater than 1900 seconds, packets sent from node 0 and to node 1 will be lost with a 50% probability. If the error rate is set to 1, then all packets will be lost between the two nodes.

This class is very useful for testing the validity of particular mechanisms in the MANET extensions. See Chapter A.1.5 for examples.

Chapter 4

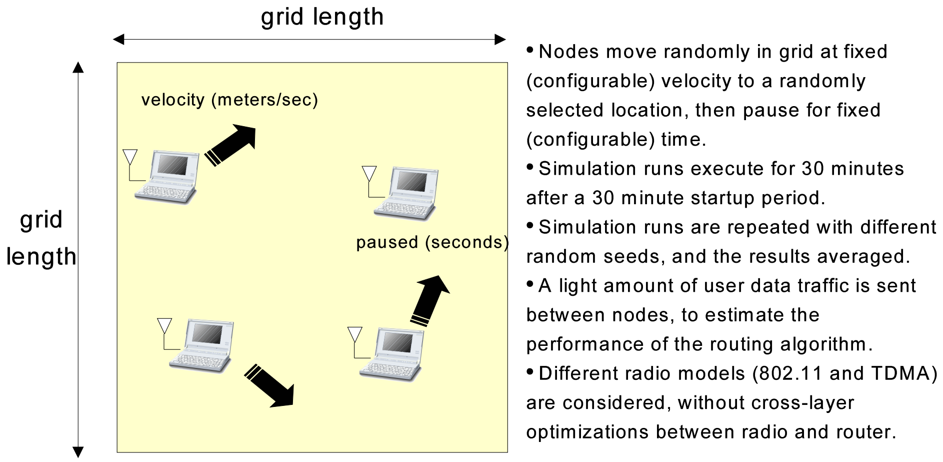
Comparative Results

Our simulation objective was to compare the core performance attributes of

- multicast-capable Point-to-Multipoint OSPFv3 interface (using standard flooding),
- Cisco’s proposed Overlapping Relays flooding, and
- Ogier’s proposed MANET Designated Routers flooding and topology reduction.

4.1 Scenario Description

The results in this section are based on a simple MANET scenario using an 802.11 MAC. Nodes moved around randomly on a square grid, as shown in Figure 4.1 (note: we did not use TDMA for these runs, and we limited runs to 30 + 15 minutes duration). Table 4.1 lists some key configuration parameters in use by our simulation. In our simulation, most paramters were set to the default values; non-default parameters are in **bold text**.



- Nodes move randomly in grid at fixed (configurable) velocity to a randomly selected location, then pause for fixed (configurable) time.
- Simulation runs execute for 30 minutes after a 30 minute startup period.
- Simulation runs are repeated with different random seeds, and the results averaged.
- A light amount of user data traffic is sent between nodes, to estimate the performance of the routing algorithm.
- Different radio models (802.11 and TDMA) are considered, without cross-layer optimizations between radio and router.

Figure 4.1: Simulation scenario.

Parameter	**Definition**	**Value or range of values**
`seed`	Random number generator seed	1 (default)
`start_time`	Ramp up time for simulation	1800 sec (default)
`stop_time`	Stop time	2700 sec (default)
`pktsize`	UDP packet size for data traffic	40 bytes (default)
`pktrate`	UDP packet rate sent across whole network	10 pkts/sec (default)
`num_nodes`	Number of nodes	**10 through 110**
`radio_range`	Max. radio range; see Chap. 6	250 m (default)
`alpha`	Link behavior at fringe of coverage; see Chap. 6	0.5 (default)
`grid_length`	Length of square grid	500 m (default)
`velocity`	velocity of node movement	10 m/s (default)
`pause_time`	node pause time between movements	40 s (default)
`wireless_interface`	type of interface	**PTMP, MANET**
`wireless_flooding`	type of MANET flooding	**broadcast, MPR, MDR**
`HelloInterval`	OSPF parameter	**2 sec**
`RxmtInterval`	OSPF parameter	**6 sec**
`MinLSInterval`	OSPF parameter	5 sec (default)
`MinLSArrival`	OSPF parameter	1 sec (default)
`PushbackInterval`	Overlapping Relays timer	2000 ms (default)
`AckInterval`	Overlapping Relays timer	1800 ms (default)
`diff_hellos`	Incremental Hellos for O. Relays	**off, on**
`BackupWaitInterval`	MDR timer	2000 ms (default)
`AckInterval`	MDR timer	1800 ms (default)
`TwoHopRefresh`	MDR timer	**3 Hellos**
`RouterDeadCount`	MDR parameter	3 (default)
`AdjConnectivity`	Degree of adjacency connectivity	**uni-, bi-, or full**
`LSAFullness`	Topology reported in LSAs	**adjacent, min-hop, full for MDR/BMDR, full**
`NonPersistentMDR`	Flag to make MDRs non-persistent	off (default)
`diff_hellos`	Differential Hellos for MDRs	**off, on**

Table 4.1: Main configurable simulation parameters.

Finally, for MANET Designated Routers (MDR), we used the max-priority-neighbor (MPN) algorithm (Section 4.2 of Ogier's draft) with `h1=3,h2=INF`. We have not experimented a lot with this parameter– only to observe that setting this value to 2 does not affect results much (this is why this parameter is not exposed as a command-line configurable parameter). This parameter can be changed and recompiled in the file `SRC/ospf6d/ospf6_interface.c`.

In short, we are interested in studying, in a MANET setting, at how the two proposals (Overlapping Relays (OR) and MANET Designated Routers (MDR)) compare with each other and with legacy Point-to-Multipoint. We used many default configuration parameters listed above. We varied the number of nodes from 10 up to 110, and we varied the basic OSPF configuration, trying to keep as many things as similar as possible through the runs. The data traffic only provides a light load on the network, to measure the forwarding performance; performance under heavy network load has not been studied.

Note: The code, scripts, and data from this chapter can be found in the ietf/random_waypoint_manet/ directory, and are the result of compiling with the -opt option ("make ietf-opt").

4.2 Baseline Comparison

The next page (Figure 4.2 displays performance results for the following three configurations:

- Baseline Point-to-Multipoint (PTMP)
- Overlapping Relay (OR)
- MANET Designated Router (MDR)

To try to equally compare OR and MDR algorithms, the following changes were made to each.

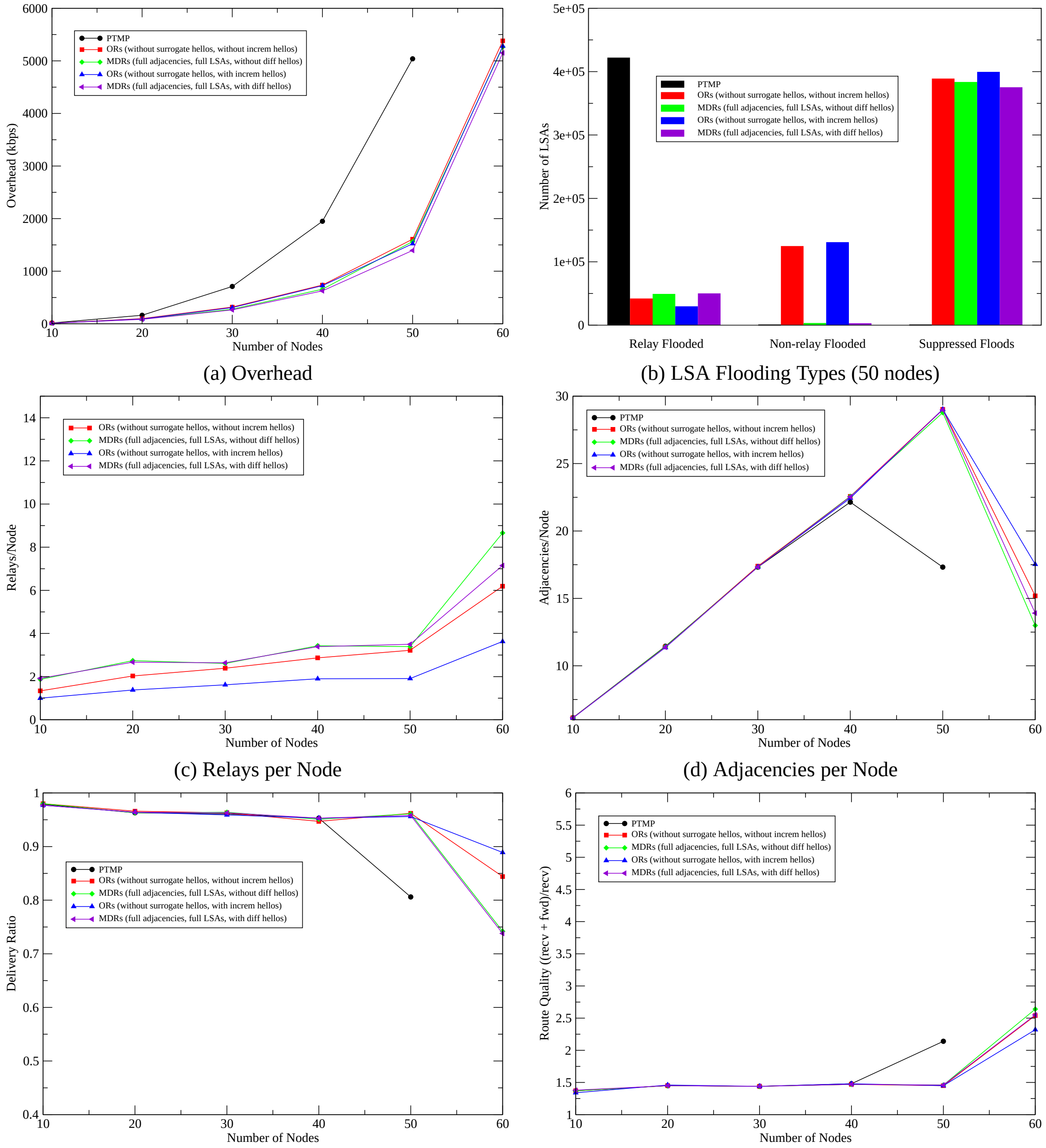
- For Overlapping Relays, we disabled the capability to treat ACKs as surrogate Hellos. Note that this is a difference between the Overlapping Relays and MDR proposals, but is equally applicable to both; we elected to remove this capability for this comparison, but will later show results with it enabled in Section 4.6.
- For MANET Designated Routers (MDR), we enabled a full topology mode. Note that in Richard Ogier's proposal, he discusses the formation of adjacencies only between MDR/BMDR routers and (backup) Dependent Neighbors. For these comparisons, the MDR case was run in a full topology mode (with every node trying to form an adjacency with every neighbor in state greater than TwoWay, regardless of whether the nodes are MDRs or not, and with LSAs reporting full adjacency information). This mode is not a mode intended by Ogier's draft, but is useful for a more equal comparison with the Overlapping Relays implementation. Later, we will show results that include the adjacency reductions described in Ogier's draft. In all results, we did not include an optimization by Ogier which changes the LSAs reported in Database Description packets. This should be tested in the future work.

Appendices C and D provide additional implementation details of our implementations of Overlapping Relays and MANET Designated Routers, respectively.

In summary, the results below are designed to highlight the differences between the core reliable MPR algorithm (Overlapping Relays) and the core "Essential CDS" (MANET Designate Router) algorithm, having disabled the latter's capability to reduce the number of adjacencies in the graph.

Note: To reproduce these results, execute the following scripts:

```
./comparative_ptmp.pl
./comparative_mpr.pl
./comparative_mdr.pl
```



(a) Overhead

(b) LSA Flooding Types (50 nodes)

(c) Relays per Node

(d) Adjacencies per Node

(e) Delivery Ratio

(f) Route Quality

Figure 4.2: Baseline comparative results

Next, we provide some discussion of the performance results shown in the above plots. We note here that the influence of incremental or differential hellos is minimal, so we did not study the details of the difference shown in these plots.

4.2.1 Overhead and Flooding Statistics

When compared with Point-to-Multipoint overhead (which increases by a factor of roughly n^3 with n the number of nodes in this scenario), the overhead is reduced by a factor of roughly two for 40 nodes and three for 50 nodes (i.e., it also scales slightly better). The 50 node case for Point-to-Multipoint actually caused overload (congestion) in the channel, which is manifested as performance degredation in some of the later plots. We see the same overload for ORs and MDRs begin at 60 nodes and very prevalant at 70 nodes.

A key result is that the flooding overhead is not substantially different between the OR and MDR algorithms. We do see that MDRs seem to degrade less than ORs at 60 nodes. Note that since these results are from a single simulation run, differences may not turn out to be statistically significant across more runs.

Although both MANET approaches yield comparable overhead, how they arrive at that total is actually quite different. One of the key differences is in how the two proposals protect (make reliable) their LSA floods. In the OR case, non-active relays wait until they hear an ACK or reflood from the next-hop neighbors that the active relays are covering, and if they do not hear such ACK in a short interval, one non-active relay will fire and the others will back off again. In contrast, the MDR approach uses backups that are not keyed off (lack of) reception of ACKs or refloods from downstream nodes (although the reception of such ACK will take the LSA off the flooding/retransmission list), but instead, decide to become "active" if they do not hear the initial flood from the DRs. In addtion, all non-active relays will pushback an initial flood for ORs, but only MBDRs will backup an intial flood for MDRs.

As a result, we observe in the data that, in the OR case, most of the overhead is due to flooding (about 47% of the total LSU overhead), both by active and non-active relays, and that retransmissions are much lower (about 5% of the total LSU overhead). In particular, in this case, there are roughly three times the number of non-active floods than active floods. In contrast, the MDR case results in much less overhead due to flooding (about 15% of the total overhead), but much more due to retransmissions (about 27% of the LSU overhead was unicast retransmissions). In this sense, the OR approach is more proactive (preemptive flooding by non-active relays, rather than waiting for retransmissions to occur).

4.2.2 Size of Relay Set and Number of Adjacencies per Node

The next two graphs provide more details on the size of relay sets for each approach, and the number of adjacencies per node. To measure the size of the relay set (normalized per node), for MPRs, we tracked the total number of active ORs per node over time, and averaged the results. For MDRs, where the DRs perform relaying for every node, we averaged the size of this DR relay set over time, then divided by the number of nodes.

The results suggest that the size of the relay node set (averaged per node) is larger when MDRs are used. We have observed this consistently in our simulations. Note that Ogier's draft proposes that the size of the relay set is tunable (can be traded off for robustness or for obtaining better routes); these results are for the MPN algorithm with `h1=3,h2=INF` only. We have not experimented much with the algorithm variations in Ogier's draft. Although MDRs have a larger relay node set size on average, the resulting overhead does not seem to be sensitive to this, as can be seen in Figure 4.2 (a). The number of adjacencies per node, plotted in Figure 4.2 (d), does not show variation.

4.2.3 Delivery Ratio and Route Quality

Finally, the last two plots show the performance of the small amount of user data traffic configured to flow on this network. Figure 4.2(e) plots the user data delivery ratio for all three approaches, and Figure 4.2(f) is a measure of the route quality or "routing stretch"; specifically, the number of UDP data receive events plus the number of UDP data forward events divided by the number of receive events (essentially, this is the average number of hops that a packet must travel from source to destination). The lower this metric, the fewer hops, on average, that packets need to take.[1] Note that this metric is between one and two when the network is not overloaded, indicating that this dense network is

[1] We also should note a general subtle point about the relationship between delivery ratio and route quality, as we have defined them– a higher delivery ratio may yield a lower average route quality, if the packets that are delivered in only a particular scenario happen also to have a long path.

successful in delivering packets in one or two hops, typically, and that the performance is comparable to that of legacy OSPF.

The plots show that the reduction in overhead for either approach does not come at the expense of data delivery performance (either raw number of packets received successfully or the routes used to forward them). The outlying poor performance in the plots is that of Point-to-Multipoint for 50 nodes and ORs and MDRs for 60 and 70 nodes, where, as indicated above, there is so much congestion in the 802.11 network due to excessive routing protocol overhead that data delivery is impaired.

4.3 Sensitivity to Varying Mobility and Density

In this section, we compare the effect of the neighbor change rate and neighbor density on the performance of ORs and MDRs. For this data, we ran ORs without surrogate hellos and incremental hellos, while we ran MDRs without differential hellos, with Full LSAs, with full adjacencies, and without the database exchange optimization for fair comparison. We used the same four quadrant definition of neighbor change rate and density in Figure E.2. The following four figures 4.3, 4.4, 4.5, and 4.6 were run for 0 to 50 nodes with 4 random seeds for each data point. Error bars indicating the standard deviation are shown on the plots.

Note: To reproduce these results, execute the following script:
./comparative_density_mobility.pl

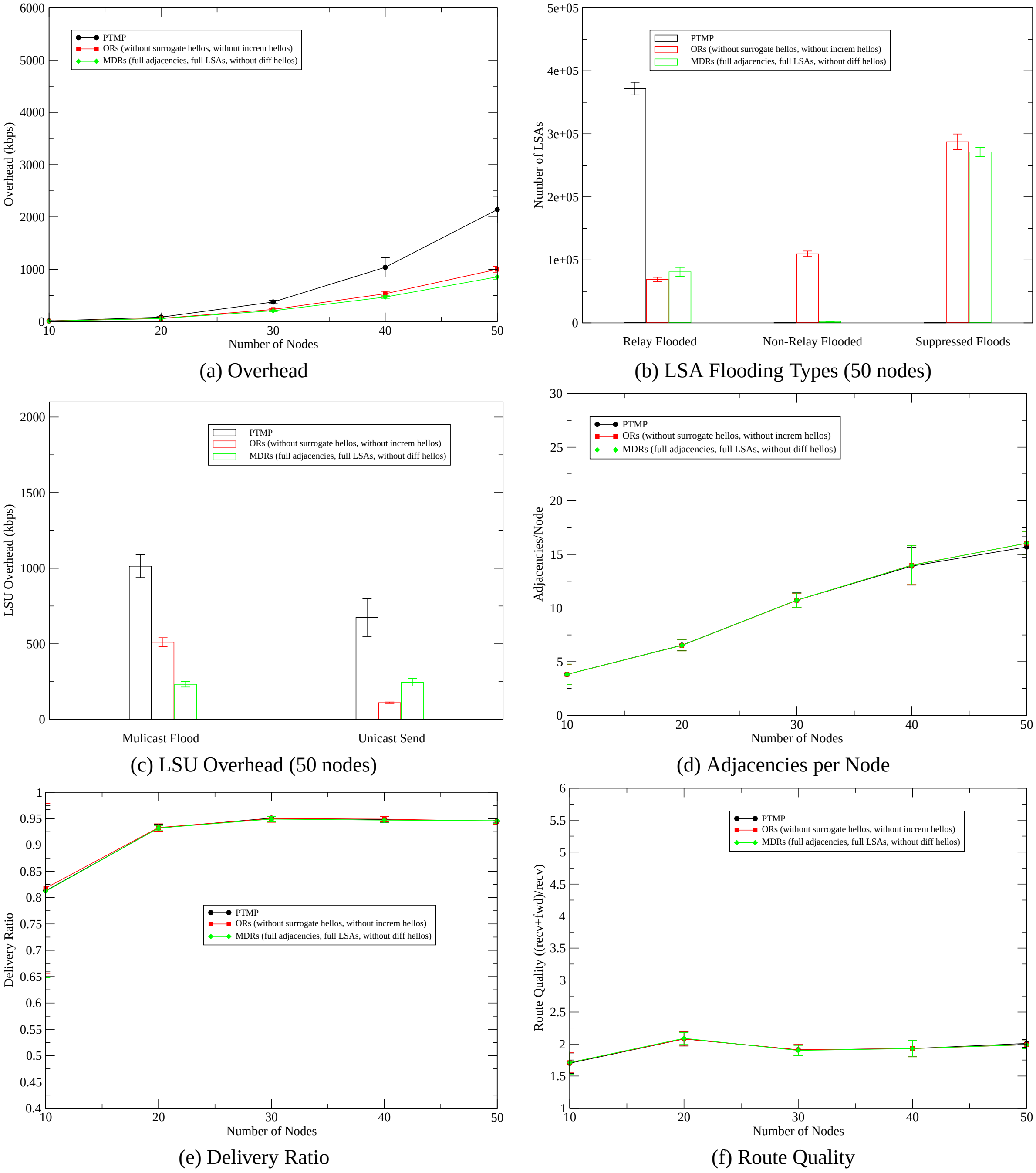
In general, the same conclusions in the previous sections of this chapter are drawn here. This data is instructive because it shows that the conclusions drawn from the single seeded run in previous sections hold when multiple runs are averaged.

The overhead in (a) is found to be about the same for both ORs and MDRs. However, the MDRs show a consistent but minor improvement over ORs for each of the overhead plots.

The LSA flooding types in (b) continue to show the trend that ORs tend to flood more LSAs using Non-Active Relays than the BMDRs flood. This is both due to the mechanism to cancel a backup and the number of backups in the network.

The LSU overhead (c) is a new plot in this section. The plot identifies the overhead created by LSUs and whether it is due to a unicast or a multicast send. Unicast sends are mostly due to LSA retransmissions. We see from the plots the MDRs tend to have more unicast LSUs than ORs while ORs tend to have more multicast LSUs than MDRs. This is again due to the the mechanism to canel a backup relay and the number of backups in the network. MDRs tend to use retransmission for backup while ORs use flooding.

The adjacencies per node (d), delivery ratio (e), and route quality (f) are basically identical for ORs and MDRs. This is consistent with the previous section.


(a) Overhead

(b) LSA Flooding Types (50 nodes)

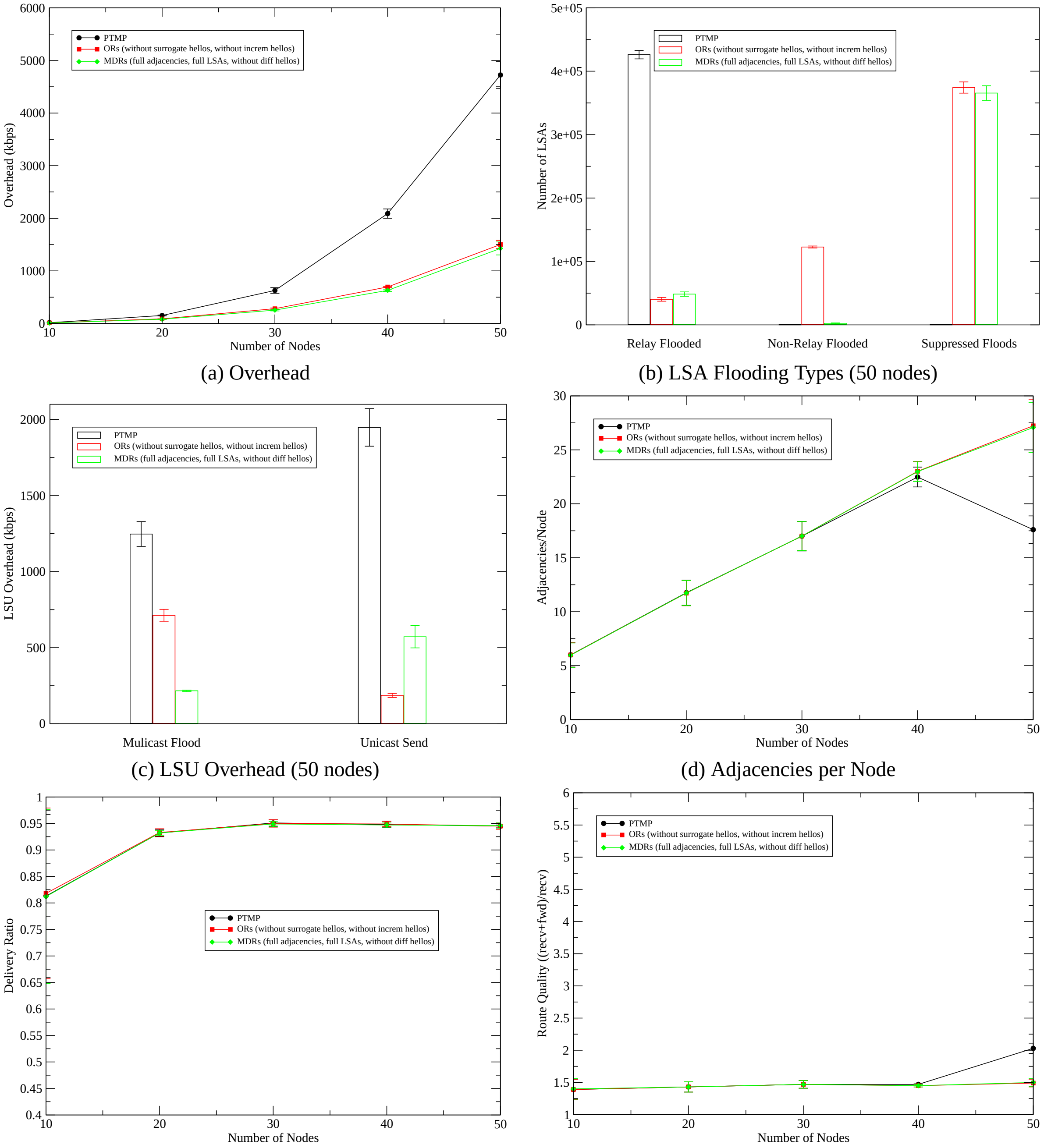
(c) LSU Overhead (50 nodes)

(d) Adjacencies per Node

(e) Delivery Ratio

(f) Route Quality

Figure 4.3: Low neighbor change rate and sparse neighborhood.



(a) Overhead

(b) LSA Flooding Types (50 nodes)

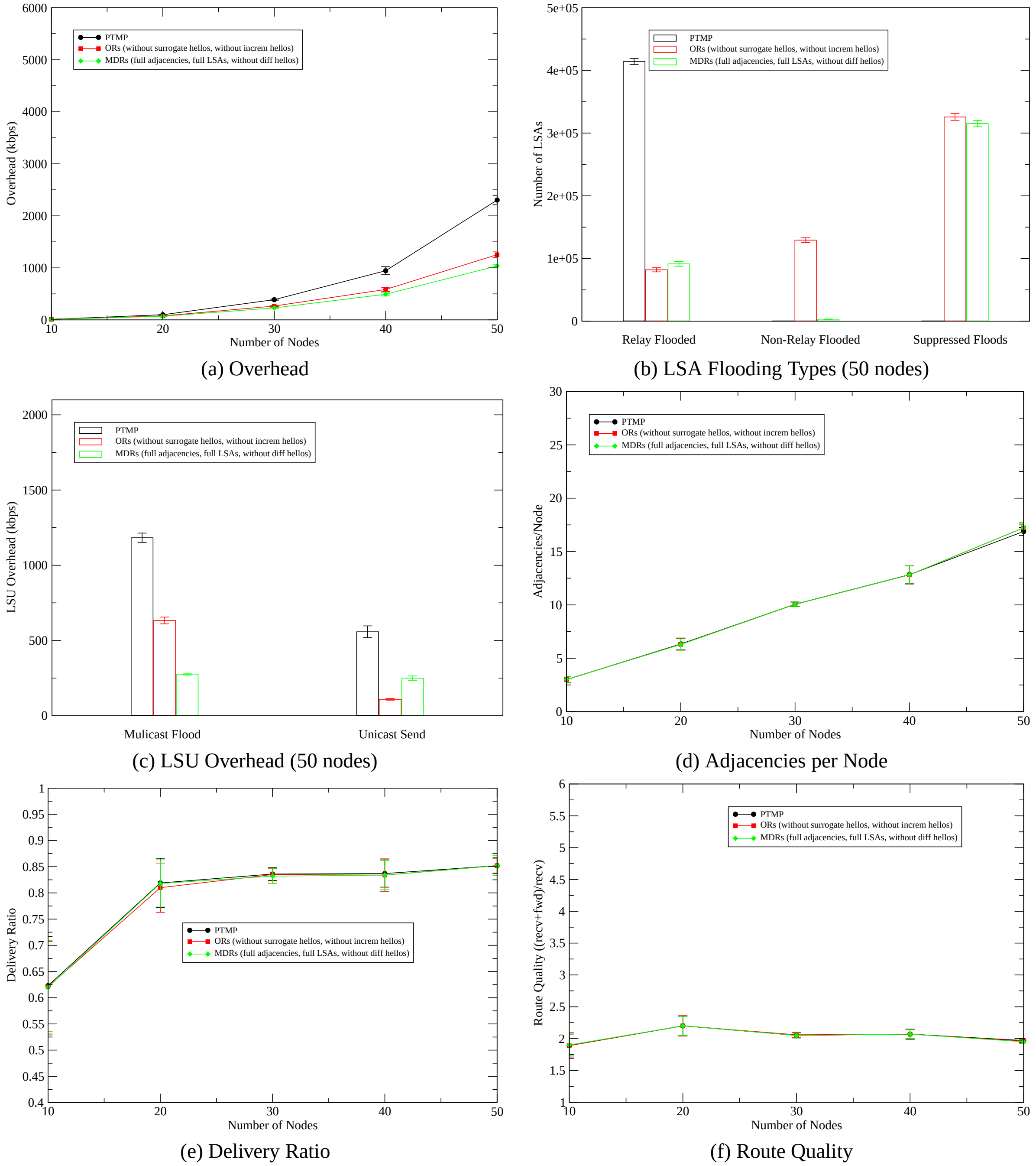
(c) LSU Overhead (50 nodes)

(d) Adjacencies per Node

(e) Delivery Ratio

(f) Route Quality

Figure 4.4: Low neighbor change rate and dense neighborhood.



(a) Overhead

(b) LSA Flooding Types (50 nodes)

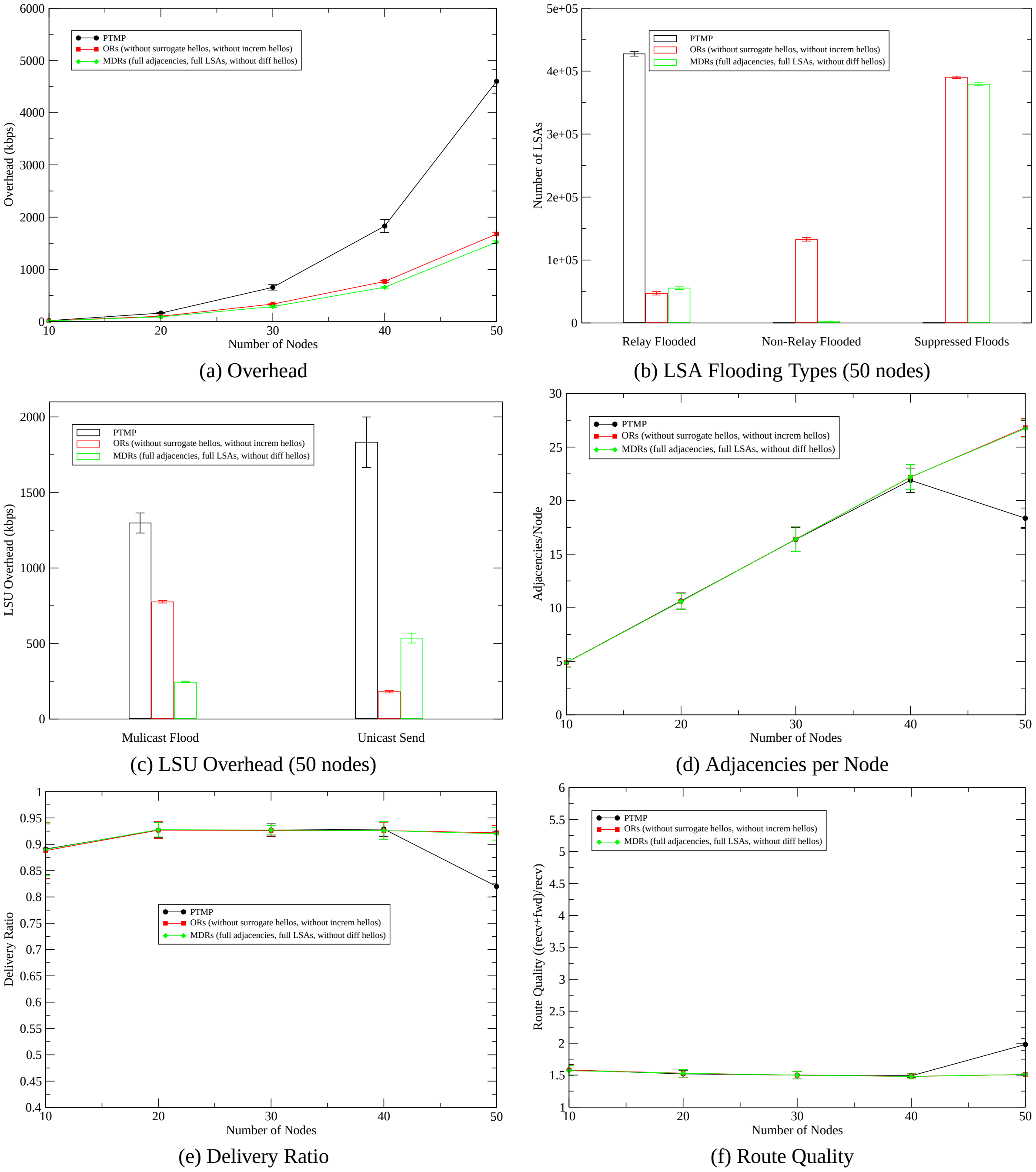
(c) LSU Overhead (50 nodes)

(d) Adjacencies per Node

(e) Delivery Ratio

(f) Route Quality

Figure 4.5: High neighbor change rate and sparse neighborhood.



(a) Overhead

(b) LSA Flooding Types (50 nodes)

(c) LSU Overhead (50 nodes)

(d) Adjacencies per Node

(e) Delivery Ratio

(f) Route Quality

Figure 4.6: High neighbor change rate and dense neighborhood.

4.4 Impact of Topology Reduction with MDRs

4.4.1 Reduced Adjacencies

An argument made in Ogier's draft is that the MDR-based algorithm lends itself to techniques that reduce the number of adjacencies in the network, along the lines of how the Designated Router improves scalability for broadcast networks. We experimented with several variants of this topology reduction. We did not use differential hellos in this comparison.

The first set of graphs can be seen in Figure 4.7. Three curves are plotted: (i) the MDR case with full topology (full adjacencies) and full LSAs (to compare with the Overlapping Relays case above in Figure 4.2), (ii) MDR with bi-connected adjacencies and (B)MDR full LSAs, and (iii) MDR with uni-connected adjacencies and (B)MDR full LSAs. The difference between full LSAs and (B)MDR full LSAs is discussed further below; in the (B)MDR case, each (B)MDR originates a full LSA, and each MDR_OTHER originates minimal LSAs.

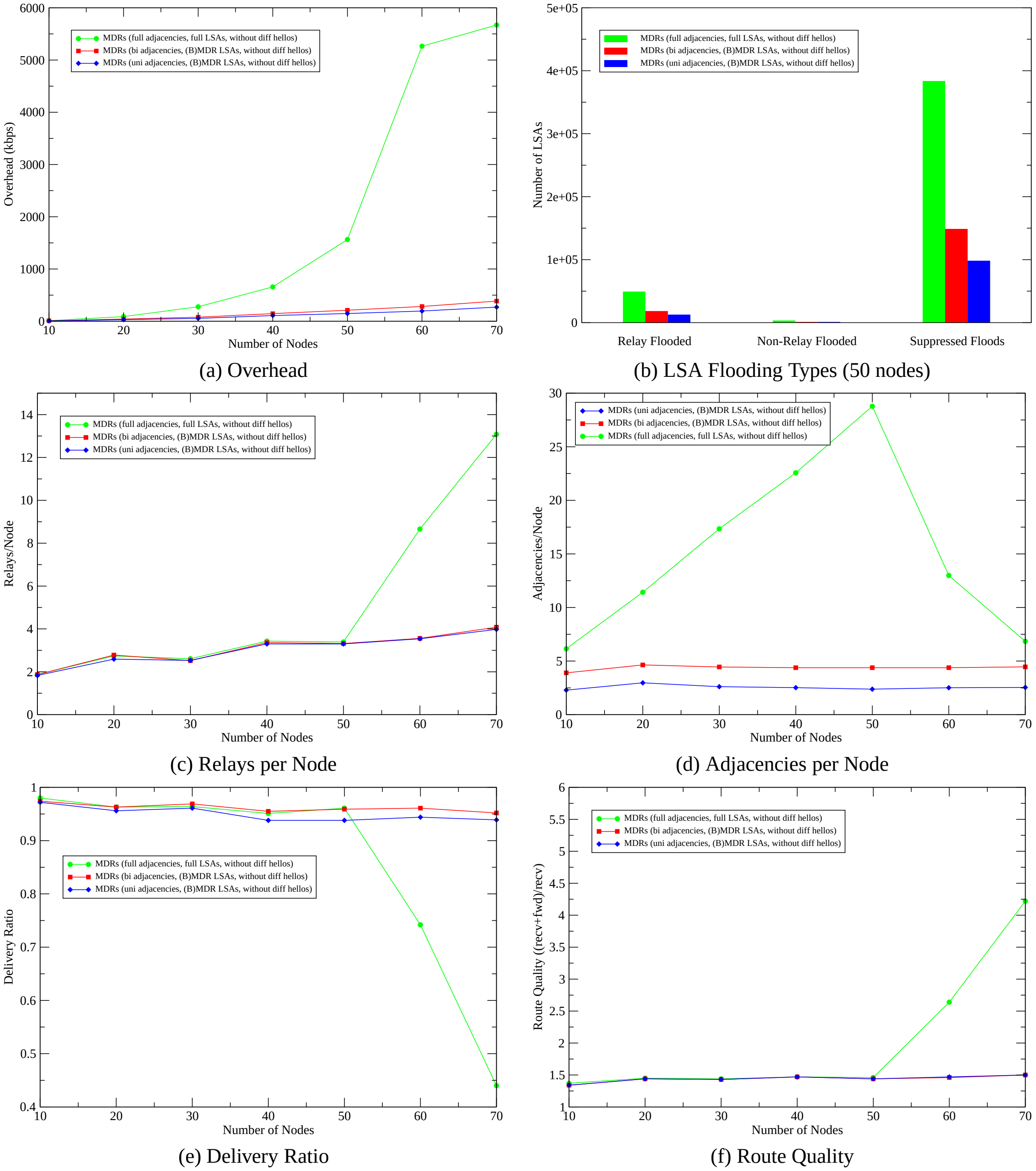
Section 5 of Ogier's draft describes how the topology reduction can be accomplished. The bi-connected case allows MDR_OTHER routers to form two adjacencies with MDRs and/or BMDRs, and the (B)MDR routers form a bi-connected backbone with their (B)MDR neighbors. In the uni-connected case, only a single adjacency is used, to reduce overhead.

In Figure 4.7(a), it should be clear that topology reduction provides substantial improvement in overhead reduction, when compared with full topology optimized flooding alone. When compared with Point-to-Multipoint (Figure 4.2), the savings are substantial. For example, the Point-to-Multipoint overhead for 50 nodes was 5 Mb/s, for fully connected MDRs or Overlapping Relays was roughly 1.5 Mb/s, and for bi-connected MDRs, the overhead is roughly 200 Kb/s. The overall scaling trend (overhead scaling with numbers of nodes) is also much improved; not quite linear, but much closer. Figure 4.7(d) underscores why this is the case; the number of adjacencies per node is relatively flat as the number of nodes increases, using reduced topology.

Figures 4.7(e) and (f) show that the forwarding performance is not significantly affected by the reduced topology, although there is a forwarding performance cost to using uni-connected adjacencies; lower overhead is traded for slightly reduced fowarding performance.

Note: To reproduce these results, execute the following script, but with different values for parameters AdjConnectivity and LSAFullness (see the usage information for details). Also, you may want to change the "tag" variable for each run to avoid the data results overwriting one another:

```
./comparative_mdr.pl
```



(a) Overhead

(b) LSA Flooding Types (50 nodes)

(c) Relays per Node

(d) Adjacencies per Node

(e) Delivery Ratio

(f) Route Quality

Figure 4.7: Topology Reduction comparative results: uni-connected vs. bi-connected adjacencies

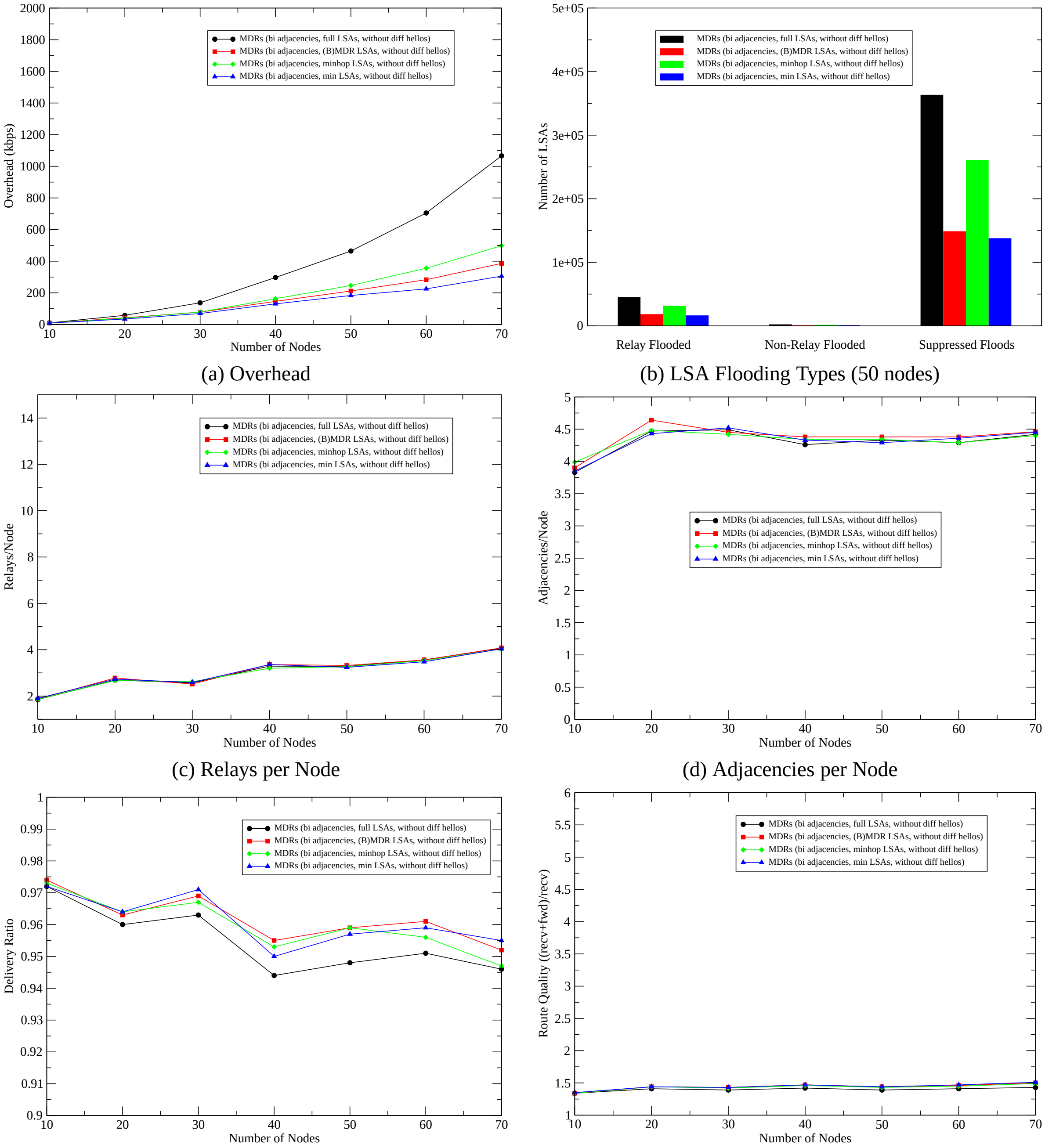
4.4.2 Routable Neighbors

A final set of variations to Ogier's algorithm are examined in Figure 4.8. These involve the differences due to partial or full topology LSAs (more details can be found in D. In the discussion below, a routable neighbor is a neighbor in state full or a neighor in TwoWay or greater that is also reachable in the SPT. Briefly, we explored the following four variations in LSA fullness, using bi-connected adjacencies (from above) in each case:

1. **"min LSAs"**: Each router advertises only its adjacencies in LSAs. Note that with bi-connected adjacency configuration, not all TwoWay neighbors become adjacent.
2. **"minhop LSAs"**: Each router includes in its LSA the minimum set of neighbors necessary to provide a minimum-cost path between each pair of neighbors that are not themselves neighbors.
3. **"(B)MDR full LSAs"**: Each MDR/BMDR originates a full LSA including all adjacencies and all "routable" neighbors, and each MDR_OTHER router originates min LSAs.
4. **"Full"**: Each router originates a full LSA containing all adjacent and "routable" (i.e., synchronized) neighbors.

The results show that, by trimming the amount of information contained in LSAs, the overhead can be reduced further (perhaps another 50% or so) than the "full" LSA case. Figure 4.8(e-f) indicates that there is a performance benefit to the delivery ratio when using partial topology LSAs, with a very slight performance degradation in route quality. The improvement in delivery ratio due to the reduction in topology information advertised is counterintuitive and unexplained at the time of this writing.

Note: To reproduce these results, execute the following script, but with different values for parameters AdjConnectivity and LSAFullness (see the usage information for details). Also, you may want to change the "tag" variable for each run to avoid the data results overwriting one another:
./comparative_mdr.pl



(a) Overhead

(b) LSA Flooding Types (50 nodes)

(c) Relays per Node

(d) Adjacencies per Node

(e) Delivery Ratio

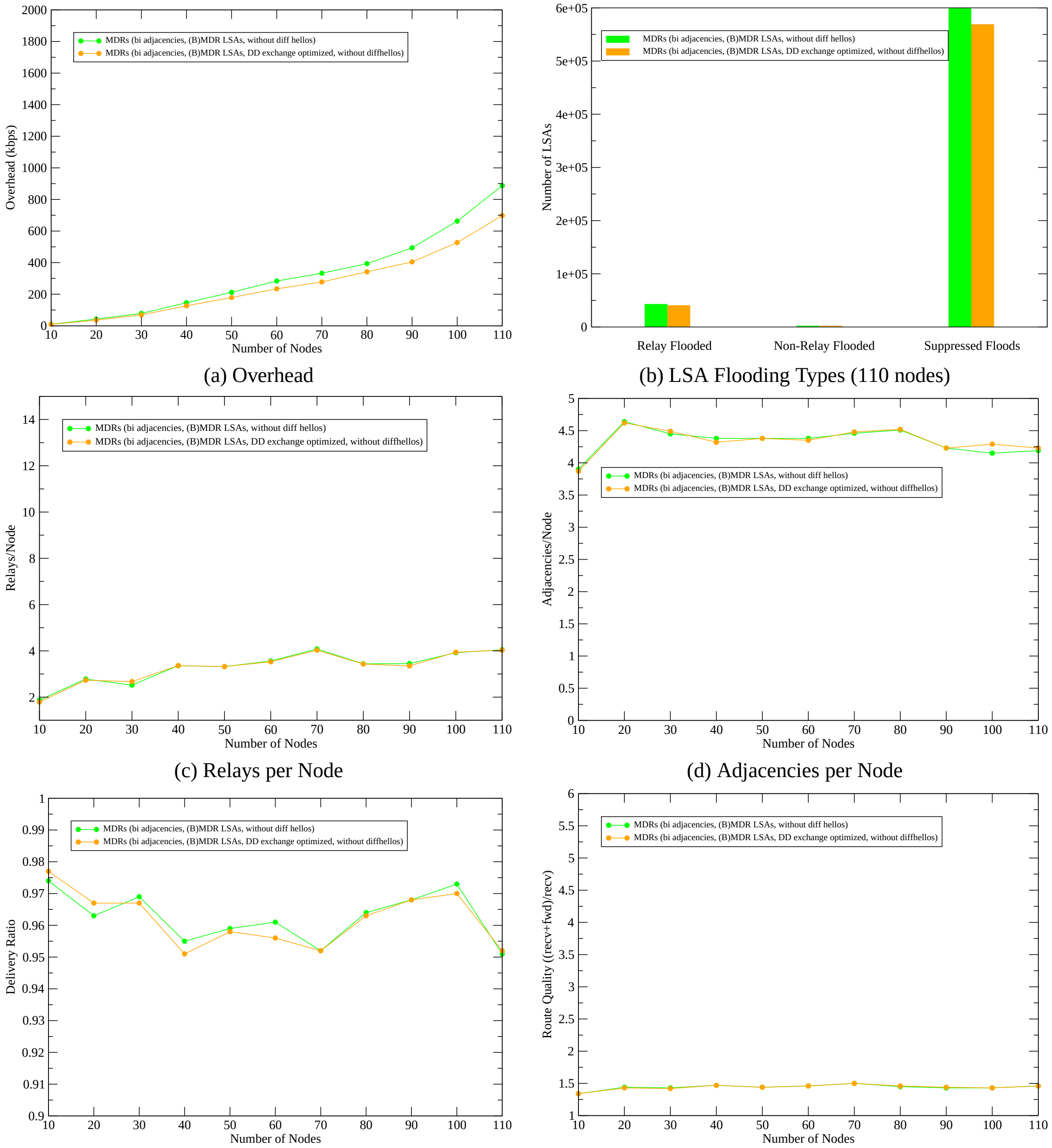
(f) Route Quality

Figure 4.8: Topology Reduction comparative results: variations on partial topology LSAs

4.4.3 Scaling of Partial Topology/Adjacency MDRs

In this section, we show the scaling properties of MDRs with biconnected adjacencies and (B)MDR full LSAs. We also show the performance of an optimzation in the database exchange process, in which a router (either the master or slave) does not include in its DD packets an LSA header if it knows the neighbor has the same or newer instance of the LSA (based on DD packets received from the neighbor). This is done simply by removing the LSA from the summary list when the same or newer LSA is received in a DD packet.

In this scenario, we were able to scale up to 110 nodes before running out of memory on our simulation machine. The database exchange optimization reduced the DD overhead by about 50% without causing degradation in the other metrics.



(a) Overhead

(b) LSA Flooding Types (110 nodes)

(c) Relays per Node

(d) Adjacencies per Node

(e) Delivery Ratio

(f) Route Quality

Figure 4.9: MDR Scaling from 0 to 110 routers.

4.5 Impact of Topology Reduction with ORs

Around the time of publication of this report, Cisco published an Internet Draft outlining a method called "Smart Peering" to reduce the number of adjacencies in a MANET network [Roy05]. The goal of this method is to reduce the number of adjacencies in dense networks without compromising routing performance (similar to the goal of the strategy discussed above in Section 4.4). Unlike the MDR technique, however, "Smart Peering" does not try to leverage the underlying flooding structure to guide its decisions as to whether to become adjacent– instead, decisions are made based on reachability of the node in the Shortest Path Tree (SPT) and possibly other heuristics on the distance to the node or the number of redundant paths in the existing SPT.

We have implemented a basic version of this draft and have obtained preliminary results.

4.5.1 Smart Peering

First we describe our implementation of Smart Peering. The algorithm allows suppression of the formation of an OSPF adjacency to "routable neighbors." A routable neighbor is a neighbor in state full or a neighbor in TwoWay or greater that is also reachable in the SPT.

Upon receiving the neighbor event TwoWayReceived, a neighbor decides whether it should be become adjacent with a neighbor by checking for a route to the neighbor in the SPT. If a route exists, an adjacency is not established. If a route does not exist then an adjacency is formed. This method forms close to the fewest number of possible adjacencies, but it can result in long flooding and routing paths. In future refinements, a route could be forced to be below a certain cost or a certain number of available routes could be enforced before deciding to suppress the possible adjacency, as described in Section 3.2 of [Roy05]. This would trade off overhead (more adjacencies) to improve forwarding performance.

If one neighbor decides an adjacency should be established and sends a Database Description (DD) packet then the receiving neighbor always chooses to become adjacent. So, a router does not discard DD packets when in state TwoWay.

Whenever a new router-LSA is installed in the Link State Database, each TwoWay neighbor must be rechecked for routability. If a route is not found, an adjacency is established.

Since the 2-hop neighborhood is determined with the router-LSA, the relay set will be different than with full topology routing. In general, the relay set will be larger because the OSPF neighborhood is less dense.

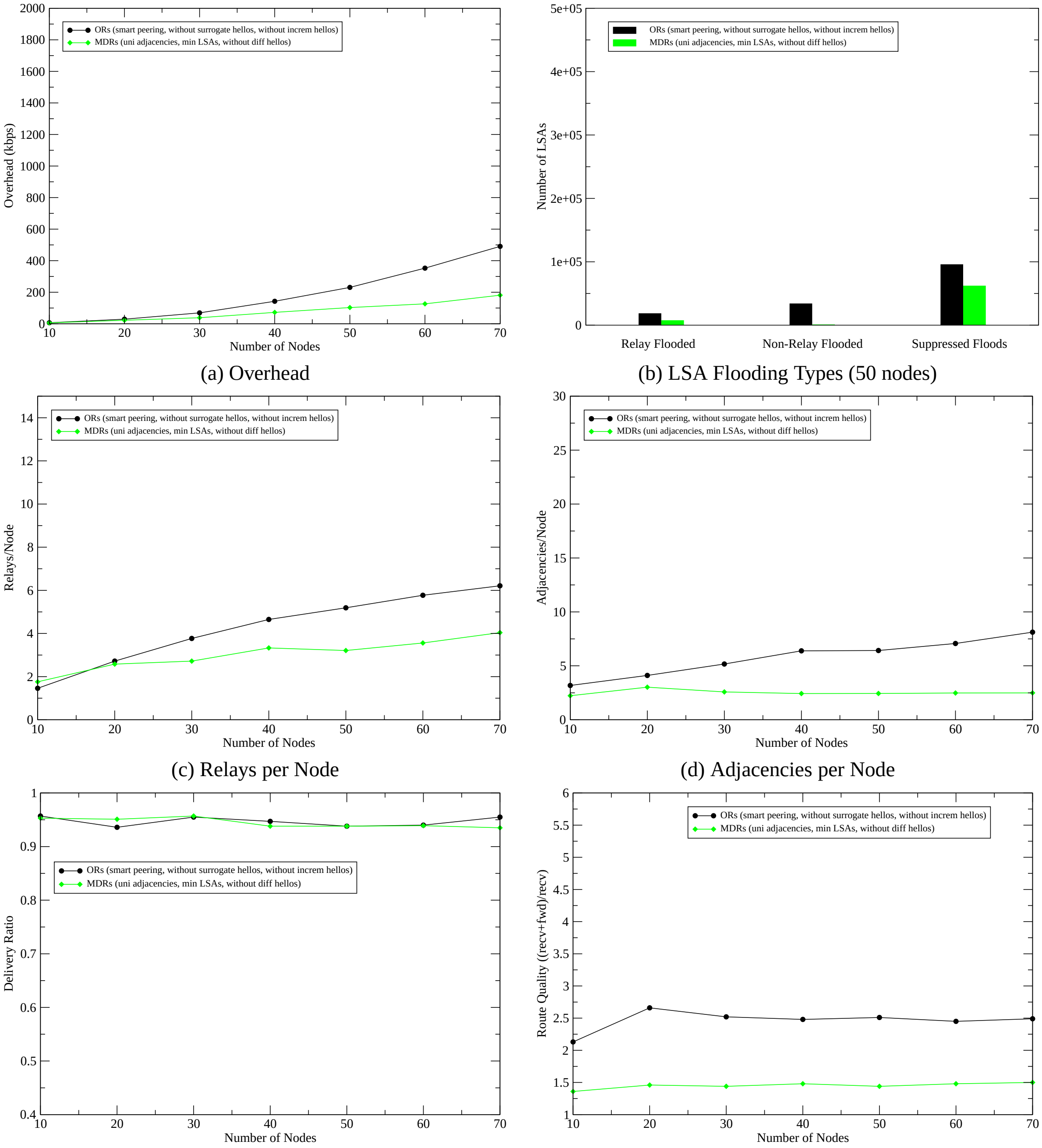
Note: To reproduce the following MDR results, execute the following script, with uniconnected adjacencies and min LSAs.
./comparative_mdr.pl
To reproduce the following OR results run the following script and set smart_peering in ospf6_interface.c to true (and recompile the code). ./comparative_mpr.pl

In Figure 4.10, Smart Peering ORs are compared with the variant of the MDR algorithm that uses uniconnected adjacencies and min LSAs, because Smart Peering is designed to have the near minimum number of adjacencies and only full adjacencies are advertised in router-LSAs.

The results for this scenario suggest, based on our preliminary implementation without heuristics for improving the routing stretch, that ORs with Smart Peering may generate more overhead than similarly configured MDRs. We hypothesize that this is a result of the number of additional adjacencies that are formed with ORs. Unlike the MDRs, the ORs do not prune adjacencies as they become redundant. Pruning adjacencies with ORs would require locally removing an adjacency with a neighbor and then retesting the routability of all neighbors, which we did not implement in the timeframe of this study. In general, it seems that it may be difficult to avoid revisiting these previous decisions to form adjacencies each time the local topology changes, which could incur substantial computational cost.

The two approaches yield approximately the same delivery ratio. In future work, the delivery ratio should be tested in different mobility and density scenarios to confirm this result.

Finally, the route quality is much worse for ORs because adjacencies are formed purely based on routablity. There was no heuristic used to ensure the routable paths were below a certain cost. Therefore, a particularly circuitous route to a neighbor could result in suppression of an adjacency that would provide much lower cost. This could potentially be alleviated if routable neighbors were included in router-LSAs; a proposal suggested in Section 3.3 of [Roy05] and explored in the next subsection.



(a) Overhead

(b) LSA Flooding Types (50 nodes)

(c) Relays per Node

(d) Adjacencies per Node

(e) Delivery Ratio

(f) Route Quality

Figure 4.10: Smart Peering

4.5.2 Routable Neighbors

It is possible to include routable (synchronized) neighbors as point-to-point links in router-LSAs, making these links available in the data plane [Roy05]. In our implementation of this method, when router-LSAs are built they include routable neighbors that are either in state full or in TwoWay or greater. A router-LSA is reoriginated when any of these links change.

One difficulty with this method is that adjacencies are formed based on whether neighors are routable, while unsynchronized links are included in router-LSAs. Therefore, there is a circular dependency with routable neighbors where a neighbor may only be routable over a path that does not consist of connected adjacencies. An implication is that it is not guaranteed that the set of adjacencies forms a connected subgraph, and the network may become unsynchronized, leading to increased possibility of routing loops. A possible way to fix this issue is to label links that are not synchronized and not include these when calculating whether a neighbor is routable (computing a "shadow" SPT, based only on adjacencies, that is used for adjacency decisions, while using the normal SPT for data plane forwarding). Such a technique may involve changes to the router-LSA format or introduction of new LSAs. We did not implement this method.

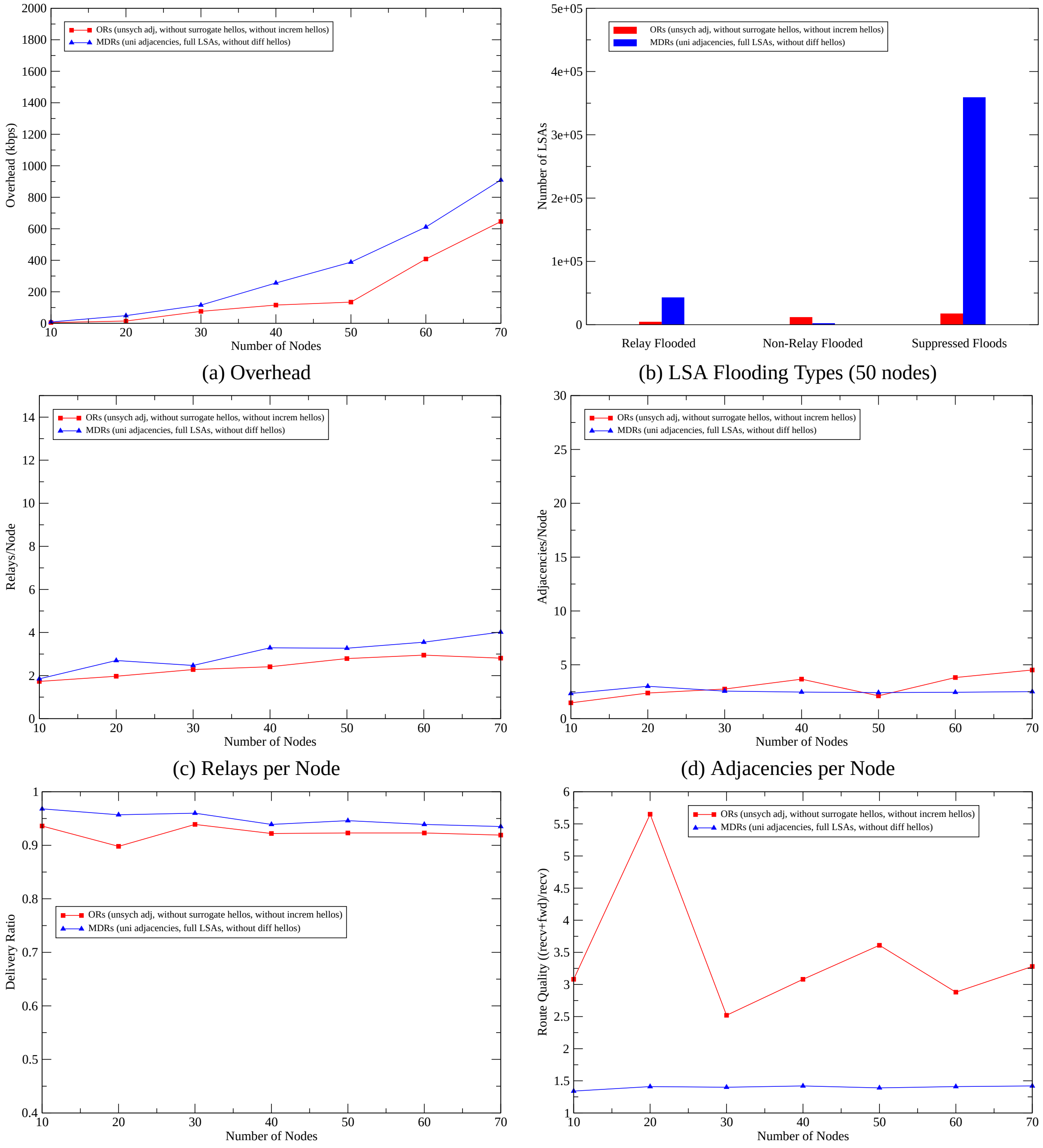
By including routable neighbors in router-LSAs, we expected that the routing stretch would be minimized and the relay set would be about the same as in the full topology version because the 2-hop neighborhood should be about the same.

Note: To reproduce the following MDR results, execute the following script, with uniconnected adjacencies and full LSAs.
./comparative_mdr.pl
To reproduce the following OR results run the following script and set smart_peering and unsynch_adj in ospf6_interface.c to true. ./comparative_mpr.pl

In Figure 4.11, Smart Peering ORs with routable neighbors are compared with the variant of MDRs using uniconnected adjacencies and full LSAs, because the ORs are designed to have the near minimum number of adjacencies and all routable neighbors are advertised in router-LSAs.

Note that in Figure 4.11(f), the routing stretch metric did not improve but instead substantially degraded. We investigated this unanticipated result and found that the average routing stretch was being distorted by a large number of packets encountering routing loops. Furthermore, our count of the average number of LSAs out of synchronization when two routers did form an adjacency was much larger in this case, and the number of adjacencies was much smaller (with many fewer database synchronization events taking place). All of this evidence suggests that the concern about this approach leading to synchronization problems in the network may be well founded. As a result, the delivery ratio and route quality are impaired.

If only adjacent paths were used for routability checks, then we would expect the overhead to be greater than the Smart Peering case, but the delivery ratio and route quality should be improved. We should also see approximately the same relay set as found in the full topology case. However, as mentioned above, we did not try to iterate this design because it requires more significant changes to the implementation, and may not be backward compatible if LSA format changes are required.



(a) Overhead

(b) LSA Flooding Types (50 nodes)

(c) Relays per Node

(d) Adjacencies per Node

(e) Delivery Ratio

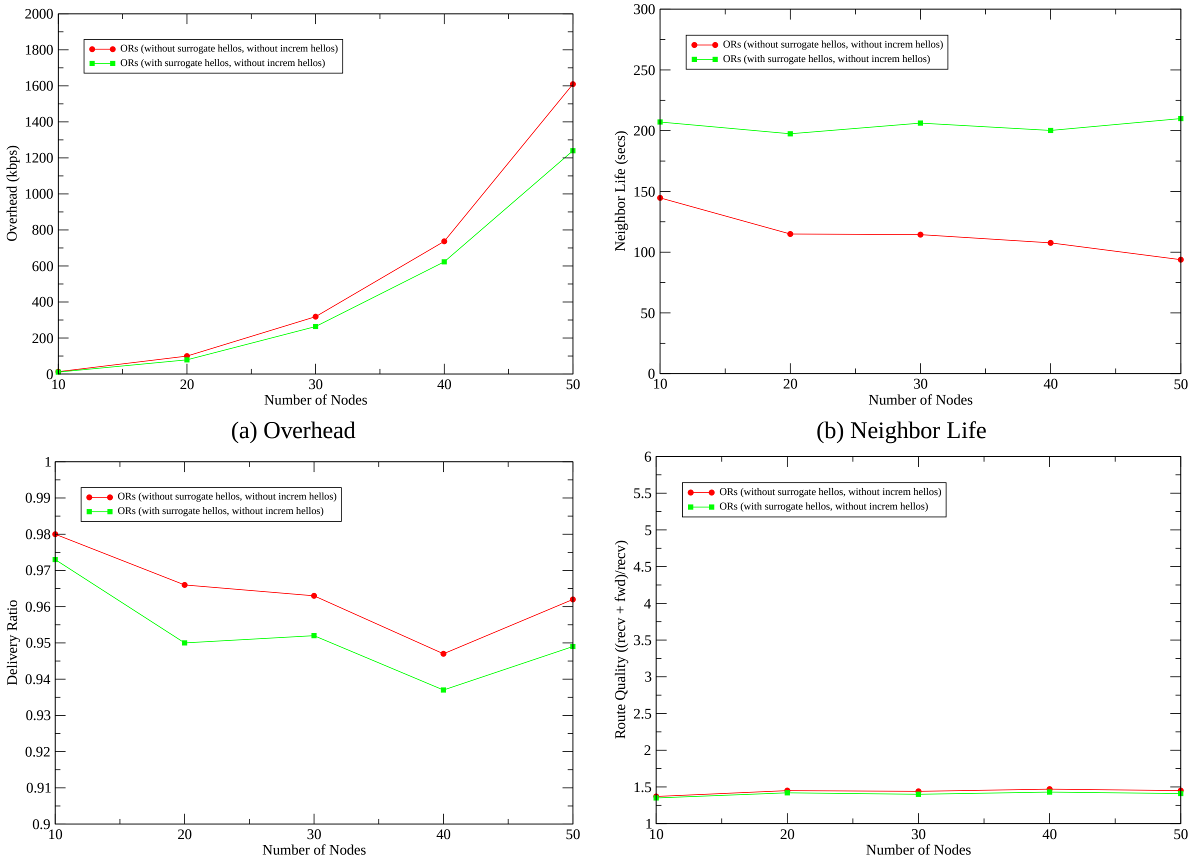
(f) Route Quality

Figure 4.11: Including Routable Neighbors in the Smart Peering Approach

4.6 Impact of ACK as Surrogate for Hello

Above, we noted that our comparative results for Overlapping Relays did not use the recommendation (Section 3.4.9, item 6 of [Cha05]) to use ACKs as surrogates for Hellos. Figure 4.12 shows some initial results that illustrate some difference in performance that occurs when enabling this feature. The results for 50 or fewer nodes are the most meaningful since the network is not saturated with OSPF overhead. Figure 4.12(a) illustrates that the use of ACKs as surrogate Hellos can actually drop the overhead. The reason for this behavior is shown in Figure 4.12(b), where it can be seen that the average lifetime of a neighbor is much higher when ACKs are used as surrogate Hellos, suggesting that such treatment has the effect of keeping neighbor relationships up for longer periods of time when they would normally fail due to loss of too many Hellos. This leads to less overhead (less topology change), but the results in Figure 4.12(c) suggest that the reduction in overhead comes at the expense of slightly worse data delivery performance.

Note: To reproduce these results, execute the following script:
./comparative_mpr.pl
However, to use the surrogate hello response, you must compile
the macro OSPF6_MANET_MPR_SH.



(a) Overhead

(b) Neighbor Life

(c) Delivery Ratio

(d) Route Quality

Figure 4.12: Surrogate Ack comparative results

Chapter 5

Summary

Our simulation results and experience with implementing both the Overlapping Relays (ORs) and MANET Designated Router (MDR) proposals has led us to the following conclusions.

First, we summarize the comparison of the core approaches used to reduce flooding overhead.

- When compared with Point-to-Multipoint interface performance, for a given number of nodes, both optimized flooding algorithms substantially reduce the amount of overhead due to OSPF operation in these networks (Figure 4.2(a)), without compromising the routing/forwarding performance (Figure 4.2(e-f)). We observed that the forwarding performance actually was better in those cases where legacy OSPF was incurring large amounts of overhead (such as running Point-to-Multipoint with 50 nodes in our scenario).

- Although results are scenario dependent, and overall magnitude of overhead reduction depends on the mobility and density of the network, we can say that for the type of scenario tested above, the core optimized flooding algorithms reduced overhead by a factor of three for 50-node networks. For small networks with this level of mobility (approximately one adjacency change every three seconds), there is improvement but it is not that significant since legacy OSPF is tolerable at that mobility level.

- In comparative testing, both implementations achieved comparable levels of overhead reduction, although they went about it with different strategies. Overlapping Relays relies on the use of non-active relays to proactively deliver flooded copies of the LSA to a target neighbor, in the absence of an ACK reception, before the original active relay's retransmission timer expires. This causes a larger number of backup flooded LSAs. In contrast, MANET Designated Routers relies more on retransmissions along adjacencies, with Backup Designated Routers reflooding LSAs only if they do not hear the primary Designated Router sending the LSA (i.e., the backup procedure does not listen for ACKs of the target node but instead for the floods of the non-backup flooding node).

- The size of the relay sets (normalized to number of relays used per node) was consistently smaller for the Overlapping Relays case than for the MPN algorithm from Ogier with `h1=3,h2=INF` (e.g., Figure 4.2(c)). However, it is questionable whether this statistic has as much meaning in a protocol with reliable flooding, such as OSPF. This is because the savings on the initial flooding overhead may be offset by the need to retransmit the LSA if the relay set is too sparse and some transmissions are lost. We believe it is more meaningful to compare the resulting overhead from all transmissions rather than just looking at the initial flood.

- We examined the sensitivity of our results to differences in node mobility and node density (Figures 4.3-4.6). Our results were consistent between these scenarios. There was a greater difference (growth) in overhead when moving from low to high density networks, under constant mobility, than when moving from low to high mobility networks, under constant density. This result suggests that approaches to cope with dense mobile networks may yield additional overhead reductions, and leads to our next observation.

While the above results, considering only the flooding optimizations, are promising, they also indicate a major limitation. Figure 4.2(a) suggests that the scaling trend for dense networks is similar to that of Point-to-Multipoint. Specifically, the overhead still scales faster than the square of the number of nodes; in Figure 4.2(a), the point at which the number of nodes practically exceeded the network capacity was roughly 50 nodes for Point-to-Multipoint but only

60 nodes for the optimized flooding. Intuitively, this makes sense, because in a dense mobile network, as the number of nodes increases, the number of adjacencies increase, causing the router-LSAs to grow correspondingly, while simultaneously increasing the number of routers affected by each router's movement (i.e., each adjacency constructed is a potential candidate for a future topology change when the pair of routers moves apart).

Ogier's MDR proposal takes an additional step to leverage the MDRs to reduce the number of adjacencies and the advertised topology. There appears to be a substantial benefit from minimizing the number of adjacencies in dense MANETs. The approaches outlined by Ogier's draft yielded significant decrease in overhead when compared even with the MANET-optimized flooding approaches of both Overlapping Relays and full-topology MANET Designated Routers, both in terms of absolute overhead reduction for a given number of nodes (Figure 4.7(a)), and the scaling trend (Figure 4.9(a)). The resulting forwarding performance is not substantially affected by such reductions (Figure 4.7(e-f)), although our results suggest that there is a tradeoff between how aggressive one tries to decrease the overhead and how much the forwarding performance is affected.

- We studied two variants for reducing the number of adjacencies in the MDR backbone and also between non-backbone routers and the backbone (Figure 4.7). Uni-connected adjacencies resulted in lower overhead (fewer adjacencies to maintain or change over time), but at a cost of reduced robustness (reduced data delivery radio).

- We also explored the performance of different variants of partial and full topology LSAs (Figure 4.8), and found that the overhead in the partial topology approaches can be further reduced from that of full topology LSAs, at the expense of a slight degradation in routing stretch (number of forwarding hops required).

A natural question is whether the above improvements shown in the MDR case can translate to the Overlapping Relays case. The analogy to this strategy would be to limit adjacencies between MPRs and their selector nodes. However, because MPRs and selectors are not symmetric (unlike MDR and MDR-other nodes), we hypothesize that such a strategy will result in many more adjacencies. We also found that a recent proposal by Cisco to reduce adjacencies without leveraging the underlying flooding backbone resulted in either increased amount of overhead or a poorly synchronized network. It remains to be seen whether such a strategy for topology reduction in the Overlapping Relays framework can approach the performance demonstrated by MDRs.

We have not studied in depth the following two design aspects of the two proposals:

- both proposals describe an Incremental or Differential Hello mechanism. The goals of both mechanisms are the same but the design is different. We have not focused on this mechanism because preliminary results indicated that the overhead due to full Hellos is secondary to the overhead due to flooding and topology. However, such mechanisms may provide additional scaling benefits especially when reducing the HelloInterval to achieve more response to mobility changes.

- Another difference between Overlapping Relays and MANET Designated Routers is how the two-hop neighborhood information is propagated; in router-LSAs or Hellos, respectively. We did not yet study the implications of this difference. Use of LSAs may provide more opportunities to suppress Hellos based on cross-layer optimizations with layer-2 protocols that maintain neighbor lists and bidirectional forwarding checks, and may leverage LSA dampening mechanisms already implemented in OSPF routers. However, use of LSAs to obtain 2-hop information can cause circular dependency problems if the 2-hop information is also used to make adjacency forming decisions.

We explored a variation in the Overlapping Relays case: whether an ACK can act as a surrogate for a Hello or not. We found that using ACKs in such a manner had the effect of reducing the topology changes (and hence the overhead), because the average lifetime of an adjacency more than doubled (Figure 4.12(b)). However, this reduced overhead came at the cost of reduced forwarding performance (data delivery ratio; Figure 4.12(c)), since it appears that using ACKs in this manner has the effect of "propping up" weak neighbor relationships (links) that may not be very reliable for data transfer.

Although we only have initial quagga implementations of these algorithms, it seems that the addition of a MANET interface type does not substantially change the structure or operation of an OSPF implementation. We have not explored how MANET interfaces might interact with other OSPF features (e.g., demand circuit operation)– the complexity might be increased, but we do not have any basis to believe that such interactions would cause fundamental problems or destabilization of the quagga OSPF implementation.

Finally, for future work, we believe that the following items may yield additional substantial improvements:

- **neighbor stability:** In a wireless environment, not all neighbors are equal. Some neighbors are more stable than others, and some neighbors have better packet transmission performance between them. The cost of picking bad neighbors for relays or adjacencies is additional instability in the routing. It seems that the next natural optimization to consider, beyond the adjacency reductions described by Ogier, are heuristics to pick good neighbors and relays, and mechanisms to assign (possibly dynamic) link costs based on link attributes.
- **interaction with OSPF hierarchy mechanisms:** One downside to performing MANET routing at layer-3 is that router-LSAs are flooded with area scope. For large OSPF areas, this could cause a lot of overhead being exported from the MANET subnet into the rest of the area.

Chapter 6

Acknowledgments

We are grateful to Richard Ogier, who provided substantial technical review of our design and results and who contributed core aspects of the MDR implementation.

We also acknowledge the helpful collaborations with a number of researchers, including Joe Macker of NRL, Thomas Clausen (formerly of INRIA), Emmanuel Baccelli and Philippe Jacquet (INRIA), Abhay Roy and Madhavi Chandra (Cisco), and Jae Kim (Boeing). Padma Pillay-Esnault and Acee Lindem (Cisco) also contributed helpful suggestions as part of the IETF OSPF MANET design team discussions.

Jeff Meegan of Boeing Phantom Works assisted in the initial port of quagga code to the *GTNets* environment, and investigated a number of techniques for improving run-time performance.

This research was funded by the Office of Naval Research under contract N00014-04-C-0003. The simulator developed for this project is an extension of two open-source software projects to which we are also indebted: the Georgia Tech Network Simulator and the Quagga Routing Suite.

Bibliography

[Bak03] F. Baker. Problem Statement for OSPF Extensions for Mobile Ad Hoc Routing. Internet-Draft (work in progress) draft-baker-manet-ospf-problem-statement-00, IETF, August 2003.

[CFM99] R. Coltun, D. Ferguson, and J. Moy. OSPF for IPv6. Request for Comments 2740, IETF, December 1999.

[Cha05] M. Chandra. Extensions to OSPF to Support Mobile Ad Hoc Networking. Internet-Draft (work in progress) draft-chandra-ospf-manet-ext-03, IETF, April 2005.

[CJ03] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). Request for Comments 3626, IETF, October 2003.

[CM99] S. Corson and J. Macker. Mobile Ad Hoc Networking (MANET): Routing Protocol Performance Issues and Evauluation Considerations. Request for Comments 2501, IETF, January 1999.

[HP04] P. Spagnolo T. Henderson and G. Pei. Design Considerations for a Wireless OSPF Interface. Internet-Draft (work in progress) draft-spagnolo-manet-ospf-design-00, IETF, April 2004.

[HSK03] T. Henderson, P. Spagnolo, and J. Kim. A Wireless Interface Type for OSPF. In *Proceedings – IEEE Military Communications Conference MILCOM*, volume 2, pages 1256 – 1261. IEEE, October 2003.

[NY05] A. Zinin B. Friedman A. Roy L. Nguyen and D. Yeung. OSPF Link-local Signaling. Internet-Draft (work in progress) draft-nguyen-ospf-lls-05, IETF, March 2005.

[OS05] R. Ogier and P. Spagnolo. MANET Extension of OSPF using CDS Flooding. Internet-Draft (work in progress) draft-ogier-manet-ospf-extension-04, IETF, July 2005.

[Ril03] George F. Riley. Large-scale Network Simulations with GTNetS. In David M. Ferrin and Douglas J. Morrice, editors, *Winter Simulation Conference*, pages 676–684. ACM, 2003.

[Roy05] A. Roy. Adjacency Reduction in OSPF using SPT Reachability. Internet-Draft (work in progress) draft-roy-ospf-smart-peering-04, IETF, July 2005.

Appendix A

User's Guide

This appendix describes the implementation of MANET extensions to OSPFv3 as an extension to the quagga[1] OSPFv3 routing daemon and the Georgia Tech Network Simulator (*GTNets*).[2] Figure A.1 illustrates how the same quagga software is supported in an actual quagga implementation and also wrapped within the *GTNets* simulation framework; this enables moving between implementation and simulation with the same code base, just by changing some compilation flags.

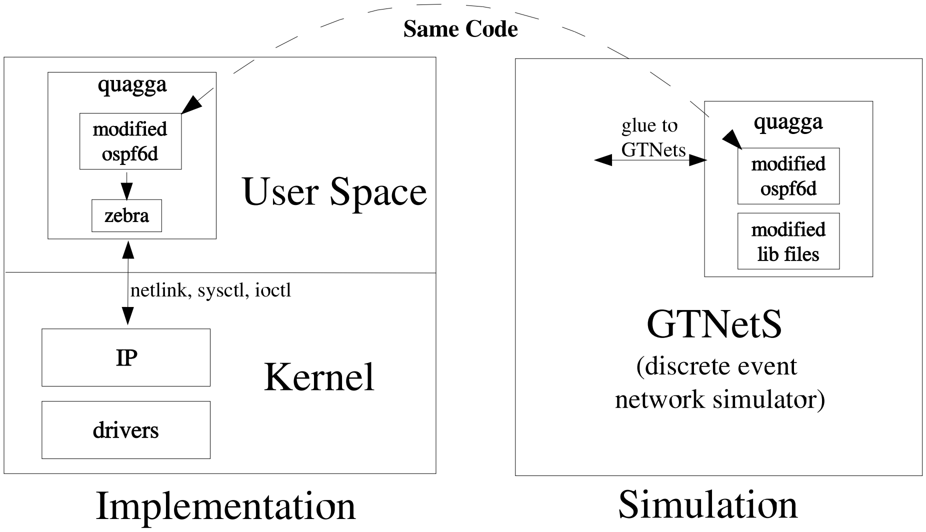


Figure A.1: Architecture of Simulation and Implementation

This packet-level, discrete-event simulator has the following features:

- OSPFv3 for IPv6 based on recent quagga software (version 0.98.4).
- Support for enabling proposals known as **Overlapping Relays**, contributed by Madhavi Chandra (editor) from a team of authors at Cisco Systems [Cha05], and a proposal known as **MANET Designated Routers**, contributed by Richard Ogier [OS05].
- (slightly modified) 802.11 and wired channel models
- detailed logging and tracing of simulation events, including all quagga daemon logging

[1] http://www.quagga.net

[2] http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS/

We have not removed any features of *GTNets*, which are described elsewhere in the *GTNets* documentation, or `quagga`, although we note that only OSPFv3 is supported in the simulation (the full `quagga` routing daemon is available if run as implementation code). Here, we focus on the types of OSPF MANET simulations that we can presently do with this simulator.

The next section describes the details for using this software in *simulation* mode. The following section describes how to use it as part of an actual quagga *implementation*.

A.1 Simulation

A.1.1 Building the Simulator

These instructions assume that you are using a recent distribution of Linux. Other *nix variants probably work also, but haven't been tested. [3]

There are two ways to build the simulator: optimized, and with debugging symbols. Optimized runs slightly faster. To build, cd into the top level *GTNets* directory, and follow the README. To build everything:

```
make clean
make depend
make all
```

Making "all" builds both an optimized and debugging (with debugging symbols) binary of each test file. The optimized runs a bit faster. There are other options in the Makefile to build subsets of the scripts. For example, to start compilation over and build only the optimized version of what is in the ietf directory:

```
make clean
make depend
```

`make` *target*`-`*type*

Table A.1 displays how various targets (example scripts, ietf scripts, or validation scripts) can be combined with the two options for build type (optimized or with debugging symbols). For example, to compile only the scripts (programs) in the validation directory, and only for optimized binaries, use `make valid-opt`.

Possible *targets*	Possible *types*
examples	opt
ietf	dbg
valid	

Table A.1: Build *GTNets* with a combination of the above target and type options; e.g., `make valid-opt`.

[3] *GTNets* has been compiled successfully on Linux with g++-2.96, g++-3.3, Microsoft Windows with MSVC-7.0, and Sun Solaris with SUNWS-CC version??. However, we have not tested anything except gcc-3.3.3 since we did the OSPF MANET extensions. We observed a compilation problem with the newer Fedora Core 3 distribution, which uses gcc-3.4– contact us if you need a patch to the file cpqtr.h to make that compiler work.

Qt animation option
A default requirement is the Qt libraries from Trolltech. These are typically included in Linux distributions if you use KDE. These are used if you want to look at animations (e.g. `testwirelessgrid1.cc` in the `EXAMPLES/` directory).
An alternative to configuring and installing a Qt distribution, if you do not care about animation, is to comment out the inclusion of the QT configuration script in the toplevel and sub directory makefiles.

```
include mk/qtcheck.mk
```

This will allow building without Qt.

Note: If you are using Fedora Core 2 or something very recent, you might have a multithreaded version of Qt installed. Even though Qt works, you don't have the right header files for *GTNets* compilation. In this case, we suggest the following:

- install the source package `qt-x11-free-3.3.3` in the directory `/usr/local/src`
- make a symbolic link `ln -s /usr/local/src/qt-x11-free-3.3.3 /usr/local/qt3`
- when you `make examples-dbg`, you need to pass in the QTDIR environment variable as such: `make examples-dbg QTDIR=/usr/local/qt3/`

A.1.2 Running the Simulator

Example simulation executables are stored in the EXAMPLES directory. Versions that are built with debugging symbols contain the suffix "-dbg", and for optimized, the suffix "-opt".

Executables that we have been using are stored in the following directories:

- `ietf/random_waypoint_manet/` and
- `ietf/link_matrix_manet`.

Try running an example script such as `ietf/random_waypoint_manet/random_waypoint_manet-opt`. It should leave a tracefile in a `random_waypoint_manet.tr` file, print summary statistics to a `random_waypoint_manet.stat` file, and print runtime progress to standard output.

The files in the `SRC/` directory are built as a library `libGTNetS-debug.{a,so}` or `libGTNetS-opt.{a,so}`. This library is linked to whatever test program you are using. The typical GNU debugger `gdb` can be used to debug if debugging symbols are present.

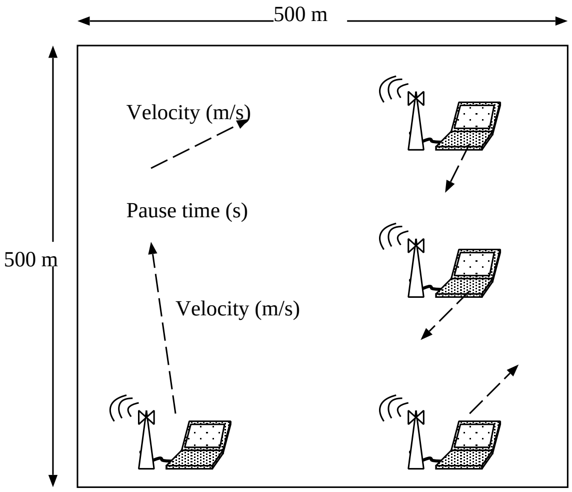
A.1.3 Configuring the Simulator

This chapter describes common command-line interface configuration options, and then walks through a sample program in Section A.1.3.

Compile-Time Options

The default Makefile in the top level directory contains the following compile-time options (the defaults enabled are shown below):

```
# To run OSPF6D, enable:
CFLAGS    += -pg -D$(OSNAME)
# The below defines provide general support for the MANET extensions
CFLAGS    += -DSIM -DETRACE -DETRACE_DEBUG
CFLAGS    += -DSIM_REPLACE_SCHEDULER -DREPEATABLE_MOBILITY -DETHERNET_BOEING
CFLAGS    += -DSIM_ETRACE_STAT -DOSPF6_JITTER
CFLAGS    += -DBOEING_ARGS
# To add TDMA USAP, add the following:
CFLAGS    += -DUSAP_ADDON
```



- variable number of nodes, fixed grid size
- velocities configured via script (2, 5, 10 m/s); pause time is 40 seconds (also configurable)
- 802.11b radios (described in Section 6.1.1) with range ~ 250 m
- no cross-layer optimizations (between router and radio)
- simulation runs for 30 minutes, after a 30 minute startup period where routers are started randomly, and no statistics are gathered
- light amount of UDP traffic sent between nodes, to assess routing performance
- OSPF retransmit and Hello intervals are 10 seconds; dead interval of 40 seconds
- Packet transmissions jittered uniformly between 0,100ms
- LSU coalescing timer of 100 ms
- (MANET) PushBack interval is 40% of retransmit int.
- (MANET) AckInterval is 90% of PushBack interval

Figure A.2: Simulation scenario.

```
# Flags to enable modified basic quagga behavior
CFLAGS    += -DOSPF6_CONFIG -DBUGFIX -DOSPF6_DELAYED_FLOOD
# To run MANET, add the following:
CFLAGS    += -DMANET -DOSPF6_MANET
# To run MPR flooding, add the following:
CFLAGS    += -DOSPF6_MANET_MPR_FLOOD
# To run surrogate hellos with MPR flooding, add the following:
#CFLAGS    += -DOSPF6_MANET_MPR_SH
# To run MDR flooding, add the following:
CFLAGS    += -DOSPF6_MANET_MDR_FLOOD
# To run the Database exchange optimzation for MDRs, add the following:
#CFLAGS    += -DOSPF6_MANET_MDR_FLOOD_DD
# To add differential hellos, add the following:
CFLAGS    += -DOSPF6_MANET_DIFF_HELLO
# To add (experimental) option to cache LSAs opportunistically,
# add the following to the above:
#CFLAGS   += -DOSPF6_MANET_TEMPORARY_LSDB
```

For many of these options, the above flags just enable support for the extension but don't automatically turn them on. For example, to enable Differential Hellos in a particular script, you need to do two things:

- enable `-DOSPF6_MANET_DIFF_HELLO` in the top-level Makefile
- set the ospf6_inst interface variable "diff_hellos" to "true". For an example of how to do this, see the `random_waypoint_maney.cc` program; by appending "diff_hellos" at the command line, it will enable differential hellos with the follwoing code snippet:

```
ospf6_inst->SetInterfaceDiffHellos(*it, ca.diff_hellos);
```

Below, we discuss how these options can be selectively enabled and disabled once they are compiled in.

Configuring the Basics

The number of nodes in the MANET can set at the command line. By default, 20 nodes are used. The size of the square topology can be set at the command line. By default, the topology is 500 meters by 500 meters. The location of the nodes is defined by a random generator of x and y coordinates with the max and min values being the size of the topology. *GTNets* provides various random variable types, so the distribution of node positions can be changed by chaning the random variable in `WirelessGridRectangular()`.

GTNets also has the ability to set the shape of the grid. Shapes are found in documentation. We have only experimented with rectangular and polar.

The time at which the simulator is stopped can be configured by changing the stop time at the command line. However, if this value is set below start time then Ospfv3 routers will never fully come up. A start time delay is useful to avoid synchronization of Ospfv3 routers and to give the routers time to initialize before staistics are collected and data is sent.

Configuring Wireless Behavior The Wireless mode is defined by the WirelessGrid constructor. See the GTNetS documentation for further configuring the wireless model.
Mobility is set to random waypoint. The random waypoint pause and velocity can be set at the command line.

Configuring OSPFv3 Ospfv3 in *GTNets* can be configured in any way that the Quagga implementation can be configured. We have passed a select few of these parameters from quagga into GTNetS. The GTNetS Ospfv3 configuration functions write the configuration to a standard quagga config file. A sample quagga config file is seen at gtnets/SRC/ospf6d/ospf6d.conf.

The GTNetS per Ospfv3 interface paramters are seen in Subsection A.1.3 and Subsection A.1.3. The most important parameter to set for wireless interface testing is the wireless interface type. Currently, two wireless interface types are available, point-to-multipoint and manet-reliable. The wireless interface type can be modified at the command line by selecting 1 for ptmp and 2 for manet. Differential hellos can be enabled on the manet interface at the command line.

The value of the hello interval can be changed from 10 seconds by changing `HelloInteval`. The neighbor dead interval can be changed from 40 seconds by changing `DeadInterval`. The retransmit interval can be modified from the command line. The per interface routing cost can be modified by changing `cost`. By default, the interace cost is set to 1.

Two additional parameters were added to GTNetS. We added the ability to jitter all the Ospfv3 packets. The amount of jitter is uniformally distributed between zero and `interface_jitter`. The default jitter is 100 msec. In addition, we added a paramter to delay the sending of LSAs. This enables us to coalesce LSAs in fewer LSUs. The delay is by default 100 msec, and it can be changed by modifying `interface_flood_delay`.

Configuring Data Traffic All scenarios that we have configured use CBR data traffic. The level of traffic is configured on a network wide basis, so the same load of data traffic is produced for any number of nodes when the packet rate and packet size are fixed. The traffic load can be configure on the command line by setting pktrate and pktsize. The default packet rate is 10 packets per second, and the packet size is 40 bytes.
All data to wireless nodes must be sent to their loopback addresses. The loopback addresses on wireless nodes has been configured to be 0.0.RouterId.0/24.

Example Test Program

This sample program, `ietf/random_waypoint_manet/random_waypoint_manet.cc`, creates a scenario similar to that found in Figure A.2. The scenario consists of N mobile nodes in a L meter square grid. Nodes move according to the random waypoint model where each node selects a destination within the grid and a velocity between 0 and V m/s. The node then moves to the destination, pauses for P seconds, selects a new destination and velocity, and repeats the process. Each node is a router and has a wireless interface with an 802.11b or TDMA radio at a carrier rate of 11Mbps. Each router also has a loopback interface. Each host generates constant bit rate (CBR) traffic to every other node by sending periodic B byte packets at a packet rate of R pkt/sec. Each simulation trial is M minutes long, of which the first D minutes of data is discarded; each OSPF router is started at a random time in the first D minutes. The wireless interface can be configured to use OSPFv3 routing with either a point-to-multipoint or manet interface.

Include Files Necessary include files are listed below.
File application-cbr.h is for CBR traffic generation. File args.h is used for processing command line input. File ipaddr.h is a class for ipv4 addresses. File mobility-random-waypoint.h provides random waypoint mobility. File node.h is the

base class for nodes. File rng.h is for random variable operations. File simulator.h is the base simulator class for *GTNets*Ḟile validation.h is the method to parse command line input used by *GTNets*Ḟile wireless-grid-rectangular.h is responsible for the number of nodes in the MANET, network topology, and wireless interface configuration.
File wlan.h supports a wireless link layer. These last three files were developed by Boeing. File application-ospf6d.h is the main driver of quagga ospf6d. File etrace.h provides packet level tracing. File estat.h provides global statistics.

```
// Copyright 2004, The Boeing Company
#include <iostream>
#include <stdlib.h>

#include "application-cbr.h"
#include "application-ospf6d.h"
#include "args.h"
#include "estat.h"
#include "etrace.h"
#include "ipaddr.h"
#include "mobility-random-waypoint.h"
#include "node.h"
#include "rng.h"
#include "simulator.h"
#include "validation.h"
#include "wireless-grid-rectangular.h"
#include "wlan.h"
```

Set Seeds Function The function `set_global_seeds()` is used to set GTNets global seeds from a single seed. When GTNets global seeds are set, results are repeatable when rerunning a scenario with the same seed. The seed should be set greater than or equal to one.

```
void set_global_seeds(int seed)
{
  // turn one seed into 6 repeatable seeds in the simulator using srand()
  unsigned long seeds[6];
  srand(seed);

  seeds[0] = rand();
  seeds[1] = rand();
  seeds[2] = rand();
  seeds[3] = rand();
  seeds[4] = rand();
  seeds[5] = rand();

  Random::GlobalSeed(seeds[0],seeds[1],seeds[2],seeds[3],seeds[4],seeds[5]);
}
```

Main Function Initialization The start of the `main()` function begins in this subsection. Here the objects in the main function are initialized. `Validation::Init()` is used to set seeding, animation, and tracing in *GTNets*Ṫhe function, `set_ospf6_config_arguments` sets all the default values used for ospf and stores them in `ca`. Execute `random_waypoint_manet-{opt,dbg} -h` at the command line to see the default values. Default values are set in `args.cc`. The function, `set_global_seeds()`, sets the global seed by calling the function in Section A.1.3.

```
int main(int argc, char** argv)
{
  struct ospf6_config_args ca;
```

```
Validation::Init(argc, argv);
Simulator s;
Interface *inter;
IFVec_t vInterface;
std::string tracefile, statfile;
char hostname[20], logfilename[20];
IPAddr_t first_rtrid;

// Default configuration arguments are located in SRC/args.cc file
if (!set_ospf6_config_arguments(argc, argv, &ca))
  return 0;

set_global_seeds(seed);
```

Output File Configuration In this subsection, the tracefile and statfile names are set. If a tag was defined at the command line, the tag string will be appended to the file name. Stat files are ended with .stat and trace files are ended with .tr.

```
// If user specifies "tag=args", this means that the suffix appended
// to the .stat or .tr filename should be the concatenation of all command
// line arguments, e.g.:
// random-waypoint-mobility-opt-num_nodes=20.seed=72.stat
if (ca.tag == "args") {
  ca.tag.erase(0,4);
  for (int q = 1; q < argc; q++) {
    if (!strcmp(argv[q],"tag=args") ||
        !strcmp(argv[q],"trace") ||
        !strcmp(argv[q],"etrace") ||
        !strcmp(argv[q],"ospfv3.log"))
      continue;
    ca.tag.append(argv[q]);
    ca.tag.append(".");
  }
  ca.tag.erase(ca.tag.length()-1, 1);
}

// Prepare output file names
tracefile.append(argv[0]);
tracefile.erase(tracefile.length()-4,4); //delete "-opt" or "-dbg" from name
statfile.append(argv[0]);
statfile.erase(statfile.length()-4,4); // delete "-opt" or "-dbg" from name

// Make sure that output file names are less than 256 char limit
if ((statfile.length() + ca.tag.length() + 7) > 256) {
  int overrun =  statfile.length() + ca.tag.length() + 7 - 256;
  ca.tag.erase(ca.tag.length()- overrun, overrun);
}

if (ca.tag.size()) {
  tracefile.append(".");
  tracefile.append(ca.tag.c_str());
}
tracefile.append(".tr");
```

```
  if (ca.tag.size()) {
    statfile.append(".");
    statfile.append(ca.tag.c_str());
  }
  statfile.append(".stat");

  cout << statfile.c_str() << endl;

  // Capture command line arguments for logging purposes
  for (int q = 0; q < argc; q++) {
    argstring.append(argv[q]);
    argstring.append(" ");
  }
```

Topology Configuration The topology of the network is configured in this subsection. The ipv4 address of the wireless interface on node 0 is set to 10.0.0.1. Each subsequent node's wireless interface will receive an address incremented by 1 from the first. To use Ospfv3 with ipv4, the ip address must be less than 128.0.0.0 because the first bit in the ipv4 address is used to indicate a linklocal address when converting to an ipv6 addresses with ospfv3.
A square topology with the bottom left corner at location (0,0) and sides `ca.grid_length` meters is created by `WirelessGridRectangular()`. The number of nodes is a constant defined by `ca.nNodes`. The nodes are uniformly distributed on the square grid. The wireless MAC layer is also specified in this function.
Random waypoint mobility is defined by fuction, `AddMobility(RandomWaypoint())` if the velocity is greater that zero. The nodes travel to a random location with the topology at a speed uniformally distributed between 0 and `ca.velocity` m/s. After reaching the destination, the node pauses for `ca.pause_time` seconds and then moves again.

```
  // ipv4 addresses must be less than 128.0.0.0 for ospf6 because we used
  // the first bit in the ipv4 address to indicate a linklocal address
  IPAddr_t firstIP = IPAddr("10.0.0.1");

  Location l(0,0); // lower left corner of rectangular grid

  // Randomize the distribution of nodes uniformly on square grid
  WirelessGridRectangular g(ca.mac_protocol,
                            l,
                            Constant(ca.nNodes), //number of nodes
                            Uniform(0.0, ca.grid_length),
                            firstIP);

  // Mobility:  Pause time is second argument, velocity is third argument
  if (ca.velocity > 0)
    g.AddMobility(RandomWaypoint(g, Constant(ca.pause_time),
                                 Uniform(0,ca.velocity)));
```

Tracing Configuration Extended tracing is configured in this subsection. Exteneded tracing was developed by Boeing and more specifics can be found in Section A.1.4. By default, tracing is disabled. If tracing is enbabled at the command line then the tracefile is opened. Extended tracing is also disabled by default. If it is enabled at the command line then fuction, `SetExtended()`, enables extended tracing. Extended tracing adds more detail to the basic packet level tracing.

The ability to turn off tracing of a specified layer is provided by function, `LayerOff()`. By default, all layers are enabled. Finally, additional events can be written to the tracefile. Events are used to display information or statistics not seen in packet level tracing. The function, `SetTraceEventLevel()`, sets the verbosity of events to display. A value of zero means no event tracing.

```
  // Per-packet tracing configuration
  if (ca.trace) {
    ETrace* egs = ETrace::Instance();
    egs->Open(tracefile.c_str());
    if (ca.etrace)
      egs->SetExtended(true);
    egs->LayerOff(1); //don’t etrace this layer
    egs->LayerOff(2); //don’t etrace this layer
    egs->SetTraceEventLevel(2);
  }
```

Statistics Configuration The collection of statistics is initialized in this subsection. By default, statistics are disabled. If statistics are enbaled at the command line then statistics begin to be collected at `start_time` seconds, and the results are output to the statistics file.

```
  // Statistics collection
  EStat* estat = EStat::Instance();
  estat->CollectEStats(statfile.c_str(), ca.start_time);
  estat->LayerOff(1);
  estat->LayerOff(2);
```

Ospf6 and CBR Intitialization The intitilization of Ospfv3 routing and CBR traffic is performed in this subsection. The OSPF6DApplication must be created to run ospf6. The collection of Ospfv3 stats is configured by `CollectStats()`. The rate of monitoring the physical layer neighbors is set by function, `LogNeighbors`. These physical neighbor statistics are printed in the statistics file. The first router id indicates the id assigned to the first node to have an OSPF6DInstance.
The rate of the cbr application is set so the whole MANET network generates `pktrate` packets per second, each of `pktsize` bytes. To avoid synchronization, the cbr traffic is started at a random time within the packet send interval.

```
  // OSPF6 initialization
  OSPF6DApplication ospf6d;
  ospf6d.CollectStats(statfile.c_str(), argstring.c_str(), ca.start_time);
  ospf6d.LogNeighbors(0.5);  // Log physical layer neighbors every 500 ms

  OSPF6DInstance *ospf6_inst;
  Uniform u(0,ca.start_time);
  first_rtrid = IPAddr("0.0.0.1");

  // CBR traffic initialization
  IPAddr remoteIP;
  char buf[32];
  Rate_t rate_bps = ((Rate_t)ca.pktrate * ca.pktsize * 8) /
                    (ca.nNodes*(ca.nNodes-1));
  // start traffic generators at a random time
  Uniform cbrstart(ca.start_time-(double)(ca.pktsize*8)/rate_bps,ca.start_time);
```

Per Node Configuration The start of a `for` loop that loops over all nodes is begun in this subsection. Each nodes radio range is set to the identical value, `radio_range`.

```
 for (Count_t i = 0; i < g.Size(); ++i)
  {
    Node* n = g.GetNode(i);
    n->SetRadioRange(ca.radio_range);
```

Ospf6 per Node Configuration In this subsection, each node is given an ospf6 instance. Also, the host name, log file name, and id of each router is set. The MinLSInterval and MinLSArrival are set here. The MinLSInterval is the minimum time between originating the same LSA. The MinLSArrival is the minimum time in which a router will accept the same LSA.

```
// OSPF6 Router Configuration
ospf6_inst = ospf6d.AddNode(n);

sprintf(hostname, "ospf6_%d", n->Id()+1);
sprintf(logfilename, "log_ospf6_%d.log", n->Id()+1);

ospf6_inst->SetHostName(hostname);
if (ca.ospfv3log)
  ospf6_inst->SetLogFileName(logfilename);
ospf6_inst->SetRouterId(n->Id()+first_rtrid);
ospf6_inst->SetMinLSInterval(ca.MinLSInterval);
ospf6_inst->SetMinLSArrival(ca.MinLSArrival);
```

Ospf6 per Interface Configuration In this subsection, the interfaces per node are configured with ospf6 interface specific parameters. The time between periodic hello sends on an interface is set with `SetInterfaceHelloInterval()`. The time interval when a neighbor dies due to inactivity of hello reception is set by `SetInterfaceDeadInterval()`. The interval to wait before retransmitting an LSA is set by `SetInterfaceRetransmitInterval()`. The time which a nonactive overlapping relay waits before flooding an LSA is set by `SetInterfacePushBackInterval()`. The time that a router waits to colesce LSAcks is set by `SetInterfaceAckInterval`. The ospf6 area this interface is on is set with `SetInterfaceArea()`. The ospf cost of sending on this interface is set with `SetInterfaceCost()`. The max jitter that is added to the sends of ospf6 packets is set with `SetInterfaceJitter()`. The packet jitter can be thought of as queing delay of `interface_jitter` msec before it is sending. Finally, the time to wait to accumulate LSAs before flooding is set with `SetInterfaceFloodDelay()`.

```
vInterface = n->Interfaces();
for (IFVec_t::iterator it=vInterface.begin(); it!=vInterface.end(); ++it)
{
  ospf6_inst->SetInterfaceHelloInterval(*it, ca.HelloInterval);
  ospf6_inst->SetInterfaceDeadInterval(*it, ca.DeadInterval);
  ospf6_inst->SetInterfaceRetransmitInterval(*it, ca.RxmtInterval);
  ospf6_inst->SetInterfaceAckInterval(*it, ca.AckInterval);
  ospf6_inst->SetInterfaceArea(*it, 0);
  ospf6_inst->SetInterfaceCost(*it, ca.cost);
  // too much jitter can cause stale ospf6 state to be sent
  ospf6_inst->SetInterfaceJitter(*it, ca.interface_jitter); //msec
  ospf6_inst->SetInterfaceFloodDelay(*it, ca.interface_flood_delay); //msec
```

Ospf6 per Interface Type Configuration The type of interface is set based on the characteristics of the interface. The wireless interface gets either a point-to-multipoint or manet ospf6 interface. If a manet interface is used then differential hellos and can be enabled, and either broadcast, MDR, or MPR flooding can be used. The PushbackInterval and BackupWaitInterval sets the time backup relay wait to flood an LSA and they are set here. MDRs have options for the level of adjacency connectivity and and the fullness of the router LSAs. The parameter alpha influences the probability of a packet being received within the radio range. When alpha = 1, all packets sent within radio range are received. See Section 3.2.1 for more details.

The loopback interface is set to Ospfv3 type loopback.

Then cbr traffic is sent from this node to the loopback address of every other router. The cbr traffic starts just prior to `start_time` seconds and ends at `stop_time - 5` seconds. The start time is jittered so synchronization among routers is avoided.

Finally, all other interface are set to Ospfv3 type broadcast. In this scenario there are no other interfaces, so this event will not occur.

```
if ((*it)->GetL2Proto()->IsWireless())
{
  if(ca.w_int_type == 1)
    ospf6_inst->SetInterfaceType(*it, "point-to-multipoint");
  else if (ca.w_int_type == 2)
  {
    ospf6_inst->SetInterfaceType(*it, "manet-reliable");
    ospf6_inst->SetInterfaceDiffHellos(*it, ca.diff_hellos);
    //set flooding type
    if (ca.w_flooding == 0)
      ospf6_inst->SetInterfaceFlooding(*it, "broadcast");
    else if (ca.w_flooding == 1)
    {
      ospf6_inst->SetInterfaceFlooding(*it, "mpr");
      ospf6_inst->SetInterfacePushBackInterval(*it, ca.PushbackInterval);
    }
    else if (ca.w_flooding == 2)
    {
      ospf6_inst->SetInterfaceFlooding(*it, "mdr");
      ospf6_inst->SetInterfaceBackupWaitInterval(*it,
                                           ca.BackupWaitInterval);
      ospf6_inst->SetInterfaceTwoHopRefresh(*it, ca.TwoHopRefresh);
      ospf6_inst->SetInterfaceRouterDeadCount(*it, ca.RouterDeadCount);
      if (ca.AdjConnectivity == 0)
        ospf6_inst->SetInterfaceAdjConnectivity(*it, "fully");
      else if (ca.AdjConnectivity == 1)
        ospf6_inst->SetInterfaceAdjConnectivity(*it, "uniconnected");
      else
        ospf6_inst->SetInterfaceAdjConnectivity(*it, "biconnected");
      if (ca.LSAFullness == 0)
        ospf6_inst->SetInterfaceLSAFullness(*it, "minlsa");
      else if (ca.LSAFullness == 1)
        ospf6_inst->SetInterfaceLSAFullness(*it, "minhoplsa");
      else if (ca.LSAFullness == 2)
        ospf6_inst->SetInterfaceLSAFullness(*it, "mdrfulllsa");
      else
        ospf6_inst->SetInterfaceLSAFullness(*it, "fulllsa");
      ospf6_inst->SetInterfaceNonPersistentMDR(*it, ca.NonPersistentMDR);
    }
  }
  if (i==0)
  {
    WirelessLink *l = (WirelessLink *) (*it)->GetLink();
    l->SetAlpha(ca.alpha);
  }
}
else if (strcmp((*it)->GetName(), "lo") == 0)
{
  ospf6_inst->SetInterfaceType(*it, "loopback");
  if (rate_bps > 0 && ca.pktsize > 0)
  {
          // CBR traffic from this router to all other router’s loopback addr
          for (Count_t j = 0; j < g.Size(); ++j)
          {
            if (i == j)
              continue;
            sprintf(buf, "0.0.%d.0", j+1); //loopback addr of routers
            remoteIP = IPAddr(buf);
            CBRApplication* cbr = (CBRApplication*) n->AddApplication(
                    CBRApplication(remoteIP,1000,NO_PORT,rate_bps,ca.pktsize));
            cbr->Start(cbrstart.Value());
            cbr->Stop(ca.stop_time - 5);
          }
        }
      }
      else
        ospf6_inst->SetInterfaceType(*it, "broadcast");
    }
```

Setting Start and Stop Times The Ospfv3 routers are started at a random time between 0 and `start_time` seconds. This prevents Ospfv3 packets from being synchronized. The simulation starts at time 0 and ends at time `stop_time` seconds. The progress is displayed at the command line every 40 seconds. Simulations are commenced by function `Run()`.

```
    ospf6_inst->Start(u.Value());  // random start time of each ospf6 router
  }
  s.StopAt(ca.stop_time);
  s.Progress(40);
  s.Run();
}
```

Example Test Programs

- **ietf/initial/basic.cc**. This program, described above in the example, runs a basic wireless network with Ospfv3 routing.
- **ietf/random_waypoint_manet/random_waypoint_manet.cc**. This program is similar to the basic.cc program, but it contains many new updates. It includes support for using a TDMA MAC based on the Unifying Slot Assignment Protocol (USAP).
- **validation/relays/basic/basic.cc** Performs validation of the MPR algorithm.
- **validation/relays/pushback1/lsu_loss.cc** Performs validation of an LSA being pushed back after lossing the initial flood.
- **validation/relays/pushback2/lsack_loss.cc** Preforms validation of losing an LSAck and then having the push-back timer expire.
- **validation/relays/pushback3/lsu_loss_pushback_lsu_loss.cc** Performs validation of an LSA being lost, having a node send out a pushback LSA and loose it, and then finally having another node send a pushback LSA.
- **ietf/link_matrix_manet/link_matrix_manet.cc** Uses an extension to ethernet (known as link matrix) that allows the bringing up and down of specifed links at particular times.
- More scripts are included in the ietf and validation subdirectories. Please contribute any interesting scripts.

A.1.4 Simulation Output

There are two main ways to look at simulation output:

1. Explicit "tcpdump"-like tracing of packets to an output file that can be post-processed later
2. Compilation of summary statistics that can be printed out at the end of the simulation

GTNets had a basic tracing capability, but we have extended it further to print out more detailed logging. This can be used to examine trace output for correct or errored behavior (much like a tcpdump output file), or can be fed into post-processing scripts to gather statistics. The file would typically be called `testname.tr`.

However, for large simulations, both the file size and the time to parse the traces can become very large, so we also provide a capability to dump summary statistics into a file at the end of the simulation run (into a file typically called `testname.stat`).

Tracing

How to Enable Tracing *GTNets* has a built-in tracing facility based on the *Trace* object. We cloned that object class to create the Extended Trace, or *ETrace* object. Either, neither, or both Trace and ETrace can be enabled on a particular script.

```
ETrace* egs = ETrace::Instance();
egs->Open("<file name>");
```

Figure A.3: Sample code to control tracing

Figure A.3 illustrates the basic C++ commands to add to your testfile to enable tracing. This snippet creates an ETrace object and associates it with a file. The filename can be anything you want. The semantics are to overwrite (not append) any existing file that may be at that location.

```
// Per-packet tracing configuration
if (ca.trace) {
  ETrace* egs = ETrace::Instance();
  egs->Open(tracefile.c_str());
  if (ca.etrace)
    egs->SetExtended(true);
  egs->LayerOff(1); //don’t trace this layer
  egs->LayerOff(2); //don’t trace this layer
  egs->TraceTopo(ca.tracetopo);  // trace topology every tracetopo seconds
  egs->SetTraceEventLevel(2);
}
```

Figure A.4: Sample code to control tracing

Figure A.4 shows how to control options on what is and isn’t traced. By default, all protocol layers on all nodes are traced. There are a number of options to control this, however:

- SetExtended(true|false). This will enable extended tracing, which basically means to trace beyond the IP header and into the OSPF packet body. Default is false.
- SetEventTraceLevel(0|1|2). This controls the level of debugging output for protocol events (non-packet events) put into the tracefile (the higher the number, the more events that are logged). Level 1 adds the following events:
  - Count of OSPF packet drops and OTHER packet drops upon end of simulation
  - Information about the number of LSAs out of sync during database exchange
  - when a unicast LSU is sent

  - statistics on the number of neighbors of a node whenever a neighbor change event occurs

  Level 2 adds the following events:

  - Prints out when the router started operation
  - Dumps relay selector lists
  - More information traced regarding neighbors changing state

- LayerOff(n). This indicates that tracing at a particular protocol layer is disabled.
- NodeOff(n). This indicates that tracing at a particular node is disabled.
- TraceTopo(n). Enable the tracing of topology information every n seconds.

```
s 3599.97947 N12 -int 1 -pkt_id 1329237  -l3proto IPv4 -src 10.0.0.13 -dst 0.0.1
4.0 -len 92 -ttl 64 -proto UDP
r 3599.97992 N8 -int 1 -pkt_id 1321800  -l3proto IPv4 -src 138.0.0.14 -dst 138.0
.0.9 -len 56 -ttl 0 -proto OSPFv3 -type LSAck
r 3599.98195 N13 -int 1 -pkt_id 1329237  -l3proto IPv4 -src 10.0.0.13 -dst 0.0.1
4.0 -len 92 -ttl 63 -proto UDP
```

Figure A.5: Sample tracefile excerpt, with sample 80-column wrap

Sample Tracefile Figure A.5 illustrates a sample tracefile excerpt from the testwirelessospf.cc script. In this example, extended tracing is off, so OSPF protocol parsing beyond the packet type is disabled. Each traceline starts with a single character event code. The following event codes are defined.

- **s**: Packet send event
- **r**: Packet receive event
- **f**: Packet forward event
- **d**: Packet drop event
- **c**: Packet collision event
- **e**: Protocol event (non-packet event)
- **n**: Node location and DR level(non-packet event)
- **a**: Adjacency involving a DR_OTHER router (non-packet event)
- **b**: Adjacency between (B)MDRs (non-packet event)

The next field is the node number, followed by a number of key-value pairs:

- **int**: Interface number that packet was seen on
- **pkt_id**: Packet unique id
- **l3proto**: Layer 3 protocol in use
- **...**: Other fields are probably self explanatory

These trace lines will contain more information (expanded) if extended tracing is enabled.

Animating MDR Topology

The topology of the OSPF-MDR interface can be displayed graphically in the form of an animated "gif" file. The animated gif is created by enabling topology tracing and by post processing the tracing to form the "gif" file. The appropriate trace lines are output by calling the function `ETrace::TraceTopo()`. If one is using `random_waypoint_manet.cc` then either `tracetopo=<>` or `tracetopoonly=<>` are called from the command line. The input parameter is the sampling rate in seconds. The final step is to run `gtnets/bin/tracetogif.pl` on the trace file. The usage of `tracetogif.pl` can be determined by running it with no input parameters. An example parameterization is shown here: `./tracetogif.pl -stop 2000 -delay 200 trace.tr`. The following would create an animated gif of the topology data between 1800 and 2000 secs, and it would shows snapshots of the topology every 2 seconds.

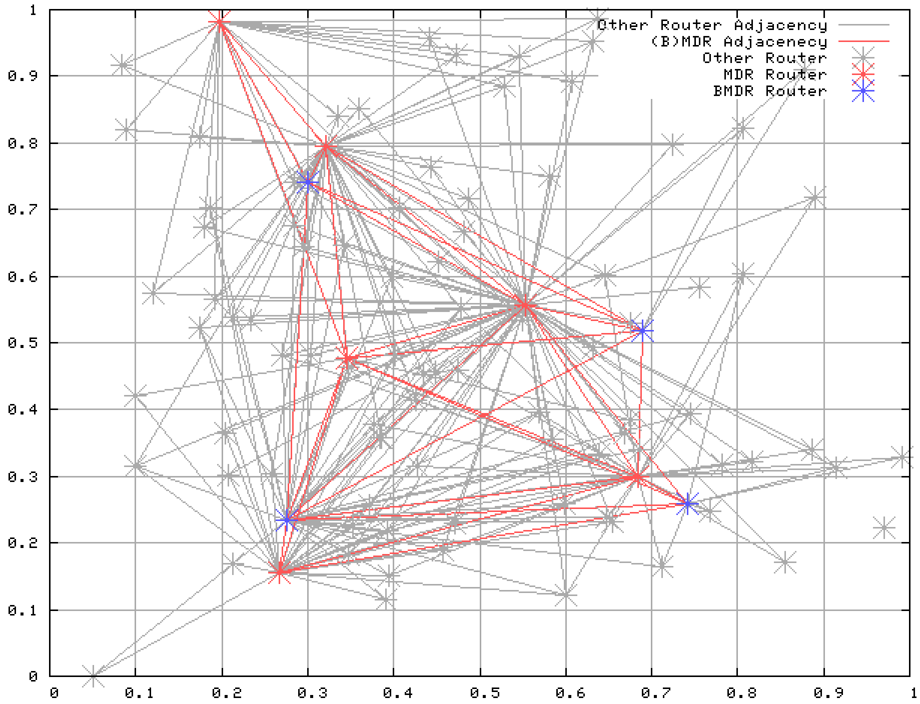


Figure A.6: 100 node OSPF-MDR topology with bi-connected adjacencies and (B)MDR full LSAs

Statistics

To be able to turn off tracing for larger scenarios, we added some counter-based statistics gathering. At present, this is turned on by adding the following lines to a particular script:

```
//Extended Statistics Configuration
EStat* estat = EStat::Instance();
estat->CollectEStats(statfilename, delay);
estat->LayerOff(1);
estat->LayerOff(2);
//Ospfv3 Statitistics Configuration
OSPF6DApplication ospf6d;
ospf6d.CollectStats(statfilename, delay);
```

Figure A.7: Sample code to control tracing

The output of the statistics will be deposited in the filename specified, with a `.stat` suffix.

Currently, EStat only generates statitics for UDP traffic, and all Ospfv3 stats are generated by `class OSPF6DApplication`. We did not use *GTNets* built-in Statistics class because that code is not compatible with quagga.

To add additional statistics to Ospfv3, the statistic name must be added to the enumeration `ospf6_stats` in `ospf6_top.h`. Each of these names are elements in the statistics array found in struct ospf6. Next, the appropriate place to count the statistic must be found, and the statisitic is counted by incrementing the element in the array statistic. Statistics in struct ospf6 are per node, so in the `application-ospf6d.cc` function `OSPF6DApplication::ExtractStats()` the node stats are summed to create global network stats. The final step in creating a new stat is to add the printout of the value in the `application-ospf6d.cc` function `OSPF6DApplication::PrintStats()`.

Logging

Logging build into quagga Ospfv3 has been integrated with *GTNets*, so the same logging produced by quagga can be produced in *GTNets*. Quagga logging creates a seperate log for each router. To use the logging in *GTNets*, the log file for each node must be named by calling the function `OSPF6Instance::SetLogFileName()`. When this function is called, logging will be generated for LSAs, Hellos, LSUs, LSAcks, LSRs, and Database Description packets. In addition, logging will be generated for neighbor and interface changes. The type of logging generated can be changed in `application-ospf6d.cc` function `OSPF6DInstance::CreateConfFile()`.

A.1.5 Validation

This section explains the validation of the OSPF modifications. Our validation scripts serve two purposes:

1. Detection of code modifications that inadvertently break the correct behavior of a protocol.
2. Isolation of particular protocol elements to ensure that they are working correctly.

In this section, we first describe the general script that can be used to detect changes to the behavior of protocols. We then follow with some examples of how we validated the specific OSPF modifications.

The "validate" Script

There is a script called `validate` in the top level directory. This script finds all executables in any subdirectories in the `validation/` directory and executes them. Each directory in `validation` contains the validation driver, an expected trace file, and expected stat file. The `validate` script compares the expected output with the current output. Any differences are output to a diff file, and they are flagged as a `FAILURE` at the command line. To run `validate` the environment variable `GTNETS_HOME` must be set to *GTNets* base directory. This is very similar to how the ns-2 simulator validation works.

The idea behind this is that if there are changes to the behavior of the protocols covered by example scripts, then the validation output will start to diverge from the known good output. The researcher can then run the `validate` script to find out what scripts are broken, examine the `diff` file produced, and determine whether the source code is broken or whether the expected known good output needs to be updated (for `validate` to pass in the future).

Basic Validation

Let's first look at a very simple network– four OSPFv3 nodes fully meshed on a broadcast-based subnet (e.g., Ethernet) but running the multicast-capable point-to-multipoint (i.e., non-MANET) interface type. We will use the "Matrix" channel, a prototypical script for which can be found in `ietf/link_matrix_manet/link_matrix_manet.cc`. "Matrix" channel is intended to allow the packet delivery ratio between any pairwise set of nodes to be altered, but the default is full mesh connectivity.

This executable is built if the `ietf/` directory was enabled; this can be done with, for example, `make ietf-opt`, `make ietf-dbg`, etc. from the top level directory (see Section A.1.1).

Let's assume that the "optimized" version of the binary has been built. Type `./link_matrix_manet-opt -help` to see the list of options and default values.

We will run with most of the defaults. However, let's enable all output tracing and logging, and set the number of nodes from its default of 20 to just 4, and turn off the minimal user data traffic by setting the packet rate to zero:
`./link_matrix_manet-opt num_nodes=4 pktrate=0 trace etrace ospfv3log`

Next, type `more link_matrix_manet-opt.stat` to look at the statistics file generated. This simulation ran for 3600 seconds, and statistics were collected over the last 1800 seconds (total of 1800 seconds simulation time). There is not much of interest to see; there were 720 Hello messages (one Hello per node every 10 seconds for four nodes and 1800 seconds yields 720 Hellos). There were 32 LSU packets sent, and (further down in the statistics output) 48 "mpr_flooded_LSAs". We will next sanity check these results by looking at the output trace file.

The detailed per-packet and per-event tracing can be found in the `link_matrix_manet-opt.tr` file. Even for this small quiet network, there are over 4500 lines of trace output generated. Grep for "Started" and you should see something like:

```
e 31.41292 N1 Ospf6 Router Started
e 57.94125 N2 Ospf6 Router Started
e 1274.60552 N0 Ospf6 Router Started
e 1634.38450 N3 Ospf6 Router Started
```

illustrating the starting times of each router for nodes 0 through 3 (the event code "e" signifies that this is an implementation event and not a packet).

For convenience, let's remove all Hello messages from the trace output:
`grep -v HELLO link_matrix_manet-opt.tr > output.tr`. Now, let's look at the resulting `output.tr` file. We know that the first router started up at time 31.4, so at around time 1831.4, there should be some re-originated LSAs. This can be seen in the `output.tr` file:

```
e 1831.41492 N1 Re-orig LSA Link -id 0.0.0.1 -advrt 0.0.0.2 -age 0 -seq 21474836
50 -len 64
e 1831.41492 N1 Schedule Flood LSA Link -id 0.0.0.1 -advrt 0.0.0.2 -age 0 -seq 2
147483650 -len 64 from (null)
e 1831.41492 N1 Re-orig LSA Intra-Prefix -id 0.0.0.0 -advrt 0.0.0.2 -age 0 -seq2
147483650 -len 72
e 1831.41492 N1 Schedule Flood LSA Intra-Prefix -id 0.0.0.0 -advrt 0.0.0.2 -age0
 -seq 2147483650 -len 72 from (null)
s 1831.53392 N1 -int 1 -pkt_id 519  -l3proto IPv4 -src 138.0.0.2 -dst 224.0.0.5-
len 176 -ttl 1 -proto OSPFv3 -type LSU -len 156 -rid 0.0.0.2 -aid 0.0.0.0 -Num2
-Lsa Link -id 0.0.0.1 -advrt 0.0.0.2 -age 1 -seq 2147483650 -len 64 -Lsa Intra-P
refix -id 0.0.0.0 -advrt 0.0.0.2 -age 1 -seq 2147483650 -len 72
```

The two LSAs (Intra-Prefix and Link) are generated, and sent in a single LSU at time 1831.53392. Scrolling down in the tracefile, we can see that nodes N0, N3, and N4 also receive and reflood this LSU. This accounts for a total of 4 LSU transmissions, and there are 4 nodes that will perform this reorigination, so this accounts for 16 of the LSUs, and 32 of the LSAs in the above cited statistics counts. Now, consider that the Router LSAs are expiring at a different time, since they were all changed after the fourth router originally came up. Therefore, each node will generate an additional LSU, causing four LSU transmissions per node (an additional 16 LSU transmissions and LSA floods) starting around time 3435. This yields our counts of 32 LSU transmissions and 48 LSA floods.

Note that if there were a quagga implementation, output would be stored in output logs; these should be now available in the `log` directory.

Finally, let's re-run the above script but with "manet" and "mpr" flooding options enabled (corresponding to the Cisco overlapping relays technique):

```
./link_matrix_manet-opt num_nodes=4 pktrate=0 trace etrace ospfv3log
wireless_interface=2
```

We can see from the stat file that we have now managed to suppress 36 of the previous 48 LSAs, resulting in only 8 LSU transmissions, but now require 19 LSAcks to compensate from the loss of implicit acks that were previously inferred via reflooding.

Radio Range

The directory `validation/radio_range` contains some scripts used to determine the radio range of the selected 802.11-based radios, based on varying the distance between nodes. These scripts are not well-documented at this time, but were used to check the radio behavior for different parameterization.

A.2 Implementation

This section describes how one can use the MANET extensions to ospf6d found in GTNetS within the Quagga implementation. Although the OSPF MANET extensions were originally begun with a prior version of Quagga, we have been tracking the progress of the Quagga project, and we are currently at quagga-0.98.4. We have been running Quagga on `Fedora Core 4`, but it should support gnu/Linux, BSD and Solaris. See `http://www.quagga.net/docs/docs-info.php#Supported-Platforms` for more details on supported platforms.

A.2.1 Applying the MANET Extensions to Quagga

We provide two methods for using the GTNetS code in Quagga. The first is to patch the Quagga code with a patch file, and the second is to import the code from GTNetS into Quagga. The following steps are common to both approaches.

- Download `quagga-0.98.4.tar.gz` from `http://www.quagga.net`.
- Unzip and extract the Quagga source by running `tar -xzvf quagga-0.98.4.tar.gz` at the command line.
- Change directory into quagga-0.98.4. We will futher refer to it as the quagga directory.

Patching Quagga

This subsection explains the simple method to patching the Quagga source. In the quagga directory, apply the patch by executing `patch -p1 < gtnets/quagga-0.98.4_manet.patch` at the command line.

Adding Changes to Quagga Manually

Here is a step by step method to take the ospf6d code from GTNetS and use it in quagga-0.98.4. This is useful if you do not want to use the provided patch but instead desire to modify the code in GTNetS and subsequently use the modified code in the implementation.

- add the following lines to quagga/lib/linklist.h:

  ```
  #ifdef GTNETS
  typedef struct list *pList;
  typedef struct listnode *plistnode;
  #endif //GTNETS
  ```

- add the following line to quagga/lib/vector.h:
```
#ifdef GTNETS
typedef struct _vector *pVector;
#endif //GTNETS
```
- copy all *.c and *.h files in gtnets/SRC/ospf6d to quagga/ospf6d/
- We added `ospf6d_mpr.{c,h}` and `ospf6d_mdr.{c,h}` to ospf6, so you need to add these files to ospf6d/Makefile.in. We suggest grepping for `ospf6_spf` and duplicating the code written for this file.
- modify the DEFS in quagga/configure to contain the neccesary flags. Make sure to keep the quotes.
```
DEFS="-DHAVE_CONFIG_H -DGTNETS -DBUGFIX -DOSPF6_CONFIG
-DOSPF6_DELAYED_FLOOD -DOSPF6_MANET -DOSPF6_MANET_MPR_FLOOD
-DOSPF6_MANET_MDR_FLOOD -DOSPF6_MANET_DIFF_HELLO -DUSER_CHECKSUM
-DOSPF6_MANET_MDR_FLOOD_DD"
```
See Section A.2.3 below for more explanation of these flags.

A.2.2 Installing the Implementation

- Create a user named quagga if it does not exist: Ex. useradd quagga -d /dev/null.
- Add the ipv6 module if it does not exist: Ex. modprobe ipv6.
- change directory into quagga
- run `./configure`
- Build the code by running `make`.

See `http://www.quagga.net/docs/docs-info.php#Installation` for details instructions on command line arguments available when running `configure`.

A.2.3 Configuring the Implementation

Compile-Time Options

The following compile-time options are available. Most should be used at all times. `USER_CHECKSUM` and `OSPF6_MANET_MPR_SH` may be useful to disable.

- `GTNETS`: Code necessary to run the GTNetS code in Quagga.
- `BUGFIX`: Bugs that are fixed in Quagga.
- `OSPF6_CONFIG`: Add new configuration options to Quagga.
- `OSPF6_DELAYED_FLOOD`: Delay LSAs for coalescing
- `OSPF6_MANET`: Code need for both OR and MDR MANET interfaces.
- `OSPF6_MANET_MPR_FLOOD`: Code needed for the OR MANET interface.
- `OSPF6_MANET_MPR_SH`: Code needed to use acks as surrogate hellos with the OR MANET interface.
- `OSPF6_MANET_MDR_FLOOD`: Code needed for the MDR MANET interface.
- `OSPF6_MANET_DIFF_HELLO`: Code needed for OR incremental or MDR differential hellos.
- `USER_CHECKSUM`: A user level checksum of the IP packet when LLS signaling is used. In Linux, this option only works for glibc > 2.3. This flag is only needed if you are testing interoperablity.
- `OSPF6_MANET_MDR_FLOOD_DD`: Code needed for an optimization to database exchange with MDR MANET interface.

The ospf6d.conf File

Each of the daemons in Quagga have their own config files. For more information on these config files see `http://www.quagga.net/docs/docs-info.php#Config-Commands`. Here we explain additions we have made to `ospf6d.conf`. A sample `ospf6d.conf` file is found at `gtnets/SRC/ospf6d/ospf6d.conf` or `quagga/ospf6d/ospf6.conf`.

The following commands have been added or modified in ospf6d. If the command has been modified, only the new input values are shown below.

- `ipv6 ospf6 network <broadcast|non-broadcast|point-to-multipoint|point-to-point|manet-reliable|loopback>`: This commnad sets the OSPF inteface type. `manet-reliable` is used for ORs and MDRs.
- `ipv6 ospf6 network flood <broadcast|mpr|mdr>`: This command is used to set the flooding type when a `manet-reliable` interface type is being employed.
- `ipv6 ospf6 ackinterval <1-65535>`: The duration of time that acknowledgements are coalesed, and the maximum time an acknowledgement will be delayed before sent.
- `ipv6 ospf6 diffhellos`: Enable differential hellos when using MDR flooding and incremental hellos when using OR flooding.
- `no ipv6 ospf6 diffhellos`: Disable differential hellos when using MDR flooding and incremental hellos when using OR flooding.
- `ipv6 ospf6 pushbackinterval <1-65535>`: For ORs, the ammount of time a Non-Active relay will delay flooding.
- `ipv6 ospf6 backupwaitinterval <1-65535>`: For MDRs, the ammount of time a Backup MANET Designated Router will delay flooding.
- `ipv6 ospf6 twohoprefresh <1-65535>`: For MDRs when using differential hellos, a full state Hello is sent every twohoprefresh Hellos.
- `ipv6 ospf6 hellorepeatcount <1-65535>`: For MDRs when using differential hellos, the amount of hellos that can't be missed without losing neighbor state.
- `ipv6 ospf6 adjacencyconnectivity <fully|uniconnected|biconnected>`: The level adjacencies maintained with neighboring routers.
- `ipv6 ospf6 lsafullness <minlsa|minhoplsa|mdrfulllsa|fulllsa>`: Definition of the point-to-point links that will be advertised in the Router LSA.
- `ipv6 ospf6 flood-delay <1-65535>`: The maximum time for which LSAs are delayed and coalesced before flooding.

Command-Line Arguments

All commands made in the `ospf6d.conf` file can be made in Quagga's VTY interface. VTY can be used to change options during run time without reseting the daemon. More information on vty can be found at `http://www.quagga.net/docs/docs-info.php#Virtual-Terminal-Interfaces`.

A.2.4 Running the Implementation

Exectute the directions below as superuser to run quagga.

- Create an `zebra.conf` and place it at `/usr/local/etc` by copying `zebra/zebra.conf.sample`
- Create an `ospf6d.conf` and place it at `/usr/local/etc` by looking at Section A.2.3.

- Run zebra by running `zebra/zebra -u root -d`.
- Run ospf6d by running `ospf6d/ospf6d -u root -d`.

See the `INSTALL` file in base directory quagga for basic installation information

A.3 License and Contributing Code

This software builds on two projects (*GTNets* and quagga) that are under GNU General Public License (GPL). Our modifications are also under GPL. Therefore, please consider this simulator as GPL'ed software.

We would like to encourage people to send in bug fixes, extensions, or to make their simulation scripts available (that produce their simulation results). By default, we will not include any contributed code, bug fixes, or patches unless you specify that you want to include said code in our future releases of this simulator (also under GPL).

Appendix B

Overview

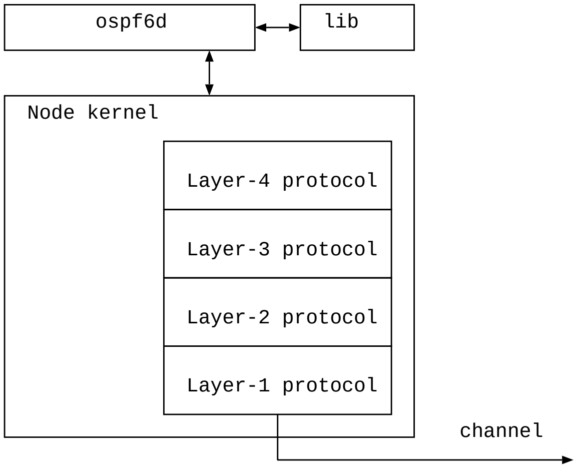


Figure B.1: Conceptual Architecture

The *GTNets* simulator is based on an object-oriented design and is written entirely in *C++*. Conceptually, the simulator consists of a number of nodes that can be interconnected by channels or links. Each node is an instance of a *C++* node class, and the node itself is a composite object that holds instances of layer-4 protocol state machines like TCP, etc.

Quagga is written entirely in C. It typically runs in user-space on a *nix platform, accessing the kernel through standard interfaces. Quagga itself consists of three pieces: a library *lib* of common data structures and functions, the *zebra* glue that interfaces with the kernel and provides inter-daemon communication when there is more than one routing daemon running (e.g., BGPd and OSPFd), and the routing protocol daemon itself, in this case *ospf6d* for IPv6. Figure B.1 shows how quagga and *GTNets* are conceptually brought together (we did not need the zebra piece).

B.1 Porting Quagga to GTNetS

Our goal was to keep the quagga source code (a very large code base) as unmodified as possible so that we could easily move code patches back and forth between implementation and simulation, and we also wanted to reduce the code that we had to write. This was a challenge because *GTNets* is an object-oriented design, while quagga code is written in C. That is, in *GTNets*, it is straightforward to create multiple nodes in a single simulation process by defining multiple instances of the Node class, but for code written in C, there is no such concept of having multiple instances of the same code running in a single process.

The options were to either port the C code into a *C++* class, or figure out a way to run the C code in the existing simulator. Because the former choice would end up touching most of the code, we did the latter.

The key to leaving most of the quagga OSPF code untouched was to have a master, simulator-wide OSPF object that was connected to all of the nodes. This OSPF meta-object holds state for a number of OSPF6DInstance objects– one per node. These OSPF6DInstance objects then hold the per-node OSPF state, make the function call-ins to the quagga `ospf6d` code, and provide some of the glue between quagga and the rest of the simulator that formerly was between quagga and the implementation kernel. Fortunately, quagga keeps the OSPF state largely in one big `struct ospf6 {}` structure (and a few other global variables), and accesses these global variables through pointers. So, the key to reusing the quagga code without touching it much is to have the OSPF6DInstance objects set all of these global pointers to point to the right instance members just before calling into the quagga `ospf6d` code.

The other major piece was reconciling the two schedulers. The implementation daemon has a scheduler based on an event queue and looping `select()` function calls to read file descriptors for input, and then periodically to read events that have queued in an event list. The quagga events are called "threads" but they really are function pointers with some metadata that allows quagga to prioritize the scheduling of the events. [1] Rather than maintain a separate quagga event list queue (which would need to be polled by the master simulator scheduler), we intercept calls to add the "thread" to the quagga event list, and instead schedule a *GTNets* event to execute at that time. The *GTNets* event contains a pointer to the thread, which itself contains the relevant function pointer. So when the event eventually executes, the OSPF6DInstance again sets the global pointers correctly before calling the function pointed to in the cached thread.

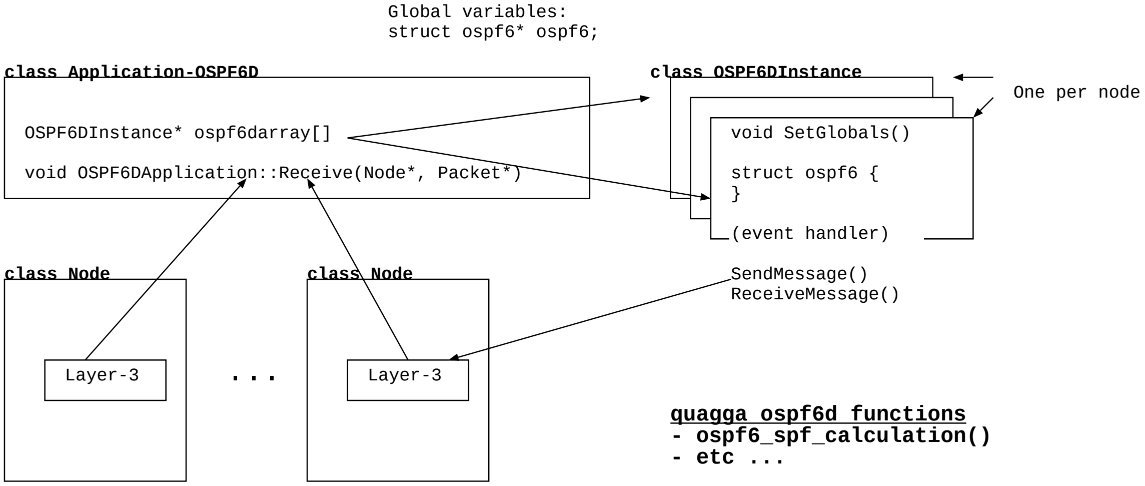


Figure B.2: Porting details

Figure B.2 is an overview of the structure of the changes. Not shown are the numerous untouched quagga `ospf6d` functions, which are called by the OSPF6Instance objects. To get a sense of how it works, consider a node receiving an OSPFv3 packet. Each node demultiplexes the packet based on protocol number ("89" for OSPF) and sends a received OSPF packet to the Application-OSPF6D object. There is one such object defined for each simulation. The Application-OSPF6D object itself demultiplexes the packet and finds the right OSPF6DInstance object to send it to (one Instance exists per node). Once inside the OSPF6DInstance (which can be thought of as a wrapper around the quagga code), packet data structures are converted to native quagga format, and all simulator global variable pointers are set to point to the Instance class members using the `SetGlobals()` function. Then, the Instance object will make a call to a quagga function such as `ospf6_receive()`, which will complete the processing of this packet.

In the reverse direction, packets being sent out of the quagga code will be picked off before making the typical `send()` system call, and shunted to the `SendMessage()` function, where packet formats are converted to *GTNets* format, and the packet is then sent to the Layer-3 object of the relevant node.

Below is a listing of the additional changes we made to the *GTNets* simulator.

- *GTNets* is only IPv4 capable, but we are running OSPFv6. Since we really don't care about the IPv6-specifics, we convert (i.e. host NAT) between IPv6 and IPv4 address and packet structures at the boundary between the OSPFv3 application and the node object, using IPv4-compatible IPv6 addresses.

[1] See http://wiki.cs.uiuc.edu/cs427/The+Thread+Mechanism+in+Zebra if you are interested in the quagga/zebra details.

- Multicast does not work in *GTNets* but we added a small hack to allow single-hop multicast of well-known OSPF multicast addresses to work
- enhanced per-packet tracing
- fixed global seeding problem that prevented repeatable mobility

B.2 Source Code Structure

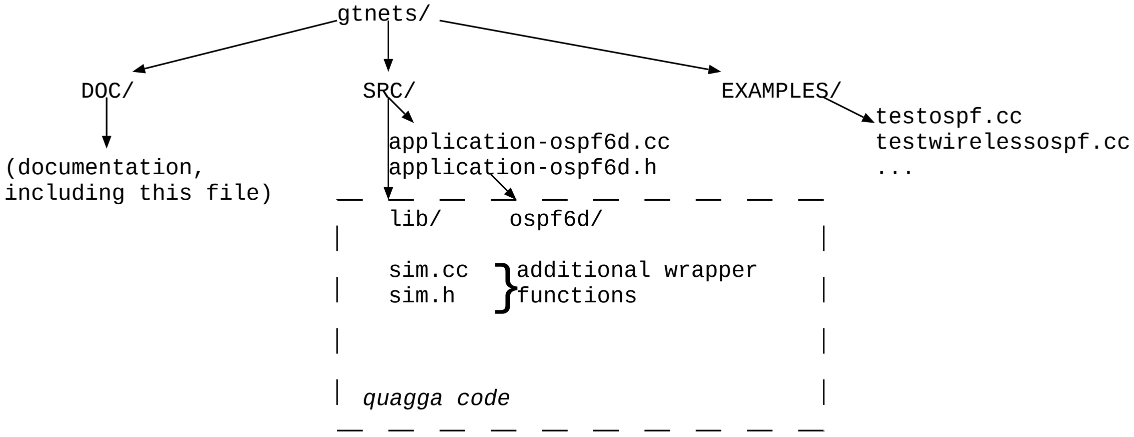


Figure B.3: Code tree

Figure B.3 shows the code tree for the simulator. Most of the source code that we wrote as wrapper is in the files:

```
SRC/application-ospf6d.h
SRC/application-ospf6d.cc
SRC/lib/sim.h
SRC/lib/sim.cc
```

All of the quagga code is in `SRC/lib/` and `SRC/ospf6d/` directories.

B.3 Design Changes to Quagga

In the course of doing this port, we discovered some bugs that needed to be fixed in quagga. We also improved the ability of quagga to perform delayed flooding of LSAs in order to coalesce multiple LSAs into one LSU. Finally, we added additional capability to jitter packet transmissions– this is more of an issue in simulation space since real-world jitter doesn't exist. Below are the details.

- Bugfixes - all tagged with the name BUGFIX
  1. ospf6_flood.c: Use thread_add_timer for delay event because thread_add_event does not delay the event.
  2. ospf6_flood.c: Set flag that a packet was flooded out the received interface.
  3. ospf6_flood.c: Respond to a received LSA that is more recent in the receiving nodes database immediately as oposed to waiting for an undetermined time.
  4. ospf6_intra.c: Fix logic error in multipath `for` loop.
  5. ospf6_lsdb.c: Clean up an LSA's expire and refresh threads when it is removed from the link state database.
  6. ospf6_message.c: Add debug message comment that was inadvertantly left out of the quagga code. This bug was fixed in later versions of quagga.

7. ospf6_message.c: Add delay when requesting an LSA from a neighbor from which LSA(s) were just received. When this delay was not present, the network was swamped with consecutive duplicate LSUs and LSRs for MinLSArrival seconds.
8. ospf6_spf.c: Delete memory that was not being freed when early exit from function `ospf6_spf_calculation()` occured.
9. ospf6_top.c: Change `for` loop to a `while` loop, so an element in the area list could be deleted.
10. ospf6_lsa.c and ospf6_intra.c,h: Enforce the MinLSInterval as specified in RFC 2328 Section 12.4 (1). The quagga code did not force an LSA to be originated at least MinLSInterval seconds after the previous origination of the same LSA. The MinLSInterval is set to 5 seconds in ospf6_proto.h.

- LSU Flooding Delay - all tagged with the name OSPF6_DELAYED_FLOOD

  1. All flooded LSAs are now delayed between 0 and flooding delay msec before they are sent. This enables LSAs to be coalesced in the same LSU packet.
  2. Retransmitted LSAs are forced to wait the retransmit interval before being sent with one exception listed below.
  3. When an LSA's retransmit timer expired and is being retransmitted to a neighbor and one or more LSAs will be retransmitted to the same neighbor within flooding delay seconds then they are sent in the same packet.
  4. See Section A.1.3 for information on configuring the delayed flooding.

- Packet Jitter - all tagged with the name OSPF6_JITTER

  1. In order to avoid synchronization of packet sends, all Ospfv3 packets were jittered between 0 and max jitter msec before sending. The delay in sending can be conceptually thought of as queuing delay on the sending interface because ospf6d is unaware of the delay.
  2. See Section A.1.3 for information on configuring the jitter.
  3. Jitter is only currently available in GtNets, but it should be added to quagga in the future.

B.4 MANET Extensions to ospf6d

We modified ospf6d to support both [Cha05] and [OS05]. See 2 for more details on the items implemented. When similar capabilities were found in both drafts the code was used by both MANET interfaces.

B.5 Design Changes to GTNetS 802.11 Radio Model

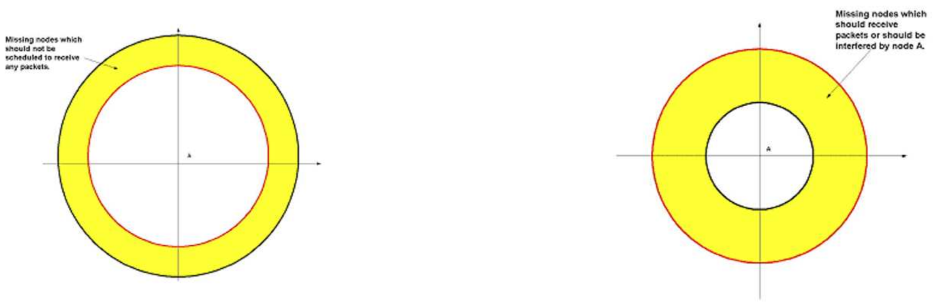
We have identified and fixed a discrepancy between the "radio range" as used by the Channel model and by the Radio/MAC model. In order to illustrate the problem, let us first review how a packet is sent from a node.

First, it builds a list of interfaces (wlan.cc:941[2] `node->BuildRadioInterfaceList()`), where it compares the distance between this node and all the other nodes with the radio range (node-real.cc:885,905,912,977). It also uses the physical distance between two nodes in the RadioItem, which is used later for channel model input. The radio range can be set by Node::SetRadioRange().

Second, it loops through the above list and compares with received power against two thresholds: csThreshold (wlan.cc:154) and rxThreshold (wlan.cc:158). It first decides whether to schedule a receiving event for that neighbor or not. If the power is greater than csThreshold, it then decides whether to mark this packet with an error or not based on the power value against rxThreshold. The received power calculation is from the channel model. For 802.11, it uses two-way. In function `double PropaTwoRay::Pr()` (propagation.cc:50), it uses default transmission power to calculate the receiving power.

Therefore, there exists discrepancy between channel and radio/MAC in terms of the view of radio range. So the following two cases will happen as illustrated by Figure B.4.

[2] All line numbers are referring to the source code of GTNetS dated 082104.


Missing nodes which should not be scheduled to receive any packets.
A
Missing nodes which should receive packets or should be interfered by node A.
A

(a) Radio range set greater than real range

(b) Radio range set less than real range

Figure B.4: Discrepancy between channel and radio/MAC view of radio range parameter.

To make minimum changes, we assume the radio range used by `BuildRadioInterfaceList()` is correct and adjust Tx power in propagation calculations based on the range and desired receiving power at the range. If we would like to continue to use on/off radio model, i.e., when distance between two nodes is less or equal to range, the packet is scheduled to receive (otherwise, the packet is not scheduled for a receiver), we set the power to be used before calling the propagation power calculations such that at the radio range the received power is rxThreshold. To preserve the same calculation efficiency, the power, path loss and crossover distance is cached.

The proposed fixes are quick and dirty. The right fixes would be to remove set range interface and replace it with set transmission power interface. In the BuildRadioInterfaceList(), the propagation model should be consulted to determine the lists. It should use something like propagation limits to improve the calculation efficiency. In the real world, the sphere/circle range model is never a good model for radio. The radios will behavior more like Figure B.5. The right channel models could deal with this problem.

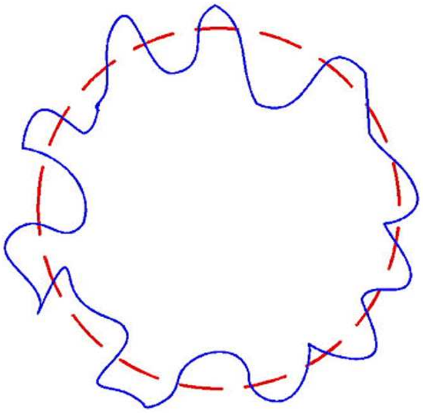


Figure B.5: Idealized (red dashed) vs. real (blue solid) models for radio range.

Appendix C

Overlapping Relays Implementation

This chapter steps through the implementation and validation of `draft-chandra-ospf-manet-ext-03.txt`.

This description is oriented towards the use of this code as simulation, but most of the comments equally apply to the implementation.

C.1 Summary of Overlapping Relays

The basic idea of overlapping relays can be understood by reading Cisco's IETF-60 presentation available at: http://www.ietf.org/proceedings/04aug/slides/ospf-4/index.html

To summarize briefly, there are a few key design aspects:

- **Source-Dependent Multi-Point Relays** To optimize the flooding forwarding behavior, rather than have all neighbors reflood a new LSA, the proposal borrows from the idea and heuristic found in OLSR, RFC 3626. Namely, each node selects a subset of its one-hop neighborhood to cover its two-hop neighborhood. These are called "active relays".

- **Use of Router LSA to discover two-hop neighborhood**

- **Backing up active relays** All one-hop neighbors of a source that are not active relays are inactive or backup relays. They only reflood packets if they do not hear an acknowledgement in a timely fashion from one of the sender's two-hop neighbors (for which it is also a one-hop neighbor). A configurable parameter called "PushbackInterval" is defined to allow an opportunity for a one-hop neighbor of the active relay to respond with an Ack or a reflood. Jitter is applied to the pushback waiting time so that, hopefully, only one of the possible non-active relays backs up the active relay. The non-active relays serve as a backup for the reliable flooding mechanism of the active relays (which still put an LSA on a retransmission list as in normal OSPF).

- **Intelligent Acknowledgments** Like in OSPF, all LSAs need to be acked by one-hop neighbors. All ACKs are multicast. Reflooding of an LSA can serve as an implicit acknowledgment. Acks are coalesced by waiting for AckInterval (randomized with some jitter) before sending. ACKs can also serve as a surrogate for a Hello packet (with no state change).

- **Incremental Hellos** To reduce the ammount of overhead created by hello broadcasts, incremental hellos are sent. Incremental hellos are different from standard hellos in that they only contain the changes in neighbor state between HelloInterval seconds. Each incremental hello is sent periodically and it contains a sequence number that identifies the state of the neighbor. If full neighbor state cannot be attained incrementally then a request followed by an answer of full state can be executed.

C.1.1 Configuring Overlapping Relays

Configuration is relatively straightforward. There are only a few options specific to Overlapping Relays:

```
PushbackInterval   : Overlapping Relays timer, default: 2000 msec
AckInterval        : Overlapping Relays timer, default: 1800 msec
diff_hellos        : enable Incremental Hellos, default: off
```

The PushbackInterval and AckInterval have dependencies on the RxmtInterval; the defaults are based on the default value of RxmtInterval (5 seconds); see the Chandra draft. Incremental Hellos can either be enabled or disabled. See the next section for the compile and runtime options.

C.2 Compiling and Running Overlapping Relays

Compiling

Use the provided Makefile (with most preprocessor defines enabled by default). The code that is specific to the Overlapping Relays implementation is mostly found in the source code (directory `SRC/ospf6d/`) by grepping for `OSPF6_MANET` and `OSPF6_MANET_MPR_FLOOD`.

There is one related define that is not enabled by default: `OSPF6_MANET_TEMPORARY_LSDB`. This is discussed below in C.3.3.

Invoking at Runtime

There are a number of relevant runtime options. For example, build the `ietf-dbg` tree and cd into `ietf/random_waypoint_manet`. Try running the following command to obtain usage information:

```
# ./random_waypoint_manet-opt -help
Usage:  ./random_waypoint_manet-opt
Basic Config:
       [seed=value] [tag=string]
       [start_time=value] [stop_time=value]
       [trace] [etrace] [ospfv3log]
Data Traffic Config:
       [pktsize=value] [pktrate=value]
Scenario Config:
       [num_nodes=value] [radio_range=value]
       [alpha=value] [grid_length=value]
       [pause_time=value] [velocity=value]
MAC Config:
       [mac_protocol=value] [alpha=value]
OSPFv3 Config:
       [wireless_interface=value]
       [wireless_flooding=value]
       [HelloInterval=value]
       [DeadInterval=value]
       [RxmtInterval=value]
       [MinLSInterval=value]
       [MinLSArrival=value]
MPR MANET Interface Config:
       [PushbackInterval=value]
       [AckInterval=value]
       [diff_hellos]
MDR MANET Interface Config:
       [BackupWaitInterval=value]
       [AckInterval=value]
       [TwoHopRefresh=value]
       [RouterDeadCount=value]
       [AdjConnectivity=value]
       [LSAFullness=value]
       [diff_hellos]
       [NonPersistentMDR]
```

The next page goes through the default values for each of these options, and describes how to configure the simulation script to use Overlapping Relays and various options.

```
Basic Config:
     seed             : random seed, default: 1
     tag              : unique string to append to trace/statfile
     start_time       : statistics discarded before this time, default: 1800 sec
     stop_time        : simulation stop time, default: 3600 sec
     trace            : enable per-packet tracing, default: off
     etrace           : enable extended tracing, default: off
     ospfv3log        : enable ospfv3 logging, default: off

Data Traffic Config:
     pktsize          : udp packet size default: 40 bytes
     pktrate          : udp pkts across whole network, default: 10 pkts/sec

Scenario Config:
     num_nodes          : number of nodes, default: 20
     radio_range:       : max range of radio, default: 250 m
     wireless link alpha: (0,1), default: 0.5
     grid_length        : length of square grid, default: 500 m
     velocity           : speed of movement, default: 10 m/s
     pause_time         : duration of pause after movement, default: 40 sec

SPFv3 Config:
     wireless_interface : type of interface ptmp=1, manet=2, default: 1
     wireless_flooding  : flooding broadast=0, mpr=1, mdr=2, default: 1

     HelloInterval      : ospf6 HelloInterval, default: 10 sec
     DeadInterval       : ospf6 DeadInterval, default: 40 sec
     RxmtInterval       : ospf6 RxmtInterval, default: 5 sec
     MinLSInterval      : ospf6 MinLSInterval, default: 5 sec
     MinLSArrival       : ospf6 MinLSArrival, default: 1 sec

MPR MANET Interface Config:
     PushbackInterval   : Overlapping Relays timer, default: 2000 msec
     AckInterval        : Overlapping Relays timer, default: 1800 msec
     diff_hellos        : enable Incremental Hellos, default: off

MDR MANET Interface Config:
     BackupWaitInterval : MDR BackupWaitInterval, default: 1000 msec
     AckInterval        : MDR AckInterval, default: 1800 msec
     TwoHopRefresh      : MDR TwoHopRefresh Hellos, default: 3 Hellos
     RouterDeadCount    : MDR RouterDeadCount, default: 3
     AdjConnectivity    : 0=Fully, 1=Uniconnected, 2=Biconnected, default: 2
     LSAFullness        : 0=only adjacent neighbors, 1=min-hop routing,
                          2=full LSAs from MDR/MBDRs, 3=full LSAs, default: 0
     diff_hellos        : enable MDR Differential Hellos, default: off
     NonPersistentMDR   : disable MDR persistent mdr/mbdrs,  default: off
```

To run with Overlapping Relays, the following non-default options are required: `wireless_interface=2`, `wireless_flooding=1`. To configure Incremental Hellos, enable the `diff_hellos` option. Finally, to configure specific values for PushbackInterval and AckInterval, configure `PushbackInterval=value`, `AckInterval=value` (values in ms).

C.3 Implementation Description

The current implementation is of Cisco draft version 03 except as noted below:

Packet Formats: The ’Request From’ TLV is packet type 6 and the ’Full State For’ TLV is packet type 7.

Neighbor Adjacencies Sec 3.3.6: Compliant with draft 03.

Sending Hellos Sec 3.3.7: Compliant with draft 03.

Receiving Hellos Sec 3.3.8: Full State TLVs are not processed or sent.

Interoperablity Sec 3.3.9: Compliant with draft 03.

Graceful Restart Sec 3.3.10: Compliant with draft 03.

Optimized Flooding Sec 3.4.1-8: The A and N bits in the relay TLV are not used in the relay selection algorithm. Willingness is not used in the selection algorithm.

Intelligent Acknowledgements Sec 3.4.9: Compliant with draft 03.

Important Timers Sec 3.4.10: Compliant with draft 03.

C.3.1 Incremental Hellos

The following cross-reference to the code is provided:

Packet Formats (Sec 3.3.1 through 3.3.5)

- The LLS datablock is defined in `ospf6_message.h` (grep for "Chandra03 3.1.2").
- The TLV header is defined in `ospf6_message.h` (grep for "Chandra03 3.1.3").
- The extended options TLV is defined in `ospf6_message.h` (grep for "Chandra03 3.1.4").
- The OSPFv3 options field are extended with new options bits: F, I, L, and AF. These extensions are defined in `ospf6_proto.h` (grep for "Chandra03 3.5").
- The SCS TLV is defined in `ospf6_message.h` (grep for "Chandra03 3.3.2"), and the options bits are defined in `ospf6_message.h` (grep for "Chandra03 3.3.2").
- The drop neighbor, Request From, and Full State For TLVs do need a struct, since they are just a list of router ids. Note: The 'Request From' TLV is packet type 6. Note: The 'Full State For' TLV is packet type 7.

Neighbor Adjacencies (Sec 3.3.6)

Building Neighbor Adjacencies: All hellos are sent periodically. The I bit is sent on all incremental hellos in `ospf6_message.c` (grep for "Chandra03 3.3.6.1 paragraph 1"). Hellos are sometimes suppressed if LSAck packets are transmitted in `ospf6_message.c` (grep for "Chandra03 3.3.6.1 paragraph 1").

Receiving a hello with the I bit set is performed by `ospf6_mpr_diff_mhello_recv()` in `ospf6_message.c` (grep for "Chandra03 3.3.6.1 paragraph 2").

- Bullet 1: New neighors are placed on the neighbor list in `ospf6_message.c` (grep for "Chandra03 3.3.6.1 paragraph 2 bullet 1").
- Bullet 2: The SCS number is set to be incremented upon the next send in `ospf6_neighbor.c` (grep for "Chandra03 3.3.6.1 paragraph 2 bullet 2").
- Bullet 3: The neigbors are removed from the neighbor list in `ospf6_neighbor.c` (grep for "Chandra03 3.3.6.1 paragraph 2 bullet 3").
- Bullet 4: Incremental hellos on broadcast links are not implemented.

Adjacency Failure:

- Bullet 1: Neighbors are brought down by calling `ospf6_neighbor_delete()` in `ospf6_neighbor.c`.
- Bullet 2: The neighbor is added to the drop neighbor list in `ospf6_neighbor.c` (grep for "Chandra03 3.3.6.2 paragraph 1 bullet 2").
- Bullet 3: The SCS number is incremented in `ospf6_neighbor.c` (grep for "Chandra03 3.3.6.2 paragraph 1 bullet 3").
- Bullet 4: The dropped neighor is sent for the next DeadInterval secs in `ospf6_message.c` (grep for "Chandra03 3.3.6.2 paragraph 1 bullet 4"). The N bit is set in `ospf6_message.c` (grep for "Chandra03 3.3.6.2 paragraph 1 bullet 4").

Sending Hellos (Sec 3.3.7)

Hellos are sent periodically using the function `ospf6_mpr_mhello_send()` in `ospf6_message.c` or `ospf6_mpr_diff_mhello_send()` in `ospf6_message.c` if incremental hellos are enabled.

- Paragraph 1: Upon intilization a multicast request is sent in `ospf6_message.c` (grep for "Chandra03 3.3.7 paragraph 1").
- Paragraph 2: If a router receives a hello from a neighbor in state INIT, a request is scheduled in `ospf6_message.c` (grep for "Chandra03 3.3.7 paragraph 2").
- Paragraph 3: Each hello with an incremeneted SCS number contains the full state associated with the SCS number (see `ospf6_mpr_diff_mhello_send()`).

Receiving Hellos (Sec 3.3.8)

- Paragraph 1: A neighbors SCS number is kept in `struct ospf6_neighbor` in `ospf6_neighbor.h` (grep for "Chandra03 3.3.8 paragraph 1").
- Paragraph 2: A device sends a request to its neighbor when it receives an SCS number less than the recorded value in `ospf6_message.c` (grep for "Chandra03 3.3.8 paragraph 2").
- Paragraph 3: If the SCS is a wrap around or one greater it is handled in `ospf6_message.c` (grep for "Chandra03 3.3.8 paragraph 3").
  - Bullet 1: The SCS number of this neighbor is incremeted if neighors are found in its neighbor list in `ospf6_message.c` (grep for "Chandra03 3.3.8 paragraph 3 bullet 1").
  - Bullet 2: The SCS number of this neighbor is incremeted if neighors are found in its drop neighbor list in `ospf6_message.c` (grep for "Chandra03 3.3.8 paragraph 3 bullet 2").
  - Bullet 3: If the local router is found in the drop neighbor list then the neigbor is placed in One-Way in `ospf6_message.c` (grep for "Chandra03 3.3.8 paragraph 3 bullet 3"). Note: The draft says to destroy the neighbor.
  - Bullet 5: If no state change is found, a request is scheduled in `ospf6_message.c` (grep for "Chandra03 3.3.8 paragraph 3 bullet 5").
- Paragraph 4: A request is sent to the neighbor if the received SCS number is greater than 1 plus the current SCS number in `ospf6_message.c:4665` (grep for "Chandra03 3.3.8 paragraph 4").

Receiving Hellos with the N Bit: A device schedules a request if it receives a hello with the N bit set, and it has not received the reported SCS number in `ospf6_message.c` (grep for "Chandra03 3.3.8.1").

Receiving Hellos with the R Bit: When a device receives a hello with the R bit set without a Request TLV or the R bit set with a Request TLV with its router id included, it schedules a full state answer in `ospf6_message.c` (grep for "Chandra03 3.3.8.2"). The answer will be multicast, and it is created in `ospf6_mpr_diff_mhello_send()`.

Receiving Hellos with the FS Bit: When a device receives a hello with the FS bit set.

- Bullet 1: it ignores the TLVs if the reported SCS number is equal to the stored SCS number at `ospf6_message.c` (grep for "Chandra03 3.3.8.3 bullet 1").
- Bullet 2: if the SCS number is different than the last SCS number then the hello is processed in `ospf6_message.c` (grep for "Chandra03 3.3.8.3 paragraph 1 bullet 2").
- Bullet 3: Not Implemented

Interoperability (Sec 3.3.9)

Reception of hello with the I bit set is handled in `ospf6_message.c` (grep for "Chandra03 3.3.9").

OSPF Graceful Restart (Sec 3.3.10)

Graceful restart checks if RouterDeadInterval has passed in `ospf6_message.c` (grep for "Chandra03 3.3.10 paragraph 2 bullet 1"). If it has not passed a normal hello is sent. If it has passed and it is first hello, then a full state hello is sent in `ospf6_message.c` (grep for "Chandra03 3.3.10 paragraph 2 bullet 3").

C.3.2 Optimized Flooding

The following cross-reference to the code is provided:

Packet Formats (Sec 3.4.5 through 3.4.7)

The F, I, and L bits are supported.

The active overlapping relays TLV (Sec. 3.4.6) is defined in `ospf6_message.h` (grep for "Chandra03 3.4.6").

The Willingness TLV (Sec. 3.4.7) is defined in `ospf6_message.h` (grep for "Chandra03 3.4.7"). However, it is never used by the sender and it is ignored except for debugging output at the receiver.

Overlapping Relay Discovery Process (Sec 3.4.2, 3.4.4)

The first paragraph of Section 3.4.2 is implemented in `ospf6_flood.c`(grep for "Chandra03 3.4.2 paragraph 1"). If you receive a new LSA, update your two-hop neighbor list accordingly. These functions are defined in `ospf6_neighbor.c` and include:

- `ospf6_update_2hop_neighbor_list()`
- `ospf6_2hop_neighbor_lookup()`
- `ospf6_add_2hop_neighbor()`
- `ospf6_2hop_list_delete()`
- `ospf6_2hop_neighbor_delete()`
- `ospf6_update_neighborhood()`

In paragraph 2 of Section 3.4.2, The ’N’ bit is not being processed in the relay selection algorithm.

Section 3.4.4 (MPR selection heuristic) is implemented in the source file `ospf6_mpr.c`. Specifically, the function `ospf6_calculate_relays()` implements Section 3.4.4. The ’A’ and ’N’ bits are not being processed and willingness is not honored.

Flooding Procedure (Sec 3.4.8, 3.4.10)

The function `ospf6_flood_interface()`, found in `ospf6_flood.c`, floods an LSA out of an interface. The function `ospf6_flood_interface_mpr()` is called if flooding is based on MPRs. This function is defined in `ospf6_flood.c` (grep for "Chandra03 3.4.8").

Section 3.4.8 is primarily implemented in the following places:

- Condition 1 of section 3.4.8 is handled in `ospf6_flood.c` (grep for "Chandra03 3.4.8 paragraph 2 condition 1").
- Condition 2 of section 3.4.8 is handled in the pushback code described below. In addition, a piece of this condition (condition 2.2) is also handled in `ospf6_flood.c` (grep for Chandra03 3.4.8 paragraph 2 condition 2.2).
- Condition 3 of section 3.4.8 is handled in `ospf6_flood.c` (grep for Chandra03 3.4.8 paragraph 2 condition 3).

The procedures for handling LSA Pushback are found in `ospf6_top.c`. These include the functions:

- `ospf6_pushback_lsa_add()`

- `ospf6_pushback_lsa_neighbor_delete()`
- `ospf6_pushback_lsa_delete()`
- `ospf6_pushback_check_coverage()`
- `ospf6_pushback_expiration()`
- `ospf6_refresh_lsa_pushback_list()`
- `pushback_jitter()`

Note that the **PushbackInterval** and **AckInterval** are both configurable at the command line. By default, we use a value of 40% of the RxmtInterval for PushbackInterval, and 90% of PushbackInterval for AckInterval. For example, if RxmtInterval is 5 seconds, PushbackInterval is 2 seconds and AckInterval is 1.8 seconds. The defaults are coded in `args.cc`. If you decide to change RxmtInterval, make sure to change PushbackInterval and AckInterval appropriately (they don't scale automatically).

Intelligent Acknowledgments (Sec 3.4.9, 3.4.10)

The conditions of Section 3.4.9 are found in the following:

- Condition 1 (all ACKs multicast) is implemented with `ospf6_lsack_send_interface()` in `ospf6_flood.c` (grep for Chandra03 3.4.9 paragraph 1 condition 1).
- Condition 2 (MUST only expect either explicit or implicit acknowledgments from peers that have not previously acked) is implemented in `ospf6_flood.c` (grep for Chandra03 3.4.9 paragraph 1 condition 2).
- Condition 3 (only acknowledge first update) is implemented in `ospf6_flood.c` (grep for Chandra03 3.4.9 paragraph 1 condition 3).
- Condition 4 (suppress explicit acknowledgements of relays) is implemented in `ospf6_flood.c` (grep for Chandra03 3.4.9 paragraph 1 condition 4).
- Condition 5 (bundling of ACKs and waiting AckInterval) is implemented with `ospf6_lsack_send_interface()` in `ospf6_message.c`.
- Condition 6 (resetting RouterDeadInterval and HelloInterval) is implemented in `ospf6_message.c` (grep for Chandra03 3.4.9 paragraph 1 condition 6).
- Condition 7 (all unicast LSAs explicitly acknowledged via multicast) is implemented in `ospf6_flood.c` (grep for Chandra03 3.4.9 paragraph 1 condition 7).
- The last paragraph of this section (suppression of transmissions due to ACKs received from a non-overlapping neighbor) is implemented in `ospf6_message.c` (grep for Chandra03 3.4.9 paragraph 2).

C.3.3 Temporary LSDB

We implemented the Temporary LSDB idea (proactive caching of snooped LSAs); the code has been implemented in the places tagged by `OSPF6_MANET_TEMPORARY_LSDB`. As you may see, this modification was somewhat intrusive into the quagga code. However, upon earlier testing of this feature, it did not seem to cause significant (or even noticeable) improvement.

Since early returns were uninspiring, we commented out this code in our Makefile, and haven't rerun it since. Since that has been many months ago, we cannot guarantee that it will work or work properly if enabled now (i.e., would probably require some tweaking and validation).

C.3.4 Deltas from the Internet Draft

Additions to Cisco's Draft

We noticed a large number of LSU unicasts due to responding to stale (outdated) LSAs. This event occurs when a stale LSA is received by a node that has a newer version of the same LSA. RFC 2328 13 (8) states that the receiving node should unicast its database copy to the sending node. This event occured often when a router "pushed back" an LSA according to the MANET interface rules for non-active relays when it would have normally flooded the LSA. With the new MANET interface, during the LSA pushback time, the router would frequently receive the stale LSA and respond according to the RFC. We changed this mechanism to not send the unicast LSA when the LSA was being pushed back because it will eventually be sent if all neighbors do not ack it. This code is found in `ospf6_flood.c` (grep for BOEING Draft Change).

As mentioned above, we maid a slight change to the procedure for receiving Hellos with the A Bit, as described above in C.3.1.

The type values for the Request and Full TLVs were incorrect, so we chose type 6 and 7 respectively. See C.3.1.

Unimplemented Features

- Incremental hellos on broadcast links are not implemented (Sec 3.3.4.1).
- Functional use of the A bit in Relays (Sec. 3.4.2, 3.4.4, 3.4.8).
- Functional use of the N bit in Relays (Sec. 3.4.2, 3.4.4, 3.4.8).
- Functional use of Willingness TLV (Sec. 3.4 7).
- Enabling this operation on all interfaces (Sec. 3.4.11).
- Detecting existence of Designated Routers and avoiding them (Sec. 3.4.11).

C.4 Validation

This section shows some validation of the operation of Cisco's overlapping relays algorithm. The detailed trace and stat files can be found in the `validation/relays/` under the specified test directory. Default command line parameters are used unless otherwise specificed under the scenario description. All figure symbols in this section can be identified by looking at the key in Figure C.1. See A.1.5 for more about the validation test suite.

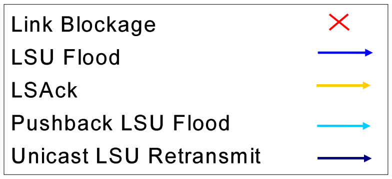


Figure C.1: Key used for figure symbols

C.4.1 MPR Flooding

This scenario (`validation/relays/basic/`) is executed for 2000 seconds of simulation time. The first 1800 seconds are used for initialization, to get the network into a relatively quiescent state. The topology of the scenario is seen in Figure C.2. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600.

- **Step A:** At time 1900 the link is broken between nodes 4 and 5. After the neighbor relationship between nodes 4 and 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.

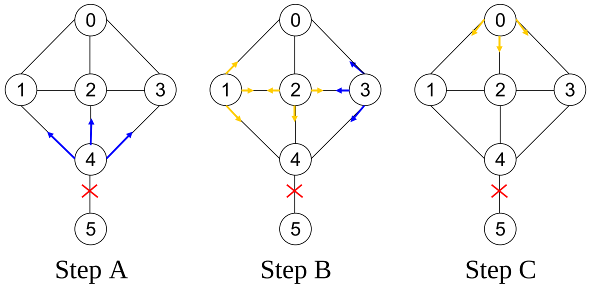


Figure C.2: MPR flooding validation

- **Step B:** With a point-to-multipoint interface, nodes 1, 2, and 3 would have forwarded the packet, but this scenario demonstrates that MPR flooding only requrires node 3 to forward the LSU. Because node 3 is the only node to forward the packet nodes 1 and 2 acknowledge the LSA with an LSAck.
- **Step C:** Node 0 acks the LSA from node 3 to finish the exchange.

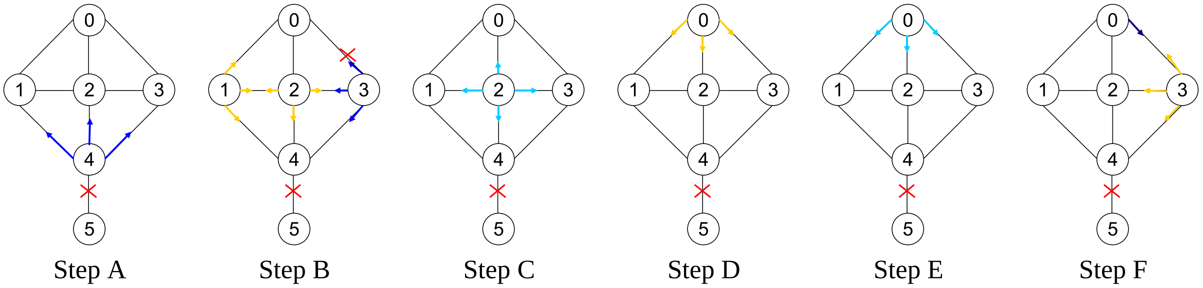


Figure C.3: Pushback due to single LSU loss

C.4.2 Pushback Operation

This sections explains the validation of the pushback technique (Cisco draft sections 3.8-3.10) that allows non-active relays to become active.

Pushback Due to Single LSU Loss

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure C.3. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600.

- **Step A:** At time 1900 the link is broken between nodes 4 and 5. After the neighbor relationship between 4 & 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.
- **Step B:** Nodes 1 and 2 ack the LSA from node 4. The MPR or relay node that floods the LSU from node 4 is node 3. The LSU does not reach node 0 when it is flooded from node 3.
- **Step C:** The LSA is sent out by node 2 when the pushback timer expires. The reception of this LSA at node 1 forces its pushback timer to be restarted.
- **Step D:** When node 0 receives the LSA it then acknowledges it.
- **Step E:** This LSA has now reached all nodes, but node 0 does not know that node 3 has received this LSA (node 0 is a non-active overlapping relay for LSAs sent from node 2), so it pushes back the LSA and then sends it upon pushback expiration.
- **Step F:** Node 3 does not respond to the LSA because it previously implicitly acked the LSA, so node 0 unicast retransmits the LSA to node 3 after the RxmtInterval expires. Node 3 finally acknowledges the reception to node 0.

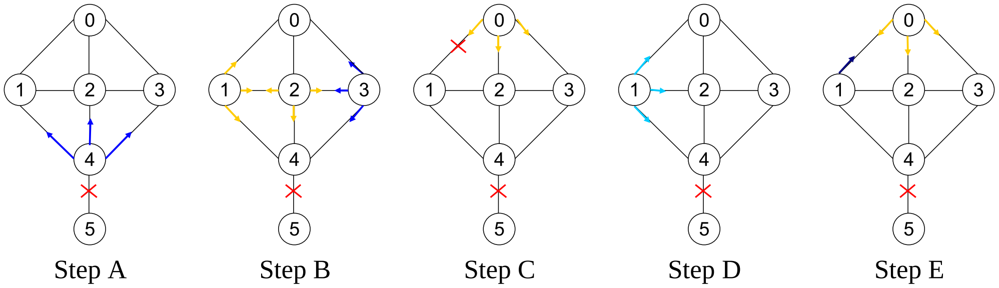


Figure C.4: Pushback due to single ack loss

Pushback Due to Single Ack Loss

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure C.4. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600.

- **Step A:** At time 1900 the link is broken between nodes 4 and 5. After the neighbor relationship between nodes 4 and 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.
- **Step B:** Nodes 1 and 2 ack the LSA from node 4. The MPR or relay node that floods the LSU from node 4 is node 3.
- **Step C:** After node 0 receives the LSA, it sends an LSAck to its neighbors. The LSAck is never received by node 1.
- **Step D:** After pushback interval, node 1 floods node's 4 router LSA because it never received an ack from node 0. Node 0 receives the LSA, but it does not respond because it has previously acknowleded this LSA.
- **Step E:** After node 1's rxmtInterval, the LSA is unicast sent to node 0. Node 0 then responds with an ack.

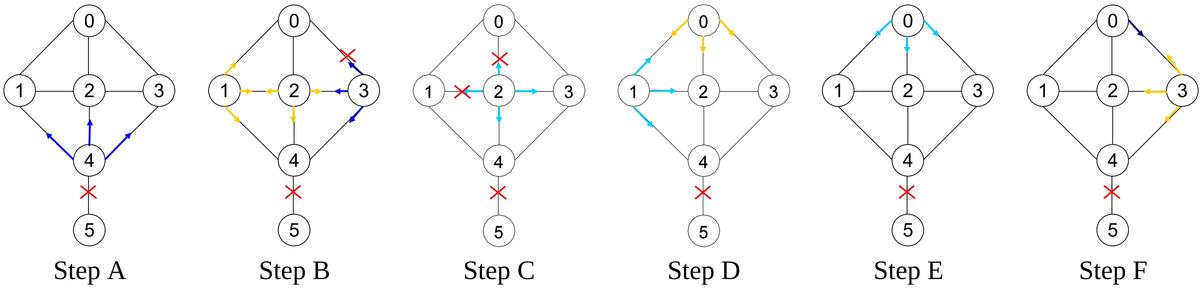


Figure C.5: Pushback due to LSU losses

Pushback Due to LSU Losses

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure C.5. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600.

- **Step A:** At time 1900 the link is broken between nodes 4 and 5. After the neighbor relationship between 4 and 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.
- **Step B:** Nodes 1 and 2 ack the LSA from node 4. The MPR or relay node that floods the LSU from node 4 is node 3. The LSU does not reach node 0 when it is flooded from node 3.
- **Step C:** The LSA is sent out by node 2 when the pushback timer expires. The LSA is not received by node 0. The reception of this LSA at node 1 forces the pushback timer to be restarted.
- **Step D:** After node 1's pushback timer expires the LSA is sent. The reception of the LSA at node 0 cause node 0 to ack the LSA.
- **Step E:** This LSA has now reached all nodes, but node 0 does not know that node 3 has received this LSA, so it pushes back the LSA and then sends it upon pushback expiration.
- **Step F:** Node 3 does not respond to the LSA because it previously implicitly acked the LSA, so node 0 unicast retransmits the LSA to node 3 after the RxmtInterval expires. Node 3 finally acknowledges the reception to node 0.

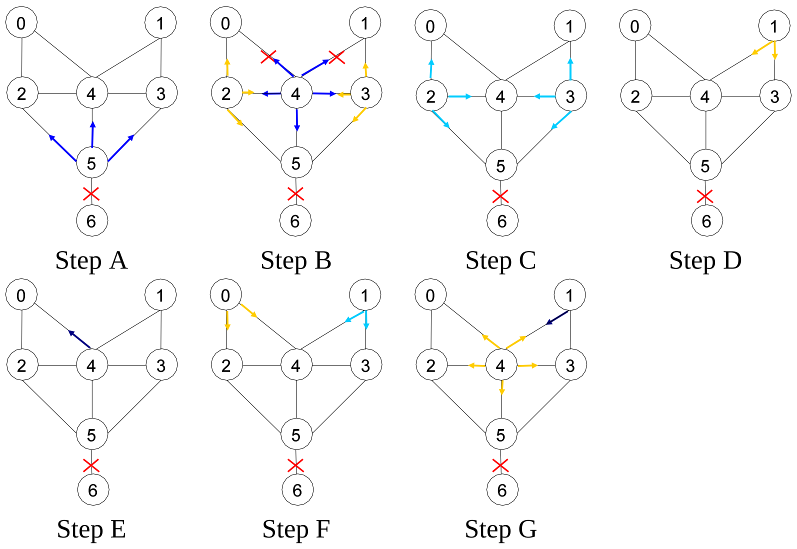


Figure C.6: Two Pushbacks due LSU loss

Two Pushbacks Due to LSU Loss

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure C.6. The following command line parameters were changed from the default values: nNodes = 7 and wireless_interface = 2.

- **Step A:** At time 1900 the link is broken between nodes 5 and 6. After the neighbor relationship between 5 and 6 expires, a new router LSA is originated by node 5. This LSA is flooded through out the network in an LSU.
- **Step B:** Nodes 2 and 3 ack the LSA from node 5. The MPR or relay node that floods the LSU from node 5 is node 4. The LSU does not reach node 0 and 1 when it is flooded from node 4.
- **Step C:** The LSA is sent out by nodes 2 and 3 when their pushback timers expire. The LSA is received by nodes 0 and 1.
- **Step D:** Node 1 acks the LSA.
- **Step E:** Node 4's retransmit timers expires for node 0 because node 0 has not acked the LSA.
- **Step F:** Node 0 acknowedges the retansmission from node 4. Node 1's pushback timer expires because it did not receive an ack from node 4. Node 1 floods the LSA. Node 4 does not respond because it previously implicitly acked the LSA.
- **Step G:** Node 1 retansmits the LSA. Node 4 acknowledges the retransmission of the LSA.

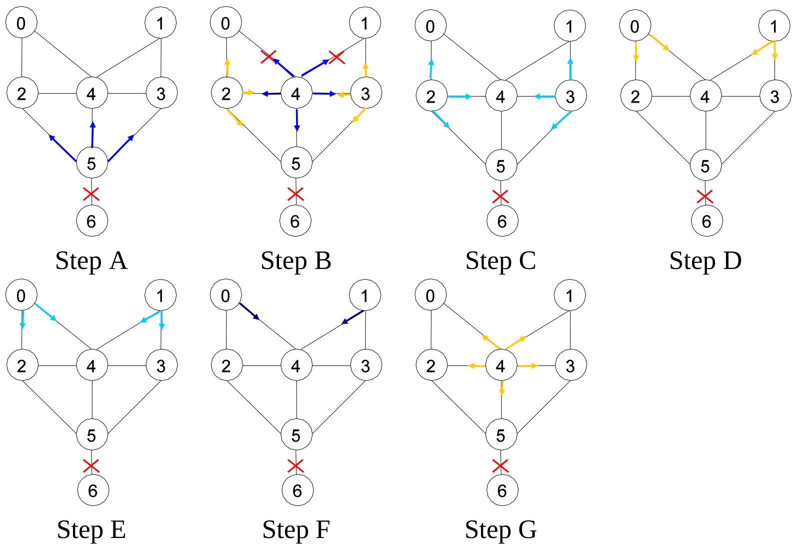


Figure C.7: Two Pushbacks due LSU loss

Two Pushbacks Due to LSU Loss with AckInterval = 1 sec

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure C.7. The following command line parameters were changed from the default values: nNodes = 7, wireless_interface = 2, and AckInterval = 1000.

- **Step A:** At time 1900 the link is broken between nodes 5 and 6. After the neighbor relationship between 5 and 6 expires, a new router LSA is originated by node 5. This LSA is flooded through out the network in an LSU.
- **Step B:** Nodes 2 and 3 ack the LSA from node 5. The MPR or relay node that floods the LSU from node 5 is node 4. The LSU does not reach node 0 and 1 when it is flooded from node 4.
- **Step C:** The LSA is sent out by nodes 2 and 3 when their pushback timers expire. The LSA is received by nodes 0 and 1.
- **Step D:** Nodes 0 and 1 ack the LSA.
- **Step E:** Nodes 0 and 1 send the LSA to node 4 due to pushback expiration, but node 4 does not ack because it had previously implicitly acked.
- **Step F:** Nodes 0 and 1 retransmit the LSA to node 4.
- **Step G:** Node 4 acknowledges the retransmissions of the LSA.

C.4.3 Additional Validation Scenarios

Additional validation scenarios can be found in `validation/relays/` that are not include here. The scenarios validate incremental hellos and flooding with multiple LSAs.

Appendix D

MANET Designated Routers Implementation

This chapter steps through the implementation and validation of `draft-ogier-manet-ospf-extension-03.txt` and the various modifications implemented since that draft was published.

This description is oriented towards the use of this code as simulation, but most of the comments equally apply to the implementation.

D.1 Summary of MANET Designated Routers

The basic idea of MANET Designated Routers can be understood by reading the below text, and also Richard Ogier's IETF-62 presentation available at:
http://www3.ietf.org/proceedings/05mar/slides/ospf-5/sld1.htm

Note: Richard Ogier contributed the summary material in the remainder of this section.

To optimize the flooding procedure, rather than have every router flood each received new LSA, each router decides, based on 2-hop neighbor information, whether it belongs to a CDS that is responsible for forwarding/flooding each new LSA. The CDS is called "source-independent" because the same CDS is used regardless of the LSA's origin.

The CDS consists of MANET Designated Routers (MDRs) and Backup MDRs (BMDRs). The MDRs by themselves form a CDS, and the MDRs and BMDRs together form a biconnected CDS to provide redundancy and robustness. The purpose of MDRs and BMDRs in a MANET is similar to the purpose of the DR and BDR in an OSPF broadcast network: to reduce the number of routers that must flood each LSA, and to reduce the number of adjacencies.

Each router decides whether it is an MDR, BMDR, or MDR Other (neither MDR nor BMDR) based on 2-hop neighbor information which is learned from Hellos, as summarized below. The CDS algorithm used for selecting MDRs and BMDRs is a generalization of the DR/BDR selection algorithm of OSPF, and gives priority first to routers with larger MDR level (for persistence and stability of MDRs), then to routers with larger Router Priority, and finally to routers with larger Router ID (to break ties). In the simulation, all routers have the same Router Priority.

D.1.1 Flooding via MDRs and BDMRs

- Each MDR forwards each new LSA the first time it is received.
- Each BMDR waits a short interval (BackupWaitInterval), and then floods the LSA only if some neighbor has not ACKed the LSA and is not covered by some neighbor from which the LSA has been received.
- MDR Other routers do not forward LSAs.
- To exploit the broadcast nature of MANETs, a new LSA is processed (and possibly forwarded) if it is received from any neighbor in state 2-Way or greater.

D.1.2 Adjacencies

Rather than have each router form adjacencies with all of its neighbors, each MDR/BMDR forms adjacencies with a subset of its MDR/BMDR neighbors to form a biconnected backbone, and each MDR Other forms adjacencies with two selected MDR/BMDR neighbors called "parents", thus providing a biconnected subgraph of adjacencies.

The parent selection is persistent, i.e., a router updates its parents only when necessary. The two parents are indicated in the DR and BDR fields of each Hello.

The persistence of the MDRs and BMDRs, combined with the persistence of the parent selection, maximizes the stability (lifetime) of the adjacencies.

Once two neighbors become adjacent, they remain adjacent as long as they have 2-way communication and at least one of them is an MDR or BMDR. Since this condition is weaker than the condition for forming an adjacency, it provides hysteresis for additional stability.

An option is provided to have the adjacencies form only a 1-connected subgraph instead of a biconnected subgraph, in order to reduce overhead and allow scalability to larger networks. This option typically results in a slightly lower deliver ratio, due to some loss of robustness.

D.1.3 Link State ACKs

All Link State ACKs are multicast. An LSA received as a multicast is ACKed only the first time it is received.

An LSA that is flooded back on the same interface is treated as an implicit ACK. Link State ACKs may be delayed up to AckInterval msec to allow coalescing multiple ACKs in the same packet. The only exception is that MDR/BDRs send a multicast ACK immediately when a duplicate LSA is received as a unicast (to prevent additional retransmissions).

Only ACKs from adjacent neighbors are processed, and retransmitted LSAs are sent (via unicast) only to adjacent neighbors.

D.1.4 Partial and Full Topology LSAs

Unlike the DR of an OSPF broadcast network, an MDR does not originate a network LSA, since a network LSA cannot be used to describe the general topology of a MANET. Instead, each router advertises a subset of its MANET neighbors as point-to-point links in its Router LSA. The choice of which neighbors to advertise is flexible, and is determined by the configurable parameter LSAFullness.

As a minimum requirement, each router must advertise all of its fully adjacent neighbors in its router LSA. This minimum choice corresponds to LSAFullness = 0. This choice results in the minimum amount of LSA flooding overhead, but does not provide routing along shortest paths.

Setting LSAFullness = 1 provides min-hop routing. Each router decides which neighbors to include in its LSA by looking at the LSAs originated by its neighbors, and including in its LSA the minimum set of neighbors necessary to provide a 2-hop path (in each direction) between each pair of neighbors that are not neighbors of each other.

If LSAFullness = 2, then each MDR/BMDR originates a full LSA (as described below), while each MDR Other originates minimal LSAs. This choice provides routing along nearly min-hop paths, and typically results in less flooding overhead than LSAFullness = 1.

If LSAFullness = 3, then each router originates a full LSA, which includes all "routable" neighbors. A neighbor is considered to be routable if the SPF calculation produces a path to the neighbor. Note that a routable neighbor need not be adjacent. However, the routability condition implies the existence of a path to the neighbor via full adjacencies, thus providing some assurance of synchronization.

D.1.5 Shortest-Path Tree Calculation

The SPF calculation differs from RFC 2328 in that it allows any routable neighbor to be a next hop to a destination. We note, however, that RFC 2328 also allows a non-adjacent neighbor to be a next hop, if both routers are fully adjacent to the DR of a broadcast network. Allowing any routable neighbor to be a next hop is a generalization of this condition to multi-hop wireless networks.

D.1.6 Differential and Full Hellos

Hellos are used both for neighbor discovery and for advertising the set of neighbors that are 2-Way or greater, to be used by neighbors to learn 2-hop neighbor information.

Differential Hellos are sent every HelloInterval seconds, except when full Hellos are sent, which happens every TwoHopRefresh Hellos. The suggested values for HelloInterval and TwoHopRefresh are 2 seconds and 3 Hellos, respectively.

Differential Hellos are used to reduce overhead and to allow Hellos to be sent more frequently (for faster reaction to topology changes). Full Hellos are sent less frequently to ensure that all neighbors have current 2-hop neighbor information.

Each Hello contains a sequence number, which is incremented each time a Hello is sent on a given interface. As in OSPF, the state of a neighbor transitions to Down if no Hello is heard for RouterDeadInterval. In addition, the state of a neighbor transitions to Init if RouterDeadCount Hellos are missed, based on the Hello sequence number.

Both differential and full Hellos may contain a list of Heard Neighbors (in state Init) and a list of Reported Neighbors (in state 2-Way or above). In addition, differential Hellos may contain a list of Lost Neighbors (which recently transitioned to the Down state). A neighbor that transitions to a different one of these three categories is included in the appropriate list of the next RouterDeadCount Hellos. This ensures that the neighbor will either learn the new state within RouterDeadCount Hellos, or will declare the neighbor to be Down or Init.

D.1.7 Configuring Overlapping Relays

The following options exist:

```
BackupWaitInterval : MDR BackupWaitInterval, default: 1000 msec
AckInterval        : MDR AckInterval, default: 1800 msec
TwoHopRefresh      : MDR TwoHopRefresh Hellos, default: 3 Hellos
RouterDeadCount    : MDR RouterDeadCount, default: 3
AdjConnectivity    : 0=Fully, 1=Uniconnected, 2=Biconnected, default: 2
LSAFullness        : 0=only adjacent neighbors, 1=min-hop routing,
                     2=full LSAs from MDR/MBDRs, 3=full LSAs, default: 0
diff_hellos        : enable MDR Differential Hellos, default: off
NonPersistentMDR   : disable MDR persistent mdr/mbdrs,  default: off
```

All of these parameters are introduced in the above summary text, except for the last one. The parameter NonPersistentMDR can toggle whether MDR selection is sticky (persistent) or is recomputed for each topology change (as described in Ogier's draft).

D.2 Compiling and Running MDRs

Compiling

Use the provided Makefile (with most preprocessor defines enabled by default). The code that is specific to the MDRs implementation is mostly found in the source code (directory `SRC/ospf6d/`) by grepping for `OSPF6_MANET` and `OSPF6_MANET_MDR_FLOOD`.

Invoking at Runtime

There are a number of relevant runtime options. For example, build the `ietf-opt` or `ietf-dbg` tree and cd into `ietf/random_waypoint_manet`. Try running the following command to obtain usage information:

```
# ./random_waypoint_manet-opt -help
Usage:  ./random_waypoint_manet-opt
Basic Config:
       [seed=value] [tag=string]
       [start_time=value] [stop_time=value]
       [trace] [etrace] [ospfv3log]
Data Traffic Config:
       [pktsize=value] [pktrate=value]
Scenario Config:
       [num_nodes=value] [radio_range=value]
       [alpha=value] [grid_length=value]
       [pause_time=value] [velocity=value]
MAC Config:
       [mac_protocol=value] [alpha=value]
OSPFv3 Config:
       [wireless_interface=value]
       [wireless_flooding=value]
       [HelloInterval=value]
       [DeadInterval=value]
       [RxmtInterval=value]
       [MinLSInterval=value]
       [MinLSArrival=value]
MPR MANET Interface Config:
       [PushbackInterval=value]
       [AckInterval=value]
       [diff_hellos]
MDR MANET Interface Config:
       [BackupWaitInterval=value]
       [AckInterval=value]
       [TwoHopRefresh=value]
       [RouterDeadCount=value]
       [AdjConnectivity=value]
       [LSAFullness=value]
       [diff_hellos]
       [NonPersistentMDR]
```

The next page goes through the default values for each of these options, and describes how to configure the simulation script to use MDRs and various options.

```
############# Options Description ################
Basic Config:
     seed             : random seed, default: 1
     tag              : unique string to append to trace/statfile
     start_time       : statistics discarded before this time, default: 1800 sec
     stop_time        : simulation stop time, default: 3600 sec
     trace            : enable per-packet tracing, default: off
     etrace           : enable extended tracing, default: off
     ospfv3log        : enable ospfv3 logging, default: off

Data Traffic Config:
     pktsize          : udp packet size default: 40 bytes
     pktrate          : udp pkts across whole network, default: 10 pkts/sec

Scenario Config:
     num_nodes          : number of nodes, default: 20
     radio_range:       : max range of radio, default: 250 m
     wireless link alpha: (0,1), default: 0.5
     grid_length        : length of square grid, default: 500 m
     velocity           : speed of movement, default: 10 m/s
     pause_time         : duration of pause after movement, default: 40 sec

SPFv3 Config:
     wireless_interface : type of interface ptmp=1, manet=2, default: 1
     wireless_flooding  : flooding broadast=0, mpr=1, mdr=2, default: 1

     HelloInterval      : ospf6 HelloInterval, default: 10 sec
     DeadInterval       : ospf6 DeadInterval, default: 40 sec
     RxmtInterval       : ospf6 RxmtInterval, default: 5 sec
     MinLSInterval      : ospf6 MinLSInterval, default: 5 sec
     MinLSArrival       : ospf6 MinLSArrival, default: 1 sec

MPR MANET Interface Config:
     PushbackInterval   : Overlapping Relays timer, default: 2000 msec
     AckInterval        : Overlapping Relays timer, default: 1800 msec
     diff_hellos        : enable Incremental Hellos, default: off

MDR MANET Interface Config:
     BackupWaitInterval : MDR BackupWaitInterval, default: 1000 msec
     AckInterval        : MDR AckInterval, default: 1800 msec
     TwoHopRefresh      : MDR TwoHopRefresh hellos, default: 3 Hellos
     RouterDeadCount    : MDR RouterDeadCount, default: 3
     AdjConnectivity    : 0=Fully, 1=Uniconnected, 2=Biconnected, default: 2
     LSAFullness        : 0=only adjacent neighbors, 1=min-hop routing,
                          2=full LSAs from MDR/MBDRs, 3=full LSAs, default: 0
     diff_hellos        : enable MDR Differential Hellos, default: off
     NonPersistentMDR   : disable MDR persistent mdr/mbdrs,  default: off
```

To run with MDRs, the following non-default options are required: `wireless_interface=2`, `wireless_flooding=2`. Other parameters may be changed from the default as desired.

D.3 Implementation Description

The current implementation is based on `draft-ogier-manet-ospf-extension-03.txt` but goes beyond it in two ways: i) the draft was not complete enough to implement (no packet formats), and ii) Richard Ogier has been moving forward on the design since that draft was published. Since the writing of this technical report, `draft-ogier-manet-ospf-extension-04.txt` was published. The implementation is consistent with this subsequent draft in most aspects, with the draft discussing a few variants that are not implemented in the code.

Richard Ogier wrote a substantial portion of this implementation, and Phil Spagnolo has assisted by extending and debugging it.

D.4 Validation

This section shows some validation of the operation of the OSPF-MDR algorithm. The detailed trace and stat files can be found in the `validation/sicds/` under the specified test directory. Default command line parameters are used unless otherwise specificed under the scenario description. All figure symbols in this section can be identified by looking at the key in Figure C.1. See A.1.5 for more about the validation test suite.

D.4.1 MDR Flooding

This scenario (`validation/sicds/basic/`) is executed for 2000 seconds of simulation time. The first 1800 seconds are used for initialization, to get the network into a relatively quiescent state. The topology of the scenario is seen in Figure D.1. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, wireless_flooding=2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600.

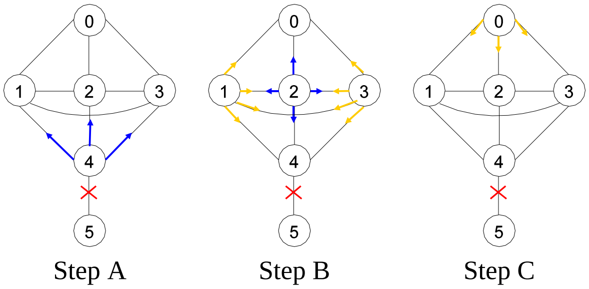


Figure D.1: MDR flooding validation

- **Step A:** At time 1900, the link is broken between nodes 4 and 5. After the neighbor relationship between nodes 4 and 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.
- **Step B:** With a point-to-multipoint interface, nodes 1, 2, and 3 would have forwarded the packet, but this scenario demonstrates that MDR flooding only requrires node 2 to forward the LSU. Because node 2 is the only node to forward the packet, nodes 1 and 3 acknowledge the LSA with an LSAck.
- **Step C:** Node 0 acks the LSA from node 2 to finish the exchange.

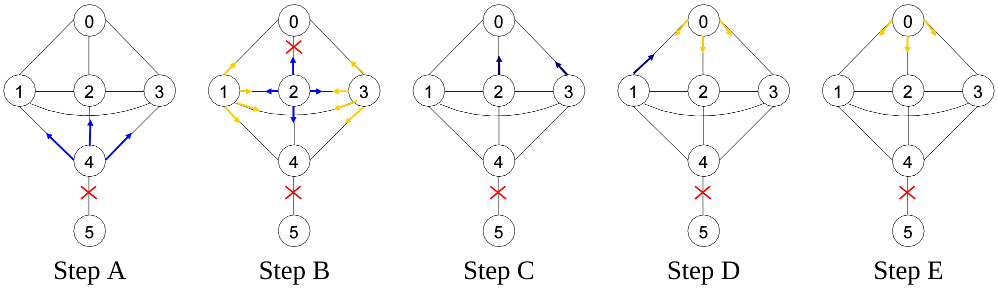


Figure D.2: Single LSU loss

D.4.2 BackupWait Operation

This sections explains the validation of the BackupWait technique that allows Backup MDRs to reflood or nodes to retransmit.

Single LSU Loss

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure D.2. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, wireless_flooding = 2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600.

- **Step A:** At time 1900, the link is broken between nodes 4 and 5. After the neighbor relationship between 4 & 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.
- **Step B:** Nodes 1 and 3 ack the LSA from node 4. The MDR that floods the LSU from node 4 is node 2. The LSU does not reach node 0 when it is flooded from node 2.
- **Step C:** The LSA is retransmited by nodes 2 and 3.
- **Step D:** When node 0 receives the LSA from 2 and 3 it then acknowledges it. The LSA is retransmited by node 1.
- **Step D:** When node 0 receives the LSA from node 1 it then acknowledges it.

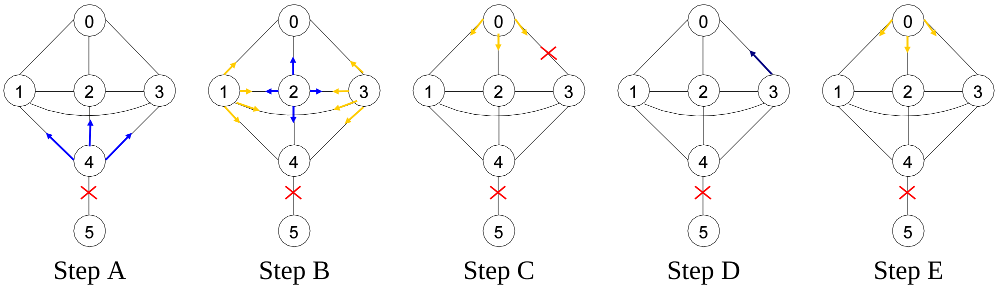


Figure D.3: Single Ack loss

Single Ack Loss

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure D.3. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, wireless_interface = 2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600.

- **Step A:** At time 1900, the link is broken between nodes 4 and 5. After the neighbor relationship between nodes 4 and 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.
- **Step B:** Nodes 1 and 3 ack the LSA from node 4. The MDR that floods the LSU from node 4 is node 2.
- **Step C:** After node 0 receives the LSA, it sends an LSAck to its neighbors. The LSAck is never received by node 3.
- **Step D:** Node 3 retransmists node 4's router LSA because it never received an ack from node 0.
- **Step E:** Node 0 then responds with an ack.

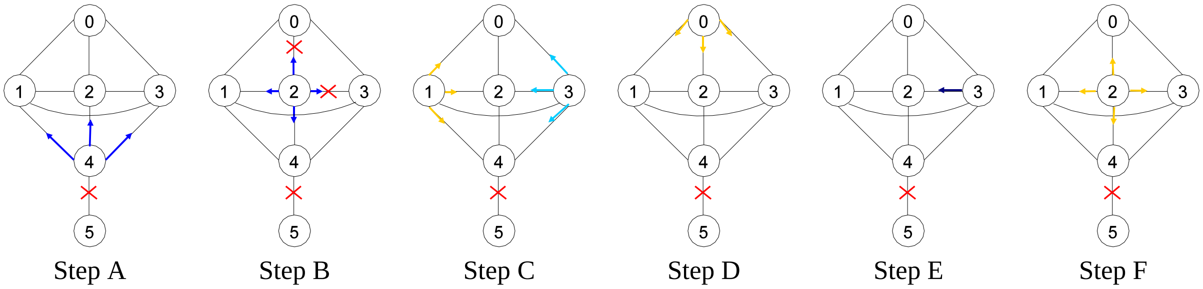


Figure D.4: Duel LSU loss

Duel LSU Loss

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure D.4. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, wireless_flooding = 2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600. used for initialization.

- **Step A:** At time 1900, the link is broken between nodes 4 and 5. After the neighbor relationship between 4 and 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.
- **Step B:** The MDR node that floods the LSU from node 4 is node 2. The LSU does not reach node 0 and 3 when it is flooded from node 2.
- **Step C:** The LSA is sent out by nodes 3 when its BackupWait timer expires. Nodes 1 acks the LSA from node 4.
- **Step D:** Node 0 acks the LSA from node 3.
- **Step E:** Node 3's retransmit timers expires because it did not hear node 2's flood.
- **Step F:** Node 2 acks the retransmit from node 3.

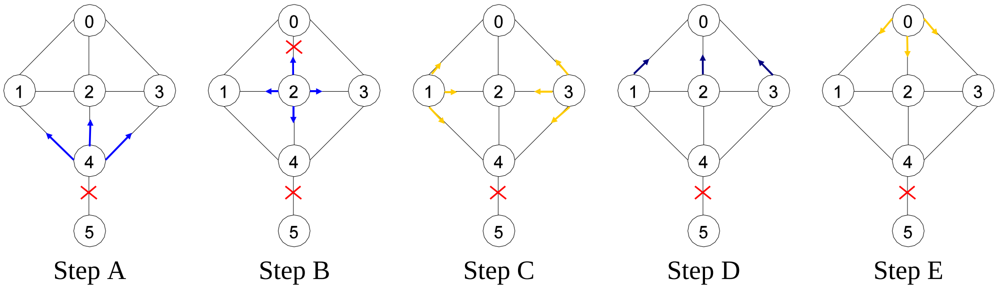


Figure D.5: Single LSU loss revisited

Single LSU Loss Revisted

This scenario is run for 2000 simulation seconds. The first 1800 seconds are used for initialization. The topology of the scenario is seen in Figure D.5. The following command line parameters were changed from the default values: nNodes = 6, wireless_interface = 2, wireless_flooding = 2, RxmtInterval = 10, PushbackInterval = 4000, and AckInterval = 3600.

- **Step A:** At time 1900, the link is broken between nodes 4 and 5. After the neighbor relationship between 4 & 5 expires, a new router LSA is originated by node 4. This LSA is flooded through out the network in an LSU.
- **Step B:** The MDR that floods the LSU from node 4 is node 2. The LSU does not reach node 0 when it is flooded from node 2.
- **Step C:** Nodes 1 and 3 ack the LSA from node 4.
- **Step D:** The LSA is retransmited by nodes 1, 2, and 3.
- **Step E:** When node 0 receives the LSA from 1, 2. and 3 it then acknowledges it.

D.4.3 Additional Validation Scenarios

Additional validation scenarios can be found in `validation/sicds/` that are not include here. The scenarios validate differential hellos and flooding with multiple LSAs.

Appendix E

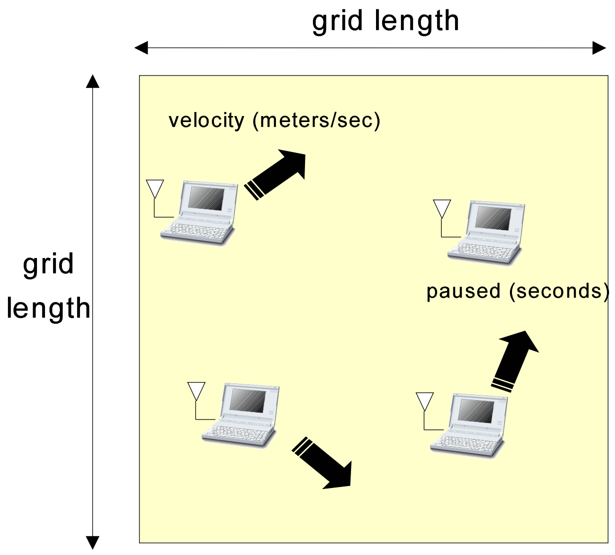
Sensitivity Analysis

This chapter describes results based on sensitivity analysis of the following:

- multicast-capable Point-to-Multipoint OSPFv3 interface (using standard flooding), and
- Cisco's proposed Overlapping Relays flooding

The code, scripts, and data from this chapter can be found in the `ietf/random_waypoint_manet/` directory.

The results in this section are based on a simple MANET scenario using an 802.11 or TDMA MAC. Nodes moved around randomly on a square grid, as shown in Figure E.1. Table E.1 lists some key configuration parameters in use by our simulation.



- Nodes move randomly in grid at fixed (configurable) velocity to a randomly selected location, then pause for fixed (configurable) time.
- Simulation runs execute for 30 minutes after a 30 minute startup period.
- Simulation runs are repeated with different random seeds, and the results averaged.
- A light amount of user data traffic is sent between nodes, to estimate the performance of the routing algorithm.
- Different radio models (802.11 and TDMA) are considered, without cross-layer optimizations between radio and router.

Figure E.1: Simulation scenario.

For this analysis, we experimented with two different values for the neighbor density and neighbor change rate which, when combined, yielded four scenarios, as shown in Figure E.2. As a reference point, the center of the diagram (0.1, 0.5) corresponds to a neighbor change rate of 1 change every 10 seconds per node (which means, for 20 nodes, that 2 nodes experienced a topology change every second, on average), and a neighbor density of 0.5, which for 20 nodes means that each node had about 10 neighbors. For shorthand, we represent the two regions of change rate as "(H)igh" and "(L)ow", and the two regions of density as "(D)ense" and "(S)parse". Considering these four scenarios as representing four "quadrants" of this parameter space, we note that the (L,S) (low change rate, sparse topology) quadrant produces the least overhead (because mobility is low and each node has fewer neighbors) while the (H,S)

Parameter	Definition	Value or range of values
Radio Range	Distance radio signals travel	250 m
Radio model	How nodes access the radio channel	802.11, TDMA
Grid size	Length and width of grid	Variable (500 m default)
Node velocity	velocity of node movement	Variable (10 m/s default)
Node pause time	Time that node pauses before moving again	40 sec
Mobility	How nodes move with the grid	Random waypoint
OSPF interface jitter	Amount of packet jitter	Uniform up to 100 msec
OSPF LSA coalesce time	Holds LSAs to allow to coalesce into bigger packets	100 msec from first scheduled LSA
OSPF interface cost	Cost used in SPF algorithm	1 (not dynamic)
OSPF HelloInterval	Standard OSPF timer	Variable (2 sec default)
OSPF RxmtInterval	Standard OSPF timer	Variable (5 sec default)
OSPF MinLSInterval	Limit on frequency of LSA generation	Variable (5 sec default)
OSPF MinLSArival	Limit on frequency of LSA reception	Variable (1 sec default)
OSPF PushBackInterval	Timer used on MANET interface	Variable (40% of RxmtInterfval default)
OSPF AckInterval	Timer used on MANET interface	Variable (90% of RxmtInterfval default)
OSPF Differential Hellos	Whether a MANET interface uses differential hellos	Variable (default is off)
OSPF LSA Caching	Whether a MANET interface uses opportunistic LSA caching	Variable (default is off)
User data packet size	Size of user data packets	40 bytes default
User data packet rate	Rate at which user data is sent into the network	10 packets/sec default

Table E.1: Main configurable simulation parameters.

produces the worst user data delivery ratio (because routes are sparse and they frequently change). Quadrant (H,D) has the highest overhead due to a large number of neighbors and link changes.

Although the above table identifies an OSPF RxmtInterval of 5 seconds, we used 10 seconds in the below tests since it resulted in better performance.

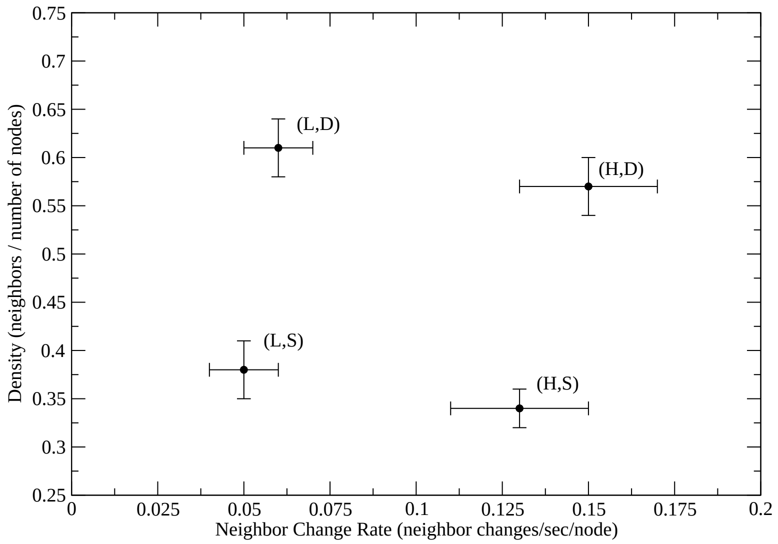


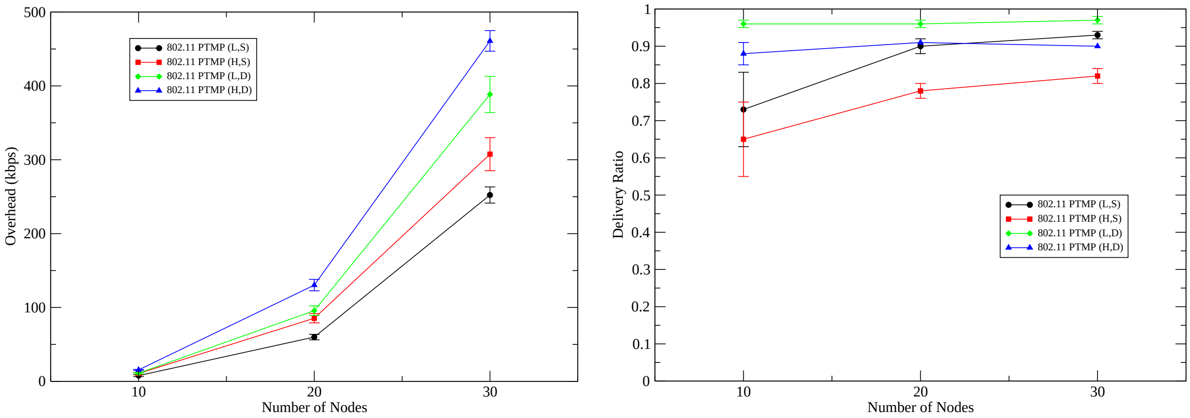
Figure E.2: Quadrants of neighbor change rate and density

E.1 Baseline Point-to-Multipoint

We begin by analyzing the OSPFv3 Point-to-MultiPoint interface (herein abbreviated as .PTMP.), modified for support of broadcast operation, in a MANET.

E.1.1 *GTNets* Point-to-Multipoint Results Using an 802.11 MAC

The results shown in Figure E.3 are for an 802.11-based radio and channel model with a Point-to-MultiPoint interface. The minimum network overhead is generated in the (L,S) quadrant, and the highest overhead in the (H,D) quadrant. This is intuitive because low density and low change require the fewest routing changes and the least amount of flooding.



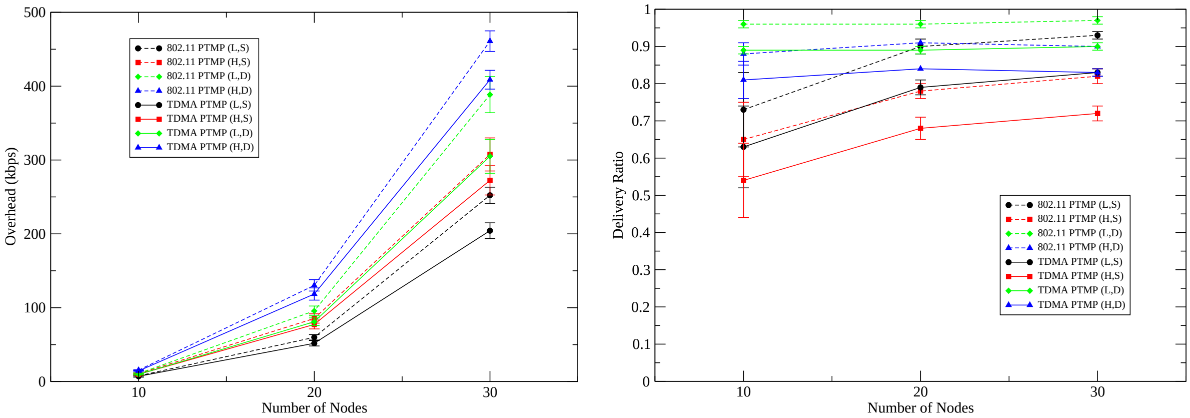
(a) Overhead

(b) Delivery Ratio

Figure E.3: MANET using an 802.11 MAC with OSPFv3 PTMP.

E.1.2 *GTNets* Point-to-Multipoint Results Using a TDMA MAC

We next illustrate the sensitivity of the results to the radio model. Figure E.4 illustrates TDMA results overlaid on the 802.11 results. Figure E.4 shows that TDMA generates slightly less overhead. This is because TDMA does not have a contention based channel; thus, there are no collisions resulting in retransmissions. However, the difference is not particularly significant. Other radio channel and interference models may yield a greater difference between the performance of TDMA and 802.11.



(a) Overhead

(b) Delivery Ratio

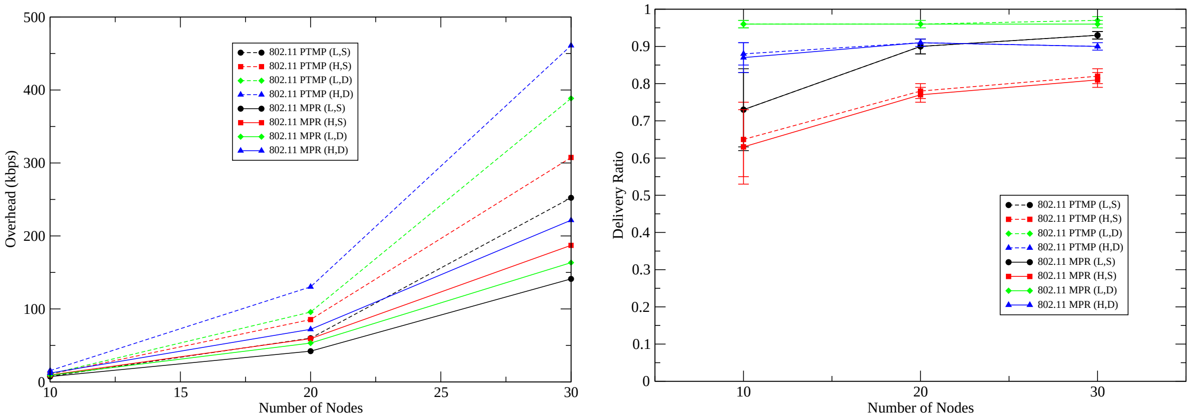
Figure E.4: Comparison of a MANET using a TDMA or 802.11 MAC with OSPFv3 PTMP.

E.2 MPRs with Overlapping Relays

E.2.1 *GTNets* MPR Flooding Results Using an 802.11 MAC

Figure E.5 compares the overhead mobility results for the MPR flooding-based interface with the baseline results shown in Figure E.3 above. The MPR flooding reduces the amount of overhead by up to 52% (the high change rate, high density case with 30 nodes). The percent reduction is reduced as the number of nodes is decreased.

Figure E.5, also compares a plot of the probability of success for user data delivery (the "delivery ratio"), shows that the choice of interface type does not worsen the delivery ratio. In other words, the overhead savings due to the MPR-based flooding does not compromise routing performance. The figure also shows that the delivery ratio is most highly dependent on the network's change rate and density.



(a) Overhead

(b) Delivery Ratio

Figure E.5: Comparison of MANET using an 802.11 MAC with OSPFv3 PTMP or MPR Flooding.

A closer look at the statistics for a pair of data points in Figure E.5 is informative. Table E.2 compares the statistics from two identical simulation scenarios (average of ten independent runs each), with the only difference being the OSPFv3 interface type. This corresponds to 30 nodes, 802.11 radio, with (H)igh mobility and (D)ense network. It is clear that, with respect to the OSPFv3 statistics, the MPR Flooding interface reduces both the initially flooded LSAs (LSU-Flooding case in the figure) and retransmitted LSAs (LSU-Unicast). Notice also that the amount of overhead from LSAcks increases, due to the need to send more acknowledgments if implicit acknowledgments (due to reflooding) are suppressed by the choice of MPRs. At the bottom of the table, we provide a breakdown of the number of LSAs that were suppressed, reflooded, or retransmitted by a non-active relay. It is clear that MPR flooding initially suppresses the transmission of a large number of LSAs. It can also be seen that the MPR selection algorithm is efficient in this fairly dense network, with each node only selected for reflooding by 2.6 of its 19 neighbors. However, the suppression of large numbers of LSAs is somewhat lessened by the need to resend some of those LSAs by non-active relays. This points out a tradeoff of efficient flooding in a wireless environment; if the initial flood is too efficient, it may have to be covered by a large number of retransmissions. The last row indicates that the delivery ratio for user data is identical for the two cases, illustrating that the routing protocol performance is not hindered by the more efficient LSA dissemination. Finally, note that the overhead consumed by Hello messages is less than 10% of the total overhead, making it an unlikely candidate for drastic improvements from differential or incremental Hellos.

E.2.2 *GTNets* MPR Flooding Results Using a TDMA MAC

Figure E.6 presents additional data that suggests that the performance of these two routing algorithms is relatively insensitive to the selection of underlying MAC protocol. TDMA and 802.11 follow the same trend in the amount of overhead that is generated at all data points.

	Point-to-MultiPoint	**MPR Flooding**
Scenario statistics		
Number of nodes	30	30
Average density (nbrs/#nodes)	0.56	0.56
Average changes/second/node	0.24	0.24
OSPF statistics		
Hello	14.9	17.3
LSAck	3.7	20.4
LSR	0.1	1.2
Database Desc.	71.0	70.7
LSU		
Flooding	349.2	86.9
Unicast	22.0	25.1
Total Overhead	461.0	221.5
Flooding statistics		
Average OSPF neighors/node	17.0	17.0
MPR selectors/node	N/A	2.6
Suppressed LSAs	N/A	208,000
Re-flooded LSAs	250,000	35,000
Non-active LSA floods	N/A	27,000
User data delivery ratio	0.90	0.90

Table E.2: Main configurable simulation parameters.

E.2.3 Parameter Sensitivity

This section presents sensitivities to various timer values in the MANET-extended OSPFv3 protocol. All simulations were run over an 802.11 MAC using overlapping relays. Table E.3 gives the default values for each of the timers that were explored. The last two entries in the table are multipliers that yield the PushBackInterval and AckInterval via the following equations:

$$PushBackInterval = (RxmtInterval)(PushBackIntervalMultiplier) \tag{E.1}$$

$$AckInterval = (PushBackInterval)(AckIntervalMultiplier) \tag{E.2}$$

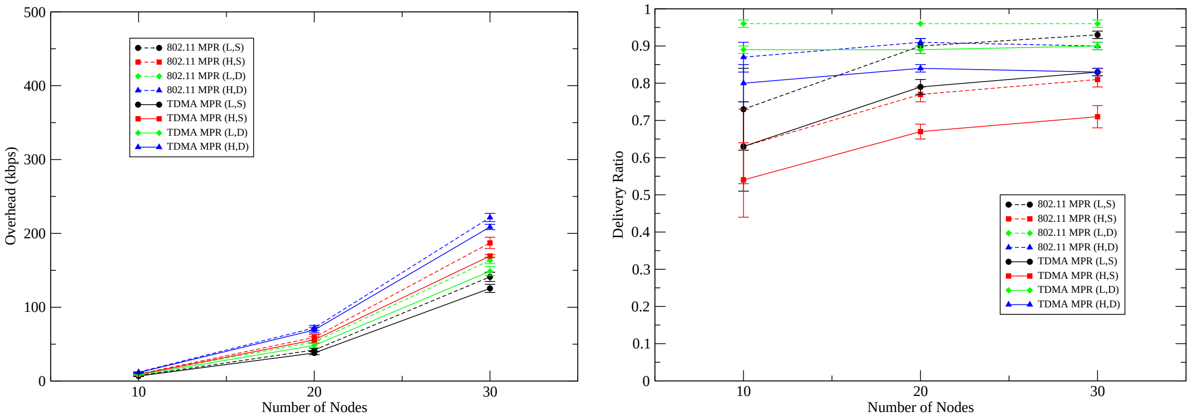
HelloInterval	2
RxmtInterval	10
MinLSInterval	5
MinLSArrival	1
PushBackInterval Multiplier	0.45
AckInterval Multiplier	0.9

Table E.3: Default OSPFv3 sensitivity values.

In each of the following subsections, the timer of interest was varied while all the other timers were held at their default values (although the router dead interval was maintained at four times the value of the Hello interval for each distinct Hello interval tried). In addition, we studied the performance under the four different mobility/density scenarios introduced in Figure E.2.

HelloInterval

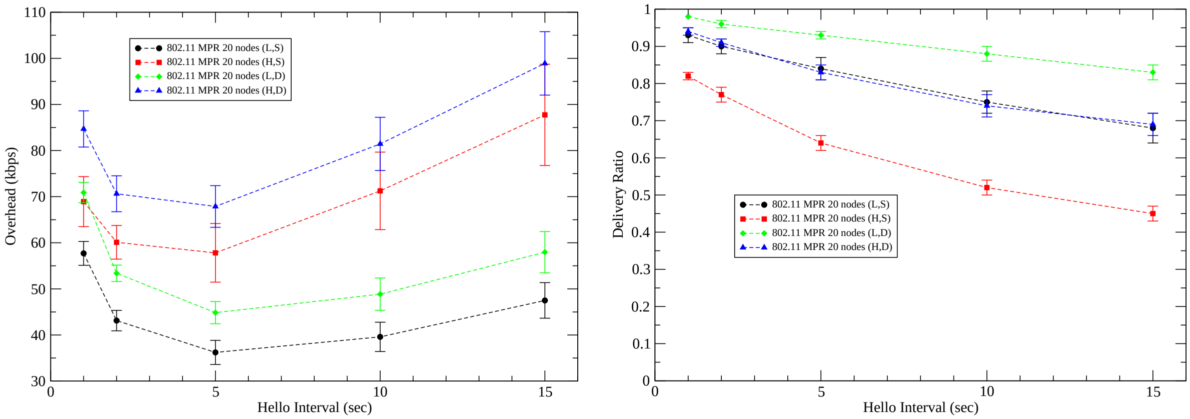
In this section, we illustrate results obtained by varying the Hello interval. The Hello interval is the periodicity at which Hellos are sent on each OSPFv3 interface. The Hello interval was changed between the values of 1, 2, 5, 10, and 15 seconds. The data (Figure E.7) indicates that it is advantageous to use a low Hello interval. A low Hello



(a) Overhead (b) Delivery Ratio

Figure E.6: Comparison of a MANET using a TDMA or 802.11 MAC with OSPFv3 MPR Flooding.

interval means each router is more aware of its current neighborhood. This enables the MPR flooding algorithm to choose MPRs more accurately for the current topology, and appears to improve the data delivery ratio. By choosing more accurate MPRs, there is less need for retransmission, and flooding can be performed in fewer hops. Figure E.7 also shows that when the Hello interval becomes too low, the savings due to knowing the neighborhood more accurately starts to be overridden by larger Hello overhead. This apparent benefit of using a low Hello interval implies that periodic differential Hellos may be more advantageous, since the overall quantity of Hello messages increases when the interval is low. Differential Hellos may reduce the size of the Hellos because the neighborhood should not greatly vary when the period is small.

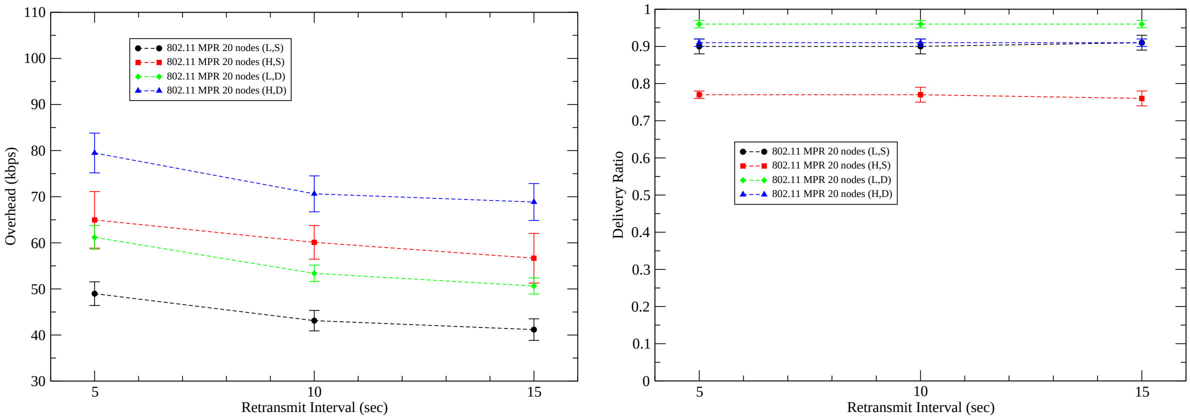


(a) Overhead (b) Delivery Ratio

Figure E.7: Sensitivity to the HelloInterval.

RxmtInterval

In this section, we explore the effect of varying the retransmit interval (RxmtInterval) parameter of OSPFv3. The retransmit interval is the duration that a router will wait for a neighbor to acknowledge an LSA. The retransmit interval was varied between 5, 10, and 15 seconds. The results in Figure E.8 suggests that the overhead is sensitive to the retransmit interval, while the data delivery ratio is insensitive. This suggests that most retransmissions are due to out of range transmissions. Any number of retransmissions will not help nodes out of range. Also, note that we did not correspondingly reduce the router dead interval, which is tied to the Hello interval, when we scaled back the retransmit interval.



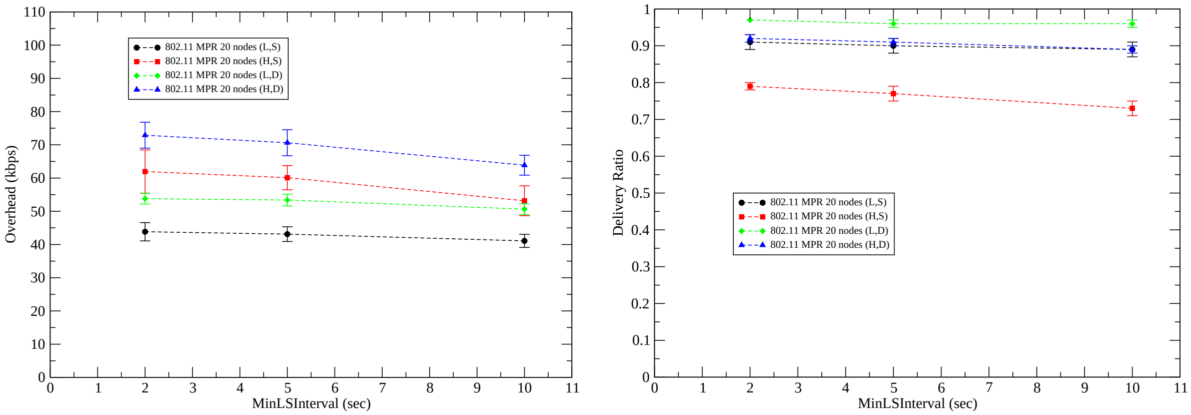
(a) Overhead

(b) Delivery Ratio

Figure E.8: Sensitivity to the RxmtInterval.

MinLSInterval

The MinLSInterval is an architectural constant in OSPFv3 that enforces a minimum time between distinct originations of any particular LSA. By default in OSPFv3, it is set to 5 seconds. MinLSInterval can be important in a mobile network because it bounds the frequency with which new LSAs can be originated (helping to reduce the overhead on the one hand, but slowing the dissemination of topology updates on the other). In this section, the MinLSInterval was varied between 2, 5, and 10 seconds. Figure E.9 indicates that the performance is not highly insensitive to this parameter. But, we do see that overhead and delivery ratio increase when MinLSInterval is smaller (better view of the current network at the expense of more updates), and overhead and delivery ratio decrease when MinLSInterval is larger (worse view of the network with the gain of fewer updates).



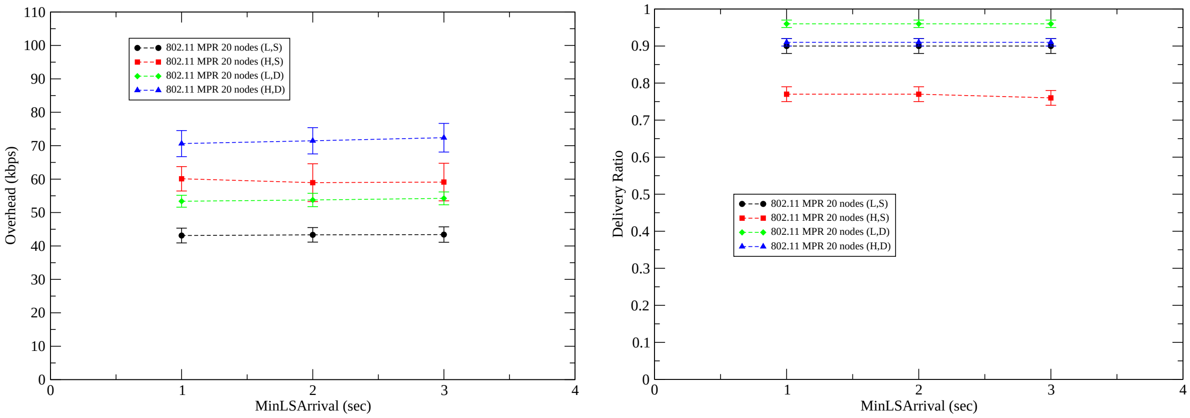
(a) Overhead

(b) Delivery Ratio

Figure E.9: Sensitivity to the MinLSInterval.

MinLSArrival

The MinLSArrival parameter is an architectural constant in OSPFv3 that enforces, on the receive side, a minimum time between processing of distinct LSA instances. By default in OSPFv3, it is set to 1 second. LSA instances received at a higher frequency are discarded. This acts as a safety valve in refusing to propagate large numbers of LSAs originating from a single router. Figure E.10 shows that our results are not sensitive to this parameter.

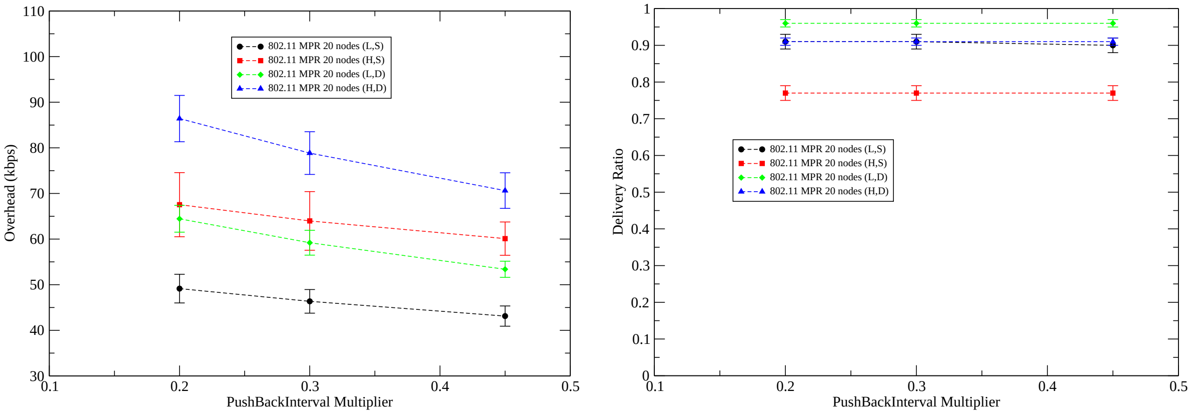


(a) Overhead (b) Delivery Ratio

Figure E.10: Sensitivity to the MinLSArrival.

PushBackInterval

In this section, we study the sensitivity to the PushBackInterval. The PushBackInterval is the time that a nonoverlapping relay node waits before flooding an LSA for which it originally suppressed the flooding. If the node does not (passively) receive some indication (acknowledgement or reflooding of the LSA from the downstream node) that the LSA was successfully received, the node will become an active relay for that LSA and the LSA will be flooded between PushBackInterval and 2*PushBackInterval seconds. This range was a design choice by Boeing. The PushBackInterval multiplier was set at 0.2, 0.3, and 0.45. These values correspond to ranges of 2 to 4, 3 to 6, and 4.5 to 9 seconds respectively, based on our default retransmit interval of 10 seconds. Figure E.11, shows that a larger PushBackInterval yields lower overhead. The increase in the PushBackInterval provides more time to receive acknowledgements, and it spreads the time over which nonoverlapping relays will send a pushed back LSA (thereby reducing the probability that two nodes will attempt to retransmit at approximately the same time). These two properties lead to lower overhead, at no expense to the delivery ratio.

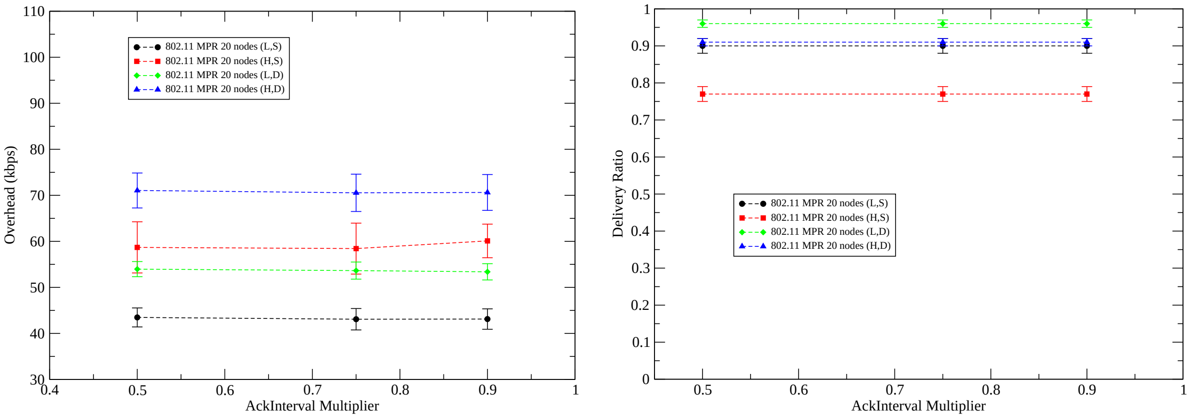


(a) Overhead (b) Delivery Ratio

Figure E.11: Sensitivity to the PushBackInterval.

AckInterval

In this section, the AckInterval parameter was varied. The AckInterval is the time for which a router waits to coalesce acknowledgements, so as to send fewer (but larger) acknowledgment packets. The AckInterval multiplier was set at 0.5, 0.75, and 0.9. These values correspond to ranges of 0 to 1.8, 0 to 2.7, and 0 to 4.05 seconds respectively. The results shown in Figure E.12 indicates that the AckInterval has little effect on the overhead or the delivery ratio in these scenarios.



(a) Overhead

(b) Delivery Ratio

Figure E.12: Sensitivity to the AckInterval.