# RapidMesh: Declarative Toolkit for Rapid Experimentation of Wireless Mesh Networks

Shivkumar C. Muthukumar
University of Pennsylvania
mshivk@seas.upenn.edu

Xiaozhou Li
University of Pennsylvania
xiaozhou@seas.upenn.edu

Changbin Liu
University of Pennsylvania
changbl@seas.upenn.edu

Joseph B. Kopena
Drexel University
tjkopena@cs.drexel.edu

Mihai Oprea
University of Pennsylvania
mihaio@seas.upenn.edu

Ricardo Correa
University of Pennsylvania
ricm@seas.upenn.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Prithwish Basu
BBN Technologies
pbasu@bbn.com

## ABSTRACT

We present the *RapidMesh* toolkit for rapid protocol simulation, implementation and experimentation of wireless mesh networks. *RapidMesh* utilizes *declarative networking*, a declarative, database-inspired extensible infrastructure that uses query languages to specify behavior. *RapidMesh* integrates a declarative networking engine with the emerging ns-3 network simulator. The same declarative specifications can also be used as actual implementations using the ns-3 network emulator, hence providing a bridge between simulation and testbed-based experimentation. We demonstrate that *RapidMesh* enables a variety of wireless routing protocols and neighbor discovery protocols can be synthesized via compact declarative specifications. We experimentally validate declarative MANET routing protocols in dynamic settings within *RapidMesh* operating in ns-3 simulation environment and on the ORBIT wireless testbed.

## Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

## General Terms

Design, Languages, Experimentation

## Keywords

Declarative networking, Simulation, Experimentation


*WiNTECH'09,* September 21, 2009, Beijing, China.


## 1. INTRODUCTION

In the past decade, there has been intense activity on the development of routing protocols for mobile ad hoc networks (MANETs). A wide variety of routing protocols have been proposed, all with their own strengths and weaknesses, and all with varying degrees of success. For example, reactive routing protocols such as DSR [7] and AODV [14] set up routing state on demand and hence are preferred for low traffic environments; proactive routing protocols such as OLSR [4] on the other hand expend network bandwidth to gather routing state with a purpose of amortizing this extra cost over multiple traffic flows – hence these are better for high traffic load environments, in general. Recently researchers have focused on the disruption tolerance aspects of MANETs that are at best intermittently connected, e.g., epidemic routing protocols [20].

Despite the proliferation of wireless routing protocols, there have been a lack of systematic tools that can enable one to carefully study the performance characteristics of these protocols under a variety of mobility settings. While extensive simulation studies (e.g. [2]) have been carried out in the past and provided useful insights into different routing protocols, they may not completely reflect real-world effects which manifest themselves in actual deployments. The recent advent of open testbeds such as Orbit [19] presents an avenue for evaluating these MANET protocols under realistic settings. However, implementing, deploying and evaluating MANET protocols on these testbeds (particularly with mobility induced in the experiment setup) remain arguably a time-consuming and tedious process.

In reality, network designers require a combination of simulation and evaluation on realistic testbeds. Simulation enables one to study routing protocols in a controlled environment where one can scale to large number of nodes under complex mobility patterns. After validating the design in simulation, an actual wireless testbed is useful for studying the protocol in a realistic setting.

In this paper, we present *RapidMesh*, a development toolkit that enables one to rapid prototyping and analyze wireless mesh protocols in simulation and on actual testbeds. *RapidMesh* utilizes *declarative networking* [11, 10], a declar-

ative, database-inspired extensible infrastructure that uses query languages to specify network behavior. A declarative approach enables modular reuse of resources and functions by allowing network programmers to say "what" they want, without worrying about the details of "how" to achieve it. This makes it easy to specify protocols, since specification is largely confined to the "what", while implementation of the "how" is automated. In addition, there are other benefits such as compactness of specifications, safety, and potential for correctness checks.

*RapidMesh* integrates a declarative networking engine with the emerging ns-3 [13, 5] network simulator, intended as an eventual replacement for the ns-2 simulator. MANET routing and neighbor discovery protocols are specified using declarative specifications, which are then compiled into ns-3 code for simulation and analysis. The same declarative specifications can also be used as actual implementations using the ns-3 network emulator, hence providing a bridge between simulation and testbed-based experimentation.

Specifically, the paper makes the following contributions:

**Declarative protocols for wireless mesh networks:** We demonstrate that *RapidMesh* enables routing protocols such as various variants of link state routing, dynamic source routing and epidemic routing to be specified compactly as declarative networks in a compact and clean fashion, typically in a handful of lines of program code. Moreover, neighborhood discovery protocols and distributed queries that monitor properties of the network, and testbed configurations (e.g. determining the current set of neighbors) can be specified using declarative networking language.

**Experimental validation:** We experimentally validate declarative MANET routing protocols in dynamic settings within *RapidMesh* operating in ns-3 simulation environment and on the ORBIT wireless testbed. Our results demonstrate that *RapidMesh* can enable a network developer to generate compact routing specifications that can be simultaneously evaluated in simulation and on an actual testbed to gain insights into protocol behavior under mobility.

The long term goal of *RapidMesh* is to provide a platform for rapid prototyping, synthesis, and deployment of new network protocols. In addition to being a valuable tool for rapid network prototyping and analysis, *RapidMesh* can potentially be used as a basis of an educational software package that packages the declarative platform with ns-3 and the ORBIT testbed, enabling students to learn and experiment with network protocols via higher level declarative abstractions.

# 2. BACKGROUND

The high level goal of *declarative networks* is to build extensible architectures that achieve a good balance of flexibility, performance and safety. Declarative networks are specified using *Network Datalog* (*NDlog*), which is a distributed recursive query language for querying networks.

Declarative queries such as *NDlog* are a natural and compact way to implement a variety of routing protocols and overlay networks. For example, traditional routing protocols such as the path vector and distance-vector protocols can be expressed in a few lines of code [11], and the Chord distributed hash table in 47 lines of code [10]. When compiled and executed, these declarative networks perform efficiently relative to imperative implementations. Recent work [21] has also shown that declarative specifications can be verified using a mechanized prover, hence enabling the bridging of specification, verification, and implementation.

*NDlog* is based on Datalog [15]: a Datalog program consists of a set of declarative *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and `...` and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* with *attributes* (which are bound to variables or constants by the query), or boolean expressions that involve function symbols (including arithmetic) applied to attributes.

Datalog rules can refer to one another in a mutually recursive fashion. The order in which the rules are presented in a program is semantically immaterial; likewise, the order predicates appear in a rule is not semantically meaningful. Commas are interpreted as logical conjunctions (*AND*). The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter. Function calls are additionally prepended by `f_`. Aggregate constructs are represented as functions with attribute variables within angle brackets (`<>`). We illustrate *NDlog* using a simple example of two rules that computes all pairs of reachable nodes:

```
r1 reachable(@S,N) :- link(@S,N).
r2 reachable(@S,D) :- link(@S,N),
           reachable(@N,D).
```

The rules `r1` and `r2` specify a distributed transitive closure computation, where rule `r1` computes all pairs of nodes reachable within a single hop from all input links (denoted by the `link`, and rule `r2` expresses that "if there is a link from `S` to `N`, and `N` can reach `D`, then `S` can reach `D`." The output of interest is the set of all `reachable(@S,D)` tuples, representing reachable pairs of nodes from `S` to `D`. By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols.

*NDlog* supports a *location specifier* in each predicate, expressed with the `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, all `reachable` and `link` tuples are stored based on the `@S` address field. To support wireless broadcast, we have introduced a *broadcast location specifier* denoted by `@*` which will broadcast a tuple to all nodes within wireless range of the node where the rule is executed.

*NDlog* queries are compiled and executed as *distributed dataflows* by the query processor to implement various network protocols. These dataflows share a similar execution model with the Click modular router [8].

## 2.1 Soft-state Storage Model

Declarative networking incorporates support *soft-state* [16] derivations commonly used in networks. In the soft state storage model, all data (input and derivations) has an explicit "time to live" (TTL) or lifetime, and all tuples must be explicitly reinserted with their latest values and a new TTL, or they are deleted.

The soft-state storage semantics are as follows. When a tuple is derived, if there exists another tuple with the same primary key but differs on other attributes, an *update* occurs, in which the new tuple replaces the previous one. On the other hand, if the two tuples are identical, a *refresh* occurs, in which the existing tuple is extended by its TTL.

For a given predicate, in the absence of any `materialize` declaration, it is treated as an *event* predicate with zero lifetime. Since events are not stored, they are primarily used to trigger rules periodically or in response to network events. Event predicates (whose names start with an additional "e") are used to denote transient tables which are used as input to rules but not stored.

For example, utilizing the built-in `periodic` keyword , node `X` periodically generates a `ePing` event every 10 seconds to its neighbor `Y` denoted in the `link(@X,Y)` predicate:

```
ePing(@Y,X) :- periodic(@X,10), link(@X,Y).
```

# 3. OVERVIEW

Figure 1 provides an overview of *RapidMesh*'s basic approach towards unifying specifications, simulation, and implementation within a common declarative framework. In the initial design phase of *RapidMesh*, a network design is used as the basis for specifying the network protocol using the *NDlog* declarative networking language. High-level invariant properties of the protocol can also be expressed in *NDlog* as *distributed queries* which raise event alarms when invariants are violated.

In the *simulation mode*, the *RapidMesh* compilation process generates ns-3 code from the *NDlog* protocol specifications and invariants. The generated code either runs as an ns-3 application, or replaces routing protocol implementations at the network layer. The generated code implements dataflows (execution plans) with a similar execution model with the Click modular router [8], which consists of elements that are connected together to implement a variety of network and flow control components. In addition, those elements include database operators (such as joins, aggregation, selections, and projects) that are directly generated from the declarative networking rules. Messages flow among dataflows executed at different nodes, resulting in updates to local tables. The local tables store the state of intermediate and computed query results which represent the network state of various network protocols.

In the *implementation mode*, declarative networking specifications are directly executed and deployed either using the P2 declarative networking system [1] or the ns-3 network emulator. *RapidMesh* currently utilize ns-3 network emulator for actual implementation. In the emulation mode, each ns-3 simulation node connects to the real physical network underneath using a raw socket. While we have successfully implemented several declarative networks using the P2 system in the past, we have chosen to adopted the ns-3 emulation environment to implement the protocols themselves. The main advantage is in ensuring that all declarative networking protocols are evaluated in simulation and emulation within a common ns-3 code base.

Since declarative networks share common functionalities such as the network stack, multiplexing tuple messages entering and leaving the dataflow, and database functionalities, all these utilities are defined in a shared *RapidMesh* library. This enables one to simplify the compilation process to only the relevant database operations that implement the distributed dataflows for the corresponding declarative network specification. This also enables one to easily incorporate *multi-query optimizations* to share computations across declarative networks in future.

# 4. DECLARATIVE SPECIFICATIONS

We demonstrate a variety of MANET routing protocols expressed using *RapidMesh*. Table 1 summarizes the declarative protocols that we have successfully implemented, and the corresponding number of rules required. In this section, we primarily focus our discussions on proactive protocols , and a brief description of neighbor discovery rules and monitoring queries. Appendix A present additional examples on DSR and epidemic routing.

| Category | Protocol | Rules |
|---|---|---|
| Reactive | Dynamic source routing | 11 |
| Proactive | Traditional link state | 15 |
| | Optimized link state routing | 34 |
| | Hazy sighted link state | 18 |
| Epidemic | Summary-vector based epidemic | 17 |

**Table 1: Declarative MANET Protocols and number of rules.**

## 4.1 Proactive Protocols

A well studied proactive protocol is the link-state protocol, in which the entire topology is disseminated to all nodes in the network. We show first an example for network-wide flooding of link-state (LS) updates in traditional link-state, followed by two variants of link-state commonly used in MANET settings. Traditional dissemination of link state information is expressed by the following *NDlog* rules:

```
ls1 lsu(@S,S,N,C,S) :- link(@S,N,C).
ls2 lsu(@M,S,N,C,Z) :- link(@Z,M,C1),
          lsu(@Z,S,N,C,W), M!=W.
```

`lsu(@M,S,N,C,Z)` is a link state update (LSU) corresponding to `link(S,N,C)`, which indicates a link between node `S` and `N` with a cost of `C`. This LSU tuple is flooded in the network starting from source node `S`. During the flooding process, node `M` is the current node it is flooded to, while node `Z` is the node that forwarded this tuple to node `M`.

Rule `ls1` generates an `lsu` tuple for every link at each node. Rule `ls2` states that each node `Z` that receives an `lsu` tuple recursively forwards the tuple to all neighbors `M` except the node `W` that it received the tuple from. Datalog tables are set-valued, meaning that duplicate tuples are not considered for computation twice. This ensures that no similar `lsu` tuple is forwarded twice.

The above LS rules perform *triggered updates* continuously: whenever a `link` is added or deleted, a corresponding `lsu` is inserted or deleted locally, and then flooded to the entire network. As an alternative, one may prefer to implement link-state via *periodic updates* by modifying rule `ls1` as follows:

```
ls1p lsu(@S,S,N,C,S) :- periodic(@S,10),
         link(@S,N,C).
ls2p lsu(@M,S,N,C,Z) :- link(@Z,M,C1),
         lsu(@Z,S,N,C,W), M!=W.
```

In rule `ls1p` we utilize the `periodic` keyword to flood once in every 10 seconds. In order to ensure freshness of `lsu` tuples, they are stored using soft-state (Section 2.1), where the lifetimes are set to be roughly the duration of periodic floods. In practice, a combination of triggered updates for

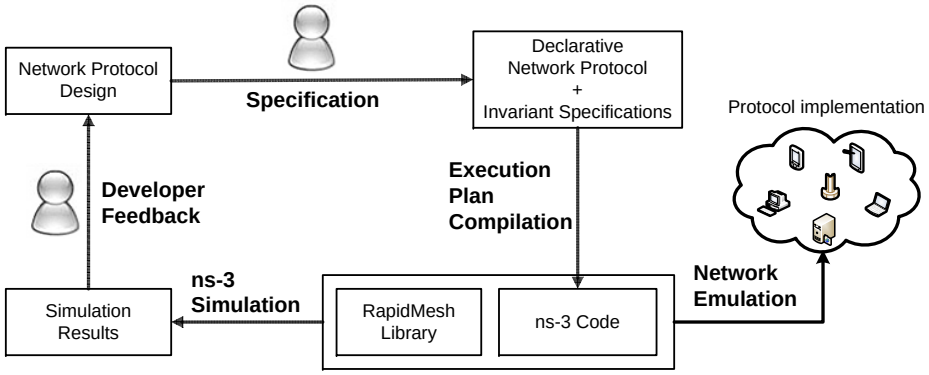


**Figure 1: Overview of *RapidMesh***

timeliness and periodic updates for robustness are used. The declarative framework enables both approaches naturally via modifications to a single rule, demonstrating the power of declarative programming. In addition, *batched triggered updates* in which updates are batched and propagated at fixed intervals can also be concisely expressed within this framework.

Our example above utilizes unicast communication, where each `link` tuple results in an `lsu` tuple being sent via unicast to each neighbor. Using the broadcast location specifier `@*` described in Section 2, the following rules broadcast link information to all neighbors within the wireless range of each node `S`:

```
ls1b lsu(@*,S,N,C,S) :- link(@S,N,C).
ls2b lsu(@*,S,N,C,Z) :- lsu(@Z,S,N,C,W).
```

Once the entire network topology, i.e., all the links, are available at each node, additional rules are required in order to compute the shortest paths with minimum cost `C` for each source `S` and destination `D`. These rules take as input the local `lsu` tuples generated, and essentially result in the execution of the Dijkstra's algorithm locally. They are shown as follows:

```
bp1 path(@M,S,N,P,C) :- lsu(@M,S,N,C,W),
          P=f_init(S,N).
bp2 path(@M,S,D,P,C) :- lsu(@M,S,N,C1,W),
          bestPath(@M,N,D,P2,C2),
          C=C1+C2, P=f_concatPath(S,P2).
bp3 bestPathCost(@M,S,D,min<C>) :-
          path(@M,S,D,P,C).
bp4 bestPath(@M,S,D,P,C) :-
          bestPathCost(@M,S,D,C),
          path(@M,S,D,P,C).
```

In rule `bp1`, 1-hop paths are built from every link, while in rule `bp2` paths are recursively constructed by concatenating shorter path with links. Rule `bp3` computes the minimum cost for paths with same sources and destinations, and rule `bp4` finally computes the best paths.

**Optimized Link-state Routing (OLSR):** A well-known proactive MANET protocol is OLSR (Optimized Link State Protocol) [4]. OLSR ensures efficient flooding by forwarding LSUs to a subset of neighbors known as multipoint relays (MPR). The union of the neighbor sets of MPRs of any node X is equal to the set of 2-hop neighbors of X. The following rules `olsr1-2` are modified from rule `ls1-2` to implement OLSR-style flooding of LSUs:

```
olsr1 lsu(@S,S,N,C,S) :- periodic(@S,10),
          link(@S,N,C).
olsr2 lsu(@M,S,N,C,Z) :- mpr(@Z,M,C1),
          lsu(@Z,S,N,C,W), M!=W.
```

The `mpr` predicate in rule `olsr2` denotes MPR tuples each storing multipoint relay node `M` for node `Z`. This predicate itself can be defined with additional rules to customize the definition of MPR.

**Hazy-sighted Link-state (HSLS):** Hazy Sighted Link State routing (HSLS) [18] is a scalable LS routing variant for handling moderate to high rate of change in network topology. This protocol attempts to control the scope and frequency of its LSU flooding scheme based on the topology of the network. The basic principle of HSLS is that route calculation of a node should not be affected significantly by link dynamics due to mobility or failure in a portion of network that is far away from this node. Hence unlike the pure LS protocol which performs a network wide flood of all LSUs, HSLS sends LSUs to the $2^k$ hop neighbors of a node with a period equal to $2^k T_e$, where $T_e$ is a nominal period. If link dynamics are high, pure LS starts thrashing because remote nodes could receive an LSU corresponding to a link that has long vanished.

Policy rules used in HSLS are expressed as follows:

```
hsls1 lsu(@S,S,N,C,S,TTL) :- periodic(@S,T),
          link(@S,N,C), T=f_pow(2,K)*Te,
          TTL=f_pow(2,K), K=range[1,10].
hsls2 lsu(@M,S,N,C,Z,K-1) :-
          lsu(@Z,S,N,C,W,K),
          link(@Z,M,C1), K>0, M!=W.
```

Rule `hsls1` is periodically fired, and the period of execution depends on $2^K T_e$. Note that here we add one more attribute for `lsu` tuple, which is `TTL` used for controlling flooding scope. In declarative networking, it is easy to modify tuples, such as adding and deleting their attributes due to the need of different protocols. Rule `hsls2` keeps forwarding LSUs if their `TTL` is larger than 0. Similar to LS, the HSLS rules can be modified to support triggered updates or batched triggered updates. In triggered updates, the flood of the corresponding LSU for each link insert/delete event is scoped using using a similar HSLS policy, where LSUs

flooded within a time interval has a fixed TTL. If triggered updates are used, in order for all LSUs to reach every node, a periodic network-wide LSU flooding needs to be carried out based on the nominal period $T_e$.

## 4.2 Neighbor Discovery and Monitoring

In all of the above examples, the `link` table is used as input to the routing protocol. The table represents the neighborhood information gathered at each node, and can itself be generated via a neighbor discovery process. For instance, an *NDlog* rule can be used to generate a periodic beacon message which is used to refresh the `link` tables at neighboring nodes. To illustrate, we consider the following two *NDlog* rules:

```
d1 beaconMsg(@*,S) :- periodic(@S,10).
d2 link(@N,S,1) :- beaconMsg(@N,S).
```

In rule `d1`, each node `S` generates a broadcast beacon message every 10 seconds that contains its address. The recipient node `N` inserts (or refresh) its local `link(@N,S,1` entry upon receiving the beacon message from `S`, and set the cost to be 1. Each `link` tuple has a soft-state TTL that can be customized based on the periodic beacon interval. Additional predicates can be added to the rule `d2` in order to specify neighbor selection policies, for instance, a node may limit its neighbors to trusted nodes or those with similar hardware configurations. The cost of each link can also be customized, e.g. based on link RTT or expected transmission count (ETX).

Since declarative networks utilize a distributed query engine to execute its protocols, *monitoring queries* can be expressed as *NDlog* rules to gather network statistics that depict the performance characteristics of the deployed protocol and the degree of mobility. For instance, a distributed recursive query can be used to compute network diameter, average link availability [17], average node degree, etc..

# 5. EVALUATION

In this section, we validate the *RapidMesh* system by developing and evaluating declarative implementations of LS and HSLS (Section 4) in simulation and emulation modes. In evaluating these two protocols, we measure per-node communication overhead (Kbps) and two well-known notions of route quality: *route stretch*, and *route validity*. Given the source and the destination of a routing request, the *stretch* of the route is the ratio of the hop count of the path selected by the routing algorithm to that of the optimal path given by the oracle with complete and instantaneous knowledge of the entire network topology. A route is *valid* if at the time it is computed from the local LSUs, the links that comprise the route are still up.

## 5.1 Simulation

Our first set of results involves executing declarative implementations of LS and HSLS within the ns-3 simulation environment. This enables us to study the protocols within a controlled environment using existing mobility models. In Section 5.2, we will repeat the same experiments in emulation using the exact same declarative specifications.

Our experiment setup consist of 35 nodes within an arena of size 550 meters by 750 meters, where nodes are configured to move at 0.5 m/s based on the random walk 2-dimensional model, often identified as a Brownian motion model. We use the identical mobility traces across all experiments in order to ensure we can compare the protocols based on a similar set of link updates.

All nodes communicate with other nodes using ns-3's 802.11b WiFi model with a range of approximately 100 meters. Based on this setup, network-wide link updates happen at an average rate of 1.3 events/second. The average node degree for the experimental duration is 5.6.

Each node executes the LS or HSLS protocols and the neighbor discovery protocol described in Section 4. We utilize *periodic* and *periodic with triggered* propagation modes. The latter approach combines both periodic flooding and triggered updates, which results in better route validity and stretch at the expense of increased communication overhead. The periodic flooding interval in LS and the nominal period of HSLS is set to 60 seconds.

Figures 2-4 compares LS and HSLS with periodic flooding based on the three evaluation metrics of per-node communication overhead, average route validity and stretch respectively. The Y-axis shows the respective metric, while the X-axis shows the elapsed time of the experiment (up to 500 seconds). To compute route validity and stretch, at every 5 second interval, we determine the fraction of routes that are valid at that instance, and for all valid routes greater than 3 hops, we compute the *average stretch* at the same instance.

Figure 2 compares LS and HSLS based on their per-node communication usage over time. Overall, as expected, HSLS incurs considerably less bandwidth due to the use of scoped flooding, incurring on average per-node communication overhead (sending traffic) of 0.29 kBps, as compared to 0.53 kBps for LS. Our HSLS implementation exhibits expected periodic scoped flooding behavior from the protocol specification. At more frequent intervals, HSLS flood peaks are lower compared to LS due to scoped flooding (at TTL of 2 and 4). These results are consistent with the expected protocol behavior and performance of LS and HSLS. Moreover, because of the scoped flooding, HSLS computes shorter paths compared to LS. The average path hop count computed by HSLS is of length 1.8, as compared to 2.2 for LS. The network-wide average longest paths (averaged every 5 seconds) computed by LS is 7.1 hops, as compared to 4.3 hops for HSLS.

Figure 3 shows the corresponding route validity of LS and HSLS. We observe that route validity is close to 1 whenever a network-wide flood occurs. However, the fraction of valid routes decreases rapidly (particularly in the case of HSLS) until the next flood occurs. Moreover, LS achieves higher route validity, achieving a route validity (averaged over the entire experiment duration) of 0.64 compared to 0.52 for HSLS.

Figure 4 shows the average stretch of all valid routes for LS and HSLS. Given the relatively small network, route stretch do not degrade significantly during the experiment. As a result, LS and HSLS achieve roughly equivalent stretch of 1.1, computed from all valid routes averaged over the entire experiment duration. In the case of LS, we observe that there is a sharp momentary increase in route stretch after 300s. This is an effect of the specific mobility pattern used in the experiment. At around 300s, nodes move towards each other to form a denser network. Since LS computes paths for the entire network, it is more vulnerable to sudden clustering of nodes compared to HSLS. In both cases, we note that both

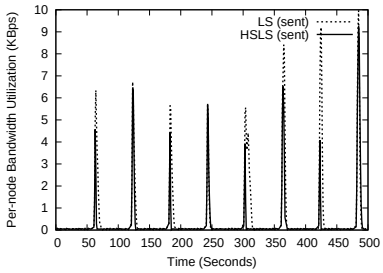


**Figure 2: Per-Node communication overhead (KBps) for LS and HSLS (periodic).**

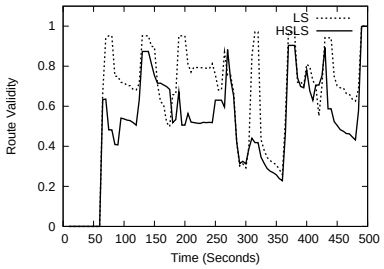


**Figure 3: Route validity for LS and HSLS (periodic).**

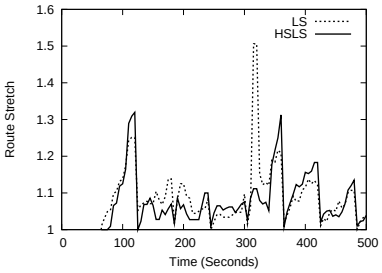


**Figure 4: Route stretch for LS and HSLS (periodic).**

protocols are able to recover and reduce route stretch to 1 at the next flood.

Overall, our experimental results validate that our declarative implementations of LS and HSLS achieve the expected performance vs route quality tradeoffs. In aggregate, LS incurs 1.8 times more bandwidth compared to HSLS, but improves overall route validity of HSLS by 19%. However, since LS utilize a more aggressive flooding strategy, we observe higher packet losses of 10% for LS compared to 4% for HSLS. Our results suggest that in a network with higher mobility, LS potentially will result in even higher losses, which may cause poor route quality compared to HSLS. Investigating the performance between these two protocols is an avenue of future work.

Figures 5-7 shows the corresponding results of executing LS and HSLS with triggered updates being used in between periodic floods. The basic specifications of LS and HSLS remain unchanged, with a few additional *NDlog* rules to trigger LSU updates when links are added or deleted. Overall, we similarly observe that the comparative differences between LS and HSLS that we observe in the earlier experiment. The additional use of triggered updates increase average per-node bandwidth utilization of periodic LS and HSLS by 17% and periodic HSLS by 21%. We note that when averaged across the entire experimental duration, route validity improved from 0.64 to 0.71 for LS, and from 0.52 to 0.57 for HSLS. This suggests that when used appropriately, triggered updates can result in overall improved route quality.

## 5.2 ORBIT Testbed

We execute the declarative implementations of LS and HSLS using the same generated ns-3 code running in emulation mode on the ORBIT wireless testbed [19]. This approach enables us to study the LS and HSLS protocols in an actual wireless environment, and also validate the observations drawn from simulations. The ORBIT testbed consists of machines with 1 Ghz VIA Nehemiah processors, 64KB cache, 512MB RAM, and supports two types of network adapters (Intel Pro-wireless 2915-based 802.11 a/b/g and Atheros AR5212-based 802.11 a/b/g). Nodes on the ORBIT testbed are placed a meter apart from one another in a grid and run with 1dBm transmit power.

Our evaluation is based on the *RapidMesh* system running in the emulation mode supported by ns-3. Each ORBIT machine executes an instance of a *RapidMesh* process running in ns-3's emulation mode. We utilize 35 testbed nodes with the Atheros adapter within a $7m \times 5m$ grid area. for our experiments. In both protocols, given our use of broadcast communication to disseminate LSUs to neighbors, RTS/CTS and retries are not invoked. We have selected 802.11a as it is less susceptible to interference on the ORBIT testbed compared to 802.11b.

Given that nodes on the ORBIT testbed are static, we emulate random walk 2-dimension mobility based on the neighborhood updates obtained from simulation traces with node speed of 0.15m/s. The neighborhood information is used to create the `link` table at each node. In each experimental run, we add and delete tuples from each node's `link` table based on mobility traces obtained from our above simulation runs. Since ns-3 emulation utilizes raw sockets, `iptables` are not applicable for filtering packets at the MAC layer based on each node's current set of neighbors. Instead, application-level filtering is done by each *RapidMesh* node which will filter incoming tuples to accept only tuples from nodes that are currently in its neighbor set determined by the current tuples stored in each node's `link` table.

This approach enables one to dynamically adjust neighborhood information on ORBIT even though the nodes are physically static. This flexibility however comes at the expense of the increased likelihood of transmission collision (and hence dropped frames) since each node's neighbors are not the ones that are physically closest on the grid. In all variants of link-state routing. to reduce the likelihood of collisions, we de-synchronize the time at which all nodes send out network-wide LSUs by spacing out the starting time of nodes. This reduces the peak bandwidth utilization when all nodes are sending LSUs to all other nodes. To reduce packet interference caused by potential simultaneous broadcast of LSUs, we add a random jitter of 0 to 500 milliseconds to every broadcast packet.

Figures 8-10 shows our experimental results for LS and HSLS with periodic flooding on ORBIT. Figure 8 shows that the peaks in per-node bandwidth utilization is lower compared to the corresponding simulation experiment in Figure 2. However, each network-wide flood last for a significantly longer period as noted from the width of each peak. This is in part due to the addition of packet jitter to desynchronize the flooding of LSUs. With the use of jitter, we note packet losses of 3.4% for LS and negligible losses for HSLS. Figures 9 and 10 shows that LS achieves better route validity compared to HSLS and roughly equivalent average route stretch. Moreover, route validity and stretch are significantly improved after each network-wide flood. These observations are consistent with our simulation results.

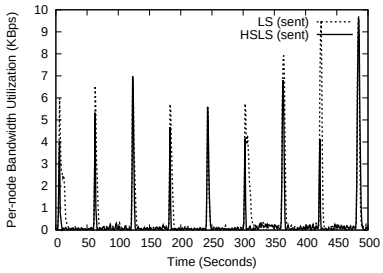


**Figure 5: Per-Node communication overhead (KBps) for LS and HSLS (periodic+triggered).**

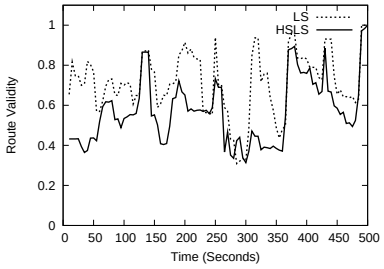


**Figure 6: Route validity for LS and HSLS (periodic+triggered).**

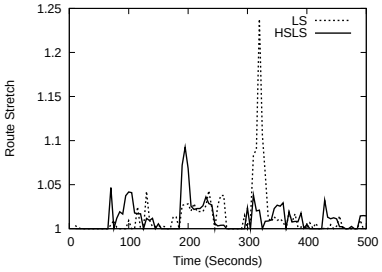


**Figure 7: Route stretch for LS and HSLS (periodic+triggered).**

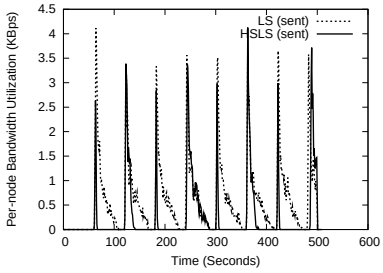


**Figure 8: Per-Node communication overhead (KBps) for LS and HSLS (periodic).**

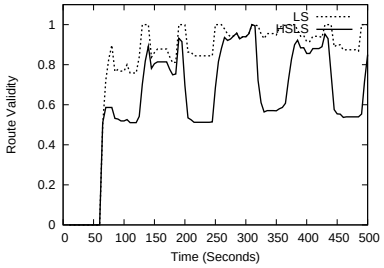


**Figure 9: Route validity for LS and HSLS (periodic).**

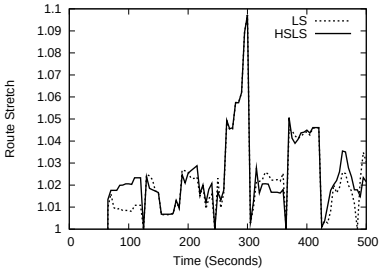


**Figure 10: Route stretch for LS and HSLS (periodic).**

Figures 11-13 shows the ORBIT experimental results for LS and HSLS with both periodic flooding and triggered updates being used. Similar to our prior observations, the bandwidth utilization is increased with the additional use of triggered updates. However, route quality has improved as a result. For instance, the average route quality of LS increased from 0.81 to 0.90 with the additional use of triggered updates. Similarly, the average route quality of HSLS increases from 0.63 to 0.69. These comparative differences are also consistent with that of our simulation results.

# 6. RELATED WORK

Prior to this paper, declarative networking has been studied primarily in wired environments, such as IP routing [11] and overlay network construction [10]. Recent work [3] has also demonstrated the feasibility of using declarative techniques to program sensor network protocols. The MANET settings present new challenges posed by the presence of mobility in the network. In addition, the variability of wireless environment presents compelling motivation for the use of declarative framework for synthesizing a variety of protocols and evaluate their performance/overhead tradeoffs under a variety of mobility patterns.

Reference [9] first proposes the use of declarative programming to prototype and adapt MANET routing protocols. This paper realizes the vision with the development of *RapidMesh* toolkit, with experimental validation in the ns-3 simulation environment and the ORBIT wireless testbed.

# 7. CONCLUSION

In this paper, we present *RapidMesh*, a declarative toolkit that enables one to rapidly specify, analyze and experiment with MANET protocols both in simulation and on an actual wireless testbed using a common code-base and runtime system. *RapidMesh* utilizes *declarative networking* [11, 10], a declarative, database-inspired extensible infrastructure that uses query languages to specify network behavior. Our initial experience of using *RapidMesh* to prototype and evaluate wireless routing protocols suggests that this is a promising approach. Declarative networking techniques can be used effectively to rapidly prototype protocols in a compact fashion, and one can rapidly deploy and evaluate the protocols in simulation and emulation. Moreover, the declarative framework enables the ability to rapidly explore a wide range of deployment and implementation parameters necessary for tuning the performance of MANET protocols.

Our immediate steps include further experimentation on the ORBIT wireless testbed, by studying the performance tradeoffs across a wide-range of reactive and epidemic protocols under different mobility patterns. In the near future, we plan to release *RapidMesh* as open-source for use in the research community. While our focus so far have been on validating *RapidMesh* on existing protocols, our ultimate goal is to develop a suite of declarative wireless protocols that can be used as building blocks by other researchers in designing their own protocols.

Another recent research initiative in the space of MANET protocol developments is *component-based routing* [6], which attempts to compose complex routing protocols from simpler components at a finer granularity than hybrid protocols.

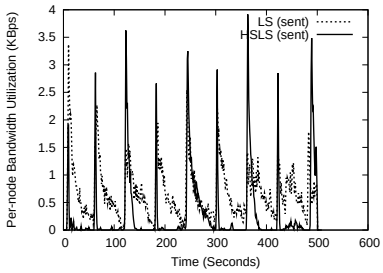


**Figure 11: Per-Node communication overhead (KBps) for LS and HSLS (periodic+triggered).**

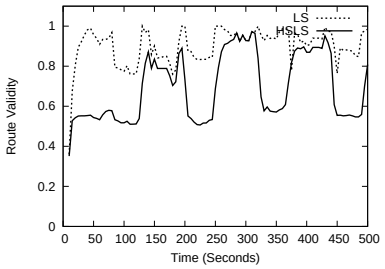


**Figure 12: Route validity for LS and HSLS (periodic+triggered).**

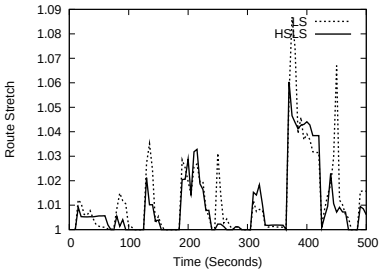


**Figure 13: Route stretch for LS and HSLS (periodic+triggered).**

Their goal is adaptability to the dynamic environment in MANETs, the focus is on the diagnosis and the subsequent improvement of a weak protocol component. Interestingly, the declarative networking language enables composability via the abilities to define predicates to be used by other rules. This has been shown in prior work [12] to be useful for synthesizing new networks from components, suggesting its applicability to component-based routing.

## 8. ACKNOWLEDGMENTS

This work is based on work supported in part by NSF grants CNS-0721845, CNS-0831376, CCF-0820208, and CNS-0845552.

## 9. REFERENCES

[1] P2: Declarative Networking System. http://p2.cs.berkeley.edu.

[2] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Mobicom*, 1998.

[3] D. C. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *5th ACM Conference on Embedded networked Sensor Systems (SenSys)*, 2007.

[4] T. Clausen and P. Jacquet. Optimized link state routing protocol (olsr). In *RFC 3626 (Experimental)*, 2003.

[5] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. B. Kopena. Network simulations with the ns-3 simulator. In *SIGCOMM Demonstration*, 2008.

[6] H. Huang and J. S. Baras. Component based routing: A new methodology for designing routing protocols for manet. In *25th Army Science Conference*, 2006.

[7] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, volume 353. 1996.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[9] C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo. A declarative perspective on adaptive manet routing. In *ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow*, 2008.

[10] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *ACM SOSP*, 2005.

[11] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *ACM SIGCOMM*, 2005.

[12] Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. MOSAIC: Unified Declarative Platform for Dynamic Overlay Composition. In *4th Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT)*, 2008.

[13] Network Simulator 3. http://www.nsnam.org/.

[14] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, LA, 1999.

[15] R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[16] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *SIGCOMM*, pages 15–25, 1999.

[17] R. Ramanathan, R. Hansen, P. Basu, R. Rosales-Hain, and R. Krishnan. Prioritized epidemic routing for opportunistic networks. In *ACM MobiOpp '07*, pages 62–66, San Juan, Puerto Rico, 2007.

[18] C. Santivanez, R. Ramanathan, and I. Stavrakakis. Making link-state routing scale for ad hoc networks. In *ACM MobiHoc '01*, Long Beach, CA, 2001.

[19] O. W. N. Testbed. `http://www.winlab.rutgers.edu/docs/focus/ORBIT.html`.

[20] A. Vahdat and D. Becker. Epidemic routing for partially-connected ad hoc networks. Technical Report CS-200006, Duke University, 2000.

[21] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Towards declarative network verification. In *11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2009.

# APPENDIX

# A. ADDITIONAL EXAMPLES

## A.1 Reactive Protocol: Source Routing

Next, we demonstrate a reactive protocol based on DSR [7]. The following set of rules show the route discovery of DSR (rules `dsr1-4`) followed by the route response (rules `dsr5-6`) traversing the best reverse path from destination to source.

```
dsr1 eRouteReq(@N,S,D,P,C) :- eQuery(@S,D),
          link(@S,N,C), P=f_init(S).
dsr2 eRouteReq(@Z,S,D,P,C):-
          shortestRoute(@N,S,D,P1,C1),
          link(@N,Z,C2),
          C=C1+C2, P=f_concatPath(P1,S).
dsr3 minCost(@N,S,D,min<C>) :-
          routeReq(@N,S,D,P,C).
dsr4 shortestRoute(@N,S,D,P,C) :-
          minCost(@N,S,D,C),
          routeReq(@N,S,D,P,C).
dsr5 eRouteReply(@Z,S,D,P2,P1,C) :-
          eRouteReq(@N,S,D,P2,C), N==D,
          Z=f_last(P2),P1=f_removeLast(P2).
dsr6 eRouteReply(@Z,S,D,P,P1,C) :-
          eRouteReply(@Z,S,D,P,P2,C),
          Z=f_last(P2),
          f_size(P2)>0, P1=f_removeLast(P2),
```

In DSR, a requesting node `S` issues an initial route request, denoted by `eQuery(@S,D)` event in rule `dsr1`. This results in a `eRouteReq` message tuples that is generated and recursively forwarded along all links (rules dsr2). The `routeReq` table is used to cache current route requests. To prune unnecessary paths, rules `dsr3-4` ensures that only the shortest path from the initial node `S` to the intermediate node `N` is maintained.

Upon reaching the destination node `D`, rule `dsr5` generates a `eRouteReply` message that is sent back recursively via rule `dsr6` along the computed best reverse path back to the requesting node `S`. The functions `f_last` and `f_removeLast` return and remove the last node from a path respectively. Rule `dsr6` reaches the initial requesting node `S` when the remaining path length is 0.

The rules for AODV [14] share similarities with DSR above, where only the next hop rather than the entire path is maintained.

## A.2 Epidemic Protocols

Epidemic routing has been proposed for reliable delivery in intermittently connected MANETs (a class of disruption tolerant networks or DTNs). A key reliability component of such protocols is the summary vector exchange as illustrated by the rules `e1-4` below:

```
e1 eBitVecReq(@Y,X,V):- summaryVec(@X,V),
          eDetectNewLink(@X,Y).
e2 eBitVecReply(@X,Y,V):- eBitVecReq(@Y,X,V1),
          summaryVec(@Y,V2),
          V=f_vec_AND(V1,f_vec_NOT(V2)).
e3 eNewMsg(@Y,I,S,D):- eBitVecReply(@X,Y,V),
          msgs(@X,I,S,D),
          f_vec_in(V,I)==true.
e4 msgs(@Y,I,S,D):- eNewMsg(@Y,I,S,D).
```

In rule `e1`, node `X` detects that a new link comes to be available, then it retrieves its local (`summaryVec`) table, consisting a bit vector where the $i$th bit denotes the receipt of the $i$th message, and then generates a `eBitVecReq` request to the neighbor `Y` connected by the new link. Upon receiving the request, node `Y` performs a bitwise AND operation (`f_vec_AND`) between the incoming summary vector `V1` and the negation (`f_vec_NOT`) of local summary vector `V2` to generate a new vector `V` which is sent back to `X`. This new vector `V` denotes messages seen by `X` but not `Y`. Rules `e3-4` then enables node `X` to filter local messages to be sent based on the bit vector `V` stored in the reply, which are then buffered in the local `msgs` table for transmission.