Simulation and Evaluation of the Reactive Virtual Node Layer

by

Mike Spindel

B.S., Massachusetts Institute of Technology (2007)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2008

Author . . . .
Department of Electrical Engineering and Computer Science
May 28, 2008

Certified by . . . .
Nancy Lynch
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Simulation and Evaluation of the Reactive Virtual Node Layer

by

Mike Spindel

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

Developing software in a wireless, ad hoc environment is an intrinsically difficult problem. One way to mitigate it is to add an abstraction layer between the software and the individual mobile devices. This thesis describes one such abstraction, the Reactive Virtual Node (RVN) Layer [1, 2, 3, 4], as well as a new simulation framework written in Python. Additionally, this thesis uses the simulator to characterize an RVN-based routing service for multihop mobile ad hoc networks. The performance of the routing service is compared to the Ad hoc On-Demand Distance Vector routing protocol, as well as a greedy geographic routing protocol.

Thesis Supervisor: Nancy Lynch
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis would have never been written without the help and support of many, many people. I would like to take a moment to thank some of the people that helped make it happen.

First, I would like to thank Nancy Lynch for hiring me as a UROP years ago, for introducing me to virtual nodes, and for providing invaluable editing and support throughout the entire thesis writing process.

I would also like to thank the rest of the Theory of Distributed Systems group. In particular, I would like to thank Calvin Newport for his feedback on this thesis and his help while working on the VN simulator.

My parents have been extremely supportive and patient throughout this entire process, and their encouragement has been invaluable to me.

Additionally, I would like to thank everyone who has asked me, "Are you done yet?" over the last few months. Without you, I may have never actually finished.

Contents

List of Figures

Chapter 1

Introduction

PDAs and laptops are prolific today, and cellular phones with WiFi support are becoming more common all the time. However, the infrastructure necessary to connect them has limits. 802.11b access points are rarely available outside of homes and businesses, and there are many places where it simply isn't feasible or economically sensible to install permanent equipment. One solution to this problem is to build networks without using fixed infrastructure by allowing nodes to talk directly to each other.

This sort of unplanned, peer-to-peer system is called a mobile ad-hoc network (MANET), and there is no shortage of applications for them. Consider the problem of coordinating emergency responders: After a natural disaster, effective communication systems are crucial to coordinating an effective response. Yet, existing infrastructure might be damaged or destroyed and constructing a new network can be prohibitively time consuming. MANETs promise to dramatically reduce setup time in these situations.

The same properties that make ad-hoc networks compelling for first responders make them well suited to rapidly building community wireless networks. Mesh networks are already in widespread use for this purpose. For example, the Freifunk [5] group in Germany and the Funkfeuer [6] group in Austria operate large mesh networks. The MIT Roofnet [7] project operates an unplanned mesh network in Cambridge, and there are dozens of similar projects across the world. Several companies are actively building wireless networks using using ad-hoc technology. Meraki Inc. [8] is a spin-off from the MIT Roofnet project. Metrix Communications [9] offers similar services.

Applications for ad-hoc networks stretch beyond routing, as well. Consider a network of small radio-enabled devices monitoring the environment. Ad hoc networks could coordinate automobiles, airplanes, or even robots.

Yet, any project that uses ad-hoc networks has to deal with numerous challenges. The underlying radio network is unreliable. The system needs to automatically configure itself and be robust against message loss. The composition of the network can change over time as nodes join and leave. Members of the network might walk into an elevator or behind a tree. As a result, designing and verifying peer-to-peer algorithms that run on ad-hoc networks is complicated and time-consuming.

1.1 Virtual Infrastructure

One strategy to make it easier to develop this kind of software is to build a well-behaved abstraction layer on top of the physical network. The abstraction can shield the programmer from many of the difficulties intrinsic to working with ad-hoc networks. This thesis is concerned with evaluating a particular abstraction: the Reactive Virtual Node Layer.

The Reactive Virtual Node (RVN) Layer builds on several other other virtual infrastructure designs. Its first recognizable form was the GeoQuorums [10] approach to implementing atomic memory, which is based on geographic "focal points" of mobile nodes that emulate the memory. GeoQuorums evolved into a full-fledged abstraction for distributed computation called virtual mobile nodes (VMN) [1]. The VMN extends the focal point to an abstract node that can move on a fixed path and execute arbitrary algorithms. That approach was further extended in [2] to allow the virtual node to move in an autonomous manner. In [4], the virtual node concept takes the form of a timed, stationary automata (VSA). Each VSA maps to a specific geographic area and provides a near real-time clock.

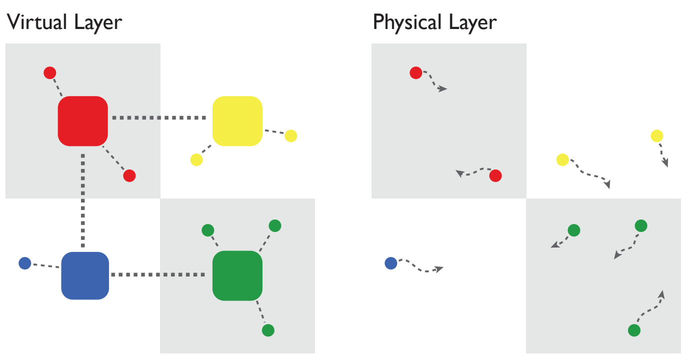


Figure 1-1: Virtual Stationary Automata. The figure on the left shows the virtual layer. Rounded rectangles represent virtual nodes and small circles represent client nodes. The physical reality is shown on the right: A number of mobile nodes move throughout the plane.

Although these designs differ in virtual node capabilities and implementation, they share a basic decomposition into virtual and client nodes. Client nodes reflect a client program run explicitly on the physical device. Client nodes are allowed to move freely and fail unpredictably. On the other hand, virtual nodes reflect a stable abstraction. They are defined by a particular geographical region (sometimes mobile, sometimes not) and are emulated by the physical devices inside the region using distributed algorithms such that the state of the virtual node is robust against the failure of individual devices. Of course, the virtual node can fail under certain conditions: If every physical device leaves its region, it necessarily loses state. Furthermore, the virtual node can be unresponsive for short periods of time if a leading device leaves the region without performing any kind of handoff.

This thesis is focused on a simplified variant of the VSA called the Reactive Virtual Node Layer [11]. Unlike the VSA, reactive nodes can transmit only in response to client messages and do not provide a clock. This simplification is intended to make the virtual nodes easier to implement and easier to program.

The reactive virtual node abstraction was initially implemented in Python and deployed on several iPaq PDAs for testing. The tests involved two basic applications: a counter and a simple traffic light. These small scale tests validated the implementation to some degree. Yet, they had a number of limitations. Namely, there were too few iPaqs to get a good sense of the behavior of large networks. The physical nature of the iPaqs was also problematic because it was difficult to construct and test a wide range of scenarios.

In order to test a larger number of nodes in a variety of different scenarios, the existing virtual node emulator (VNE) implementation was overhauled and modified so that it could run in a simulated environment. For convenience, the simulator [1] is also written in Python and provides an easy to use testbed for mobile ad-hoc network systems.

1.2 Ad hoc Routing

In addition to presenting the simulator, this thesis investigates the properties of the Reactive Virtual Node Layer by using it to implement a routing protocol for mobile ad-hoc networks.

Traditional routing protocols tend to fall into one of two categories: distance vector or link state.

In a distance vector protocol, routers broadcast the destinations they can reach to neighboring peers. Each route has an associated cost metric, and each router uses information from its neighbors to build a table, which maps every destination to the optimal next hop. One variation on this strategy, called path

[1] Up-to-date source code and documentation for the simulator and RVN framework can be found at `https://carbide.mit.edu/trac/`.

vector routing, stores the entire path in the routing table rather than just the next hop. This modification makes it much easier to avoid routing loops.

Link state protocols broadcast connectivity information (in other words, the state of each router's links) throughout the network. Each router maintains a picture of the network and runs a shortest path algorithm to decide the optimal hop for each destination.

All of these approaches are reflected in MANET protocols. However routing in MANETs is substantially more difficult than in typical wired networks, and the protocols reflect the disparity. MANETs are much more dynamic and they have to worry about limited power, limited channel capacity, and high latency. This means that traditional distance vector protocols have problems with routing loops and traditional link state protocols have trouble maintaining a current, coherent map across the entire network.

Dynamic-Sequenced Distance Vector (DSDV) [12] is a distance vector protocol that tags routes with sequence numbers in order to avoid routing loops. DSDV is a proactive protocol, which means that it actively maintains complete routing tables at each router, even when nodes are not transmitting messages.

Proactive protocols benefit from having a complete routing table available at all times, so there is no latency associated with route lookup when a message is sent. Routing protocols for wired networks are typically proactive. However, in wireless networks, the cost of maintaining the table can overshadow the benefit. This observation leads to reactive, or on-demand, protocols that postpone route discovery and maintenance until the route is actually needed. For example, Ad hoc On-Demand Distance Vector (AODV) [13] is an on-demand variant of DSDV. Some protocols take a hybrid approach as well. For example the Zone Routing Protocol (ZRP) [14] uses a proactive protocol to communicate between nearby nodes and an on-demand protocol for distant ones.

Dynamic Source Routing (DSR) [15] is an on-demand, path vector protocol. As the name suggests, messages are source routed, which means that each messages route is specified in the message's header, which makes it easy to avoid routing loops.

Link state protocols are also used in MANET routing. Optimized Link State Routing (OLSR) [16] is a proactive link state protocol that attempts to optimize for MANETs by minimizing the size of control messages and by minimizing the flooding of control messages. OLSR is particularly notable because it is widely used in community wireless projects.

Different MANET protocols also vary the granularity of their routing schemes. The protocols discussed thus far route in terms of individual nodes. However, wireless networks lend themselves to other levels of granularity as well. In particular, some protocols attempt to route in terms of a geographical location or a physical trajectory.

Greedy Perimeter Stateless Routing (GPSR) [17] is a good example of a geographic routing protocol. It attempts to route towards a physical location using a greedy policy - in other words, it sends a message to the neighbor nearest the target location. When that is not possible, for example, when every neighboring node is farther away from the destination than the current node, it activates a second transmission mode to route around the perimeter.

Yet, the number of different routing protocols continues to multiply. Why? The problem is that none of the available systems really meet the performance, scalability, or security requirements necessary for large MANET deployments. Solving this problem requires research in a variety of directions, including better radio technology, better MAC protocols, and better routing techniques.

This thesis is concerned with optimizing along a particular axis: software complexity. All of the routing protocols discussed above route at the level of physical nodes, which means that they inherit all of the complexity intrinsic to managing an individual nodes connectivity. As MANET protocols evolve in terms of performance, they are likely to become even more complicated. This thesis proposes a solution: Use the RVN Layer as the basis for a MANET routing algorithm.

1.3 Organization

In Chapter 2, I discuss the Reactive Virtual Node Layer in detail and describe the various assumptions that it makes. In Chapter 3, I discuss the implementation of the simulator and the implementation of the Virtual Node Emulator. In Chapter 4, I illustrate how to use the simulation environment by walking through how to use the simulator to measure VN overhead. In Chapter 5, I use the simulator to test the compare the performance of an RVN-based routing scheme, AODV, and a simple geographic routing scheme. I conclude in Chapter 6.

Chapter 2

The Reactive Virtual Node Layer

The Reactive Virtual Node (RVN) Layer provides software developers with a stable abstraction for programming in MANETs. Rather than explicitly considering individual physical nodes, a programmer can write software for a stationary, reliable, *virtual* node. These nodes, called RVNs, are emulated by nearby physical nodes. This chapter begins by defining the physical model that the RVN Layer assumes, and it proceeds by describing and defining the roles of client and virtual nodes.

2.1 Physical Model

The RVN Layer is intended to run on top of a variety of potentially different physical devices. This situation is modelled as a finite set $Q = \{q_1, q_2, ..., q_{pmax}\}, pmax \in \mathbb{Z}^+$ of mobile timed input/output automata [18] moving in the 2D plane. The set $P = [1, pmax]$ is the set of mobile node indices. Mobile nodes are allowed to move arbitrarily. A node q_p's position is referred to as $loc(p)$, and its speed is bounded by a constant v_{max}.

Every node receives information about its current position and the global time every ϵ_{sample} time. Every node has a local clock that proceeds at the rate of real time and is resynchronized every ϵ_{sample} time. This update service is intended to model the information provided by a GPS positioning system.

Physical nodes also have the ability to do essentially arbitrary computation. For the sake of simplicity, I assume that all local computation does not take any time. This assumption is reasonable given the type of computation required by the algorithms under study: the power and time requirements will always to be dominated by network activity. Nodes are allowed to suffer from stopping failures. Once a node stops, then it halts all local computation and does not send any more messages.

Each mobile node has access to a broadcast service, P-bcast. Because the RVN Layer is intended to run on top of a variety of devices in a variety of environments, it is not reasonable to assume a fixed transmission radius. Rather, every mobile node is allowed to have a different broadcast range depending on both geographical location and orientation: Define the set of functions $trans\_r_p : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}$ on two coordinates *src* and *dst* such that $trans\_r_p(src, dst)$ is the maximum reliable transmission distance for node q_p, $p \in P$ at position *src* toward position *dst*.

P-bcast guarantees two properties: reliable local delivery and integrity. *Integrity* guarantees that every message received was previously broadcast. *Reliable local delivery* guarantees that every message broadcast by a node will be received by every in-range node in a timely manner: Every message broadcast by node q_p will be received within d seconds by every node q_q such that $|loc(p) - loc(q)| \leq trans\_r_p(loc(p), loc(q))$ for the entire transmission period.

2.2 Regions

A particular RVN is contained by its geographical *region*. A virtual node region is a geographical area having the property that every radio in the region can reliably broadcast to, and receive data from, every other radio in the region and neighboring regions.

The location of virtual nodes and their corresponding regions is global knowledge.

2.2.1 Definitions

The set J consists of all region names.

The function $region : \mathbb{R}^2 \rightarrow J \cup \{reg_{nil}\}$ maps a point to the corresponding region name. If a node is not in any region, it returns the symbol reg_{nil}.

Let the function $nbrs : J \rightarrow 2^J$ return the set of all of a region's neighbors. For all neighboring regions, the following constraint must hold: for all node indices p in P, every point r such that $region(r) = j$, and every point r' such that $region(r') \in \{j\} \cup nbrs(j)$, $|r - r'| \leq trans\_r_p(r) - 2\epsilon_{sample} \cdot v_{max}$.

2.3 Virtual Broadcast

Clients and RVNs have access to an additional broadcast service, V-bcast, that provides additional guarantees and capabilities on top of P-bcast. V-bcast allows clients and RVNs to broadcast to other nearby clients and RVNs. V-bcast inherits the integrity property from P-bcast and has an analogous reliable local delivery property: V-bcast guarantees that if a client or RVN in a region j broadcasts a message, then every client and RVN in $\{j\} \cup nbrs(j)$ for the entire transmission period d will receive the message.

V-bcast provides two additional guarantees. V-bcast guarantees the *inverse* of reliable local delivery: If a client is not in a region neighboring a broadcast it will not receive the message. V-bcast also guarantees that messages will have a *total ordering*: All broadcast messages will be received by clients and RVNs in the same order.

Since the virtual broadcast service has a maximum range bounded by the physical broadcast range of every participating physical node, the size of regions and the choice of neighbors is constrained as described in Section 2.2.1.

2.4 Client Nodes

Client nodes are the manifestation of physical nodes in the RVN abstraction. As such, the capabilities of a client correspond very closely to those of a physical node. Clients are modelled as the set $C = \{c_1, c_2, ..., c_{pmax}\}$, $p \in P$ of timed input-output automata. Each client has access to the location of its host physical node. It also has access to a real-time clock. However, a client can communicate only via the V-bcast service.

This definition creates a one-to-one relationship between client nodes and physical nodes. Although it makes conceptual sense to allow a single physical node to run multiple clients, the definition leaves the ability out for the sake of simplicity. This restriction does not result in a loss of generality because multiple client programs can simply be merged into a single, monolithic program.

2.5 Reactive Virtual Nodes

An RVN v_j is an event-driven automaton running in region j. RVNs are receive-event-driven, which means that their operation is defined purely in terms of a `msgReceived` handler. When called with a received message, the handler can arbitrarily change the RVN state and optionally return a set of messages to be broadcast with V-bcast. RVNs do not have any other external actions or mechanisms to change state. Notably, RVNs do not have direct access to a clock.

Because an RVN is emulated by the physical nodes in its region, it is natural to understand an RVN's failure modes in terms of the behavior of physical nodes. In order to capture this in terms of the RVN Layer abstraction, an RVNs failure modes are defined in terms of clients.

1. If there are no active clients in region j, the RVN v_j has failed.

2. If an active client is continuously in a failed region j for at least $t_{restart}$, then the RVN v_j restarts.

Chapter 3

Simulation

3.1 Overview

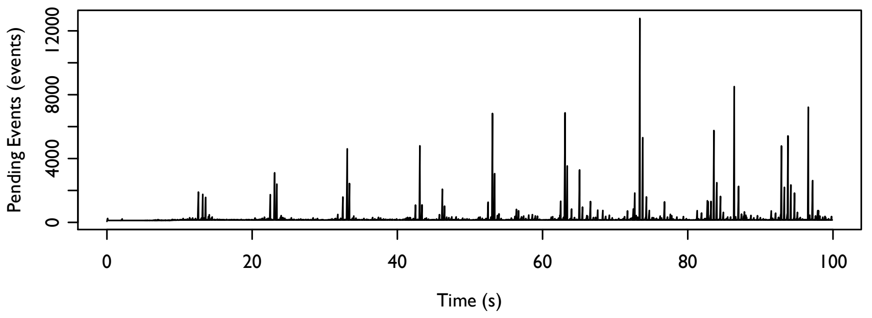
The simulator is divided into two parts: a lightweight, protocol-agnostic event simulator and the Virtual Node Emulator (VNE) implementation. The event simulator provides a number of services useful for simulating simple ad-hoc networks. In particular, it provides a mobility system for simulated mobile nodes that is compatible with standard ns-2 mobility traces, and it provides a broadcast medium simulation for transmitting between nodes. The medium can be configured to simulate an unreliable connection in which a certain percentage of packets are dropped.

The simulator provides a simple, process-oriented interface for simulated software. The simulated mobile nodes simply run a set of "tasks" defined by a scenario file. The tasks are implemented using tasklets in Stackless Python [19] – essentially coroutines – and an explicit event queue. In order to keep simulated code as close to reality as possible, the simulator provides dynamically loadable classes that override common function calls with simulated analogues. For example, calls to `time()` return simulation time and calls to `sleep()` reschedule a task during simulation.

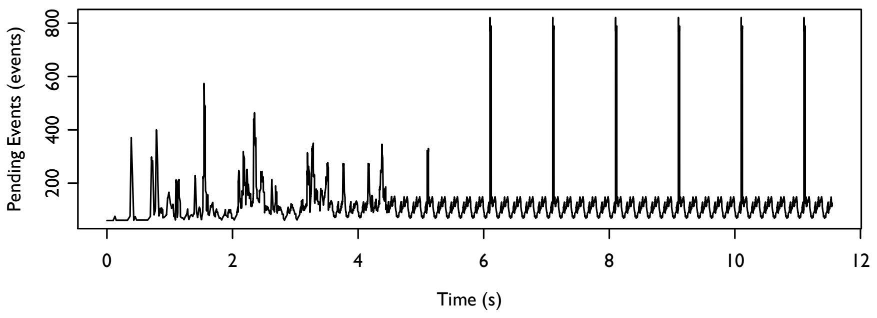
3.1.1 The Pending Event Set

Perhaps the heart of any discrete event simulator is the actual list of pending events. Different protocols and system configurations can generate dramatically different event profiles, and it is crucial that the event set implementation efficiently support a wide variety of different distributions of timed event sequences.

The problem is particularly aggravated by the virtual node layer. Figure 3-1a illustrates the characteristic distribution of event times generated by VN routing simulations. The VN framework is very process-



(a) VNE Event Distribution



(b) AODV Event Distribution

Figure 3-1: Event distributions from two representative simulations

heavy and has a number of different polling loops, which leads to very large, very brief peaks of activity. This stands in contrast to other routing schemes. For example, Figure 3-1b was generated during an AODV routing simulation. It shows chaotic behavior while routes are being established and periodic behavior during normal operation. However, the AODV implementation is much less process-oriented, so the peaks are much smaller.

A simulator event set needs to support three key operations: It must be able to QUEUE new events, to DEQUEUE the earliest pending event, and to REMOVE any particular event. In addition to these operations, a good event set will be stable, in other words, two events scheduled for the same time will be dequeued in the same order that they are enqueued. Although stability certainly is not required for simulation correctness, it does make simulations easier to debug.

Pending event sets are sufficiently important that dozens of implementations are described in the literature. While developing this simulator, I tested three: A simple implementation built on Python heaps, the calendar queue [20] implementation from ns-2, and an implementation of a sluggish calendar queue [21].

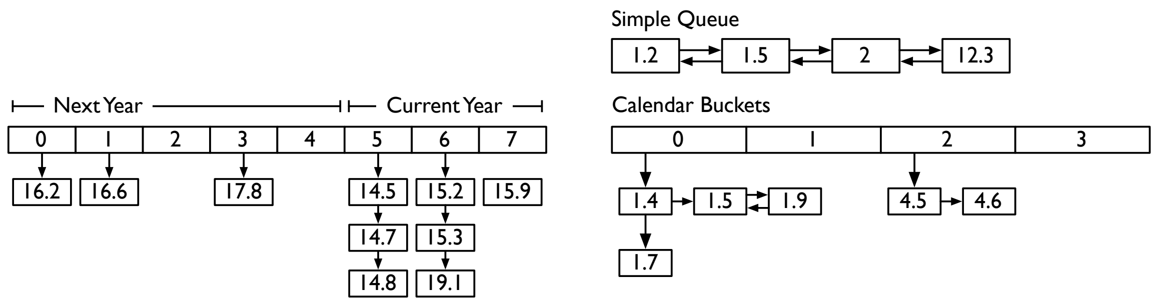
The first event set is a binary heap backed by a Python list. Binary heaps support ENQUEUE, DEQUEUE, and REMOVE operations in $O(\log n)$ time, which means that they scale reasonably well to large sets. However, heaps are not intrinsically stable, and the events have to be annotated with an additional counter to differentiate events scheduled at the same time. The heap operations for this implementation are provided by the `heapq` library that is distributed with Python. The library is written using Python's C API and is quite fast.

The design of the calendar queue, illustrated in Figure 3-2a, was inspired by the ubiquitous desk calendar. In a desk calendar, events are scheduled by simply writing them on the page corresponding to the appropriate day. There is a page for every day of the year, and if you need to schedule something for a future year, you can note the year next to the event. The computer equivalent of the desk calendar is an array of linked lists. Each list represents a metaphorical "day" in the array. Events in the distant future are allowed to wrap around. The structure avoids excessive sorting, seeking, and any notion of an overflow list.

Calendar queues support the standard event set operations in $O(1)$ average time. However, in order to prevent the lists for individual days from getting too long, the queue needs to resize itself periodically by adjusting the length of days and years. Nonetheless, performance can suffer noticeably if events are heavily clustered in just a few buckets.

The calendar queue implementation that is tested here is derived from the ns-2 simulator. The queue is implemented in C++, and a thin wrapper was written using Python's C API. It is important to note that the performance figures for this queue are not necessarily representative of the current version of ns-2. Although the implementation tested here is standard in ns-2 version 2.32, the calendar queue was substantially updated in ns-2 version 2.33.

Sluggish calendar queues are another data structure that attempts to provide performance superior to standard calendar queues. The design, illustrated in 3-2b, is inspired by the observation that most events



(a) Calendar Queue (b) Sluggish Calendar Queue

Figure 3-2: Calendar Queue vs. Sluggish Calendar Queue

in network simulations are enqueued in increasing order. Sluggish queues take advantage of this tendency by integrating a basic list queue with a calendar queue. The queue tries to add new events to the list first. If the event does fit within three elements of either the front or back of the list, the list is moved into the calendar queue and the new event is used to initialize the new list. The upshot is that as long as the simulation queues things in forward order, queue operations are performed only on the linked list. A beneficial side effect is that the structure tends to handle large bursts of events very efficiently. This queue is implemented directly in C and uses a thin Python wrapper.

Each event set implementation was tested against the two event distributions in Figure 3-1. The VN sequence has roughly 300,000 operations split evenly between ENQUEUE and DEQUEUE, as well as a handful of REMOVE operations. The AODV sequence has five million operations split evenly between ENQUEUE and DEQUEUE.

Each implementation was tested by replaying these representative event sequences and timing the total execution time. The structures are implemented in either C or C++, but they are used in a Python simulator. In order to understand the penalty imposed by the Python runtime, each event set was tested directly in its native language and in Python. The test sequences were loaded into memory prior to testing, but the measurements intentionally include the cost of allocating memory for event handles. All measurements were performed on a 2.4 GHz Intel Core 2 Duo. All of the Python tests were run with garbage collection disabled and the Psyco JIT compiler enabled.

The measurement results are collected in Figure 3-3, and the most dramatic result comes from the calendar queue. It is nearly 20 times slower than the sluggish queue in the native tests and four times slower in Python tests because the queue's resize and seeking heuristics have trouble with the peaks generated by the VN simulation. In general, the sluggish calendar queue shows the best, and most consistent, performance of the event sets. However, the penalty imposed by the Python runtime renders the sluggish queue implementation almost indistinguishable from directly using heaps. Profiling the Python runtime revealed that most of the difference was caused by the additional object allocation, the Python event loop, and Python method dispatch.

The result of this testing is that the sluggish calendar queue implementation is used by default. Because the sluggish queue requires a C module to be compiled, it's possible that it won't be available. If the simulator cannot find the sluggish queue module, it falls back on the stable heap implementation. The performance figures also suggest that this is close to the best event set performance that can be achieved while

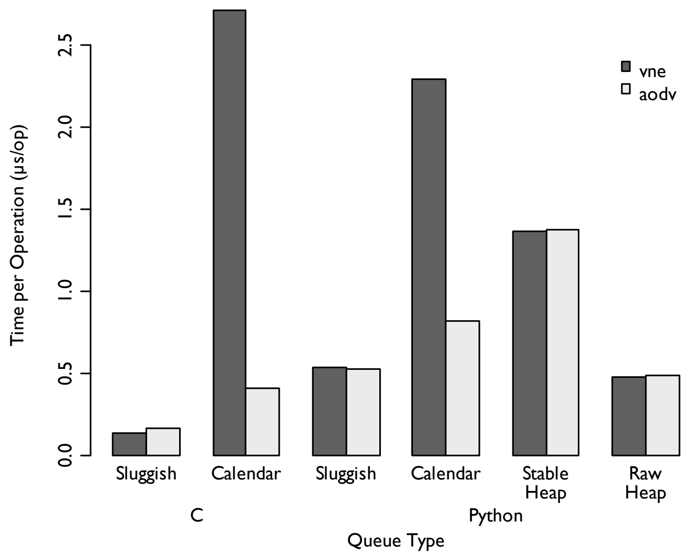


Figure 3-3: Event Set Performance

using the Python dispatch mechanism.

3.1.2 Physical Model

The simulator's channel and radio propagation models, such as they are, are lumped into a single module called the `BroadcastMedium` and provide the following capabilities:

Transmission Radius It is possible to specify a maximum transmission radius for nodes.

Transmission Lag Transmission lag can be configured: The simulator has provisions for specifying a constant transmission lag for local and remote reception. It also has provisions for distributing remote broadcast lag over a normal distribution.

Message Loss Message loss can be modelled probabilistically: the simulator is capable of simply dropping a certain percentage of messages in transit. The intention is to provide the ability to observe how a protocol reacts to message loss without providing the (misleading) illusion that the observed drop pattern is physically accurate.

The `BroadcastMedium` also provides a convenient place to add other propagation related features. For example, the simulator also implements a feature called the "Faraday cage" that allows a particular mobile node to be cut off from radio communication for a period of time.

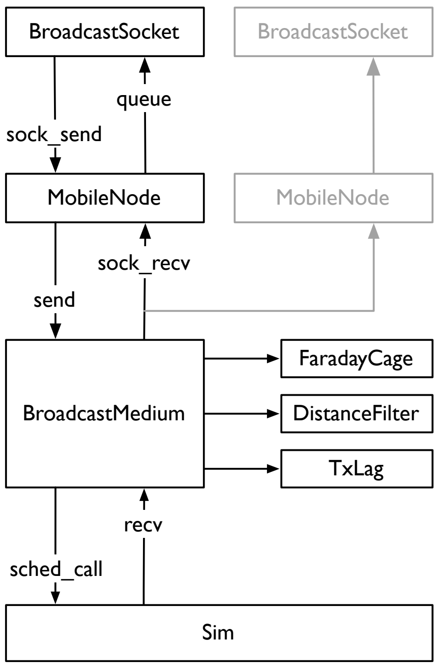


Figure 3-4: The Simulator Network Stack

Notice that the simulator does not implement an antenna model or a detailed model of radio propagation. Nor does it incorporate a physically accurate channel model or any kind of collision modelling. However, the simulator architecture and language are intended to make these features straightforward to add. This design decision was primarily influenced by concerns about development time and the performance cost implied by an interpreted language.

3.1.3 Networking Model

The simulator networking stack, illustrated in Fig 3-4, is intended to balance simplicity and flexibility.

Messages, represented by arbitrary Python objects, are originated by `MobileNode` objects and passed to the `BroadcastMedium`. The `BroadcastMedium` performs the tests described in the previous section to determine whether or not the message needs to be dropped. Next, a call to the `BroadcastMedium`'s `recv` method is scheduled in the main simulator event queue. When the event is triggered, `BroadcastMedium` has another chance to filter the event before passing it back to in-range `MobileNode` objects.

By design, `BroadcastMedium` offers a simple and generic interface for message transmission. However, it is not particularly pleasant to use directly from simulated software. For this purpose, the simulator offers a simplified version of the BSD sockets interface, called `BroadcastSocket`.

In order to support IP routing protocols, AODV in particular, the simulator contains an extremely minimal IP stack. The stack currently includes modules that perform port multiplexing with UDP and addressing with IP. The implementation takes advantage of the fact that messages are just Python objects to avoid serialization and deserialization performance penalties.

Listing 3.1 demonstrates the code necessary to instantiate a `MobileNode` with a simple IP stack. Although the simulator provides convenient methods that can do much of this wiring for the user, this example elucidates how the simulator threads necessary state among objects.

One of the organizing principles of the simulator implementation is the *default instance*. Typical usage requires only one simulator object, and it would be tedious to always refer to it explicitly. Yet, sometimes, for example when constructing unit tests, it is necessary to create additional instances. In order to accommodate both of these goals, the simulator provides default, module level instances of important classes. For example, in Listing 3.1, `sim.MobileNode(addr)` creates a node in the context of the default `Sim` object.

Code Listing 3.1 (Instantiating a Mobile Node with a simple IP stack)**.**

```python
import sim, simsockets, aodv

def make_node(addr):
    n = sim.MobileNode(addr) # Create a MobileNode
    mac = simsockets.MacNull(n, addr) # Connect a MAC protocol to node n
    ip = simsockets.IP(mac, addr) # Connect an IP interface to mac
    udp = simsockets.UDP(ip) # Connect a UDP interface to ip

    ip.route = aodv.Aodv(n, ip, udp) # Set the IP routing protocol to AODV
    ip.protocols[simsockets.PROTO_UDP] = udp.queue # Set the UDP message handler

    return (n, udp)
```

The rest of the code listing wires together a simple IP stack. The stack is composed of a MAC interface, an IP interface, and a UDP interface. Addresses are represented as arbitrary Python objects. Although there is nothing preventing a simulation from using proper IP or Ethernet addresses, it isn't required. This design makes it easy to set up a quick simulation with integer node addresses. The `MacNull` protocol is an empty MAC protocol. It serves only to add an extra addressing layer to allow routing protocols to be accurately expressed. When the `MacNull` object is instantiated, it is automatically connected to the `MobileNode`'s `queue` method. Similar connections are made when the `IP` and `UDP` objects are instantiated. The `Aodv` object encapsulated behavior for the AODV routing protocol. It's specified by setting the `route` field in the `IP` object. The `IP` protocol dispatch is controlled by the `protocols` dictionary.

3.2 The Virtual Node Emulator

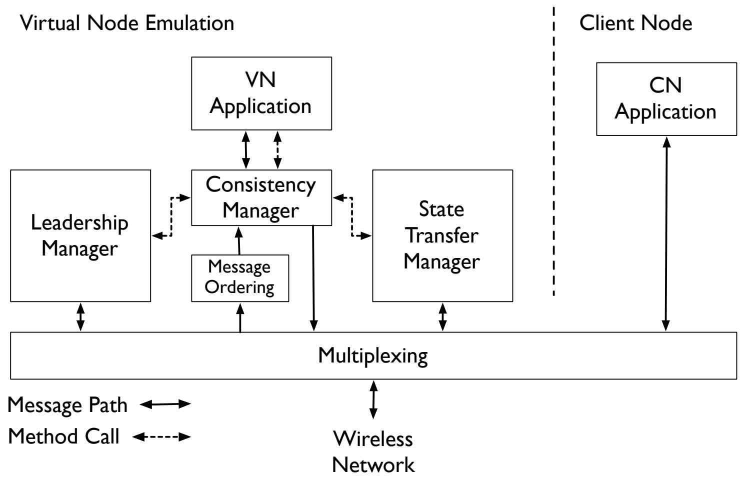


Figure 3-5: Simulator and VNE Architecture. The multiplexing and wireless network layers are provided by the simulator.

The Virtual Node Emulator (VNE) is the piece of software that runs on each physical node to emulate the RVN Layer.

The core services of the VNE are divided into easily swappable modules, as illustrated in Figure 3-5. The three core management services abstract most of the functionality necessary to emulate a virtual node: The Leadership Manager provides a protocol to determine which physical device is "leading," or speaking for, a particular virtual node. The State Transfer Manager synchronizes new devices with the current state of the virtual node, and the Consistency Manager coordinates the other two services while watching for communication anomalies and keeping the virtual node application synchronized with the other devices in the region.

3.2.1 Consistency Manager

The Consistency Manager [1] is responsible for coordinating the other system components and keeping the state of the virtual node consistent across mobile nodes.

Its job starts when the mobile node enters a new region. The Consistency Manager queries the Leadership Manager, described in Section 3.2.3, to determine whether or not its node is leading by invoking its `amLeading()` method. If the node turns out to be leading, it means that the virtual node had failed and lost

[1]Source code for the Consistency Manager is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/vne/services/consistency-shim.py`.

its state. In response, the Consistency Manager sets the application to an initial state and enters normal operation.

If the node is not leading, the next order of business is to acquire a current copy of the RVN state and bring the emulator into sync with the other nodes emulating the RVN. The actual state transfer process is triggered by invoking the `join()` method on the State Transfer Manager. The State Transfer Manager, described in Section 3.2.4, returns two pieces of information: the current application state and the timestamp *last_msg* from the last message processed by the leader.

However, the acquired state may be out of date by the time the Consistency Manager gets it. In order to handle this, the Consistency Manager plays every message timestamped after *last_msg* back to the application. At that point the application is up to date and the Consistency Manager can enter normal operation. Once the Consistency Manager has the current RVN state, it informs the Leadership Manager by setting the Leadership Manager's `has_state` field to true.

In the case that there is message loss, it is possible for leadership management to return improperly. In particular, it is possible for the Consistency Manager to believe that it is leading a freshly restarted virtual node when it should not be. This condition is detected by the Leadership Manager, which notifies the Consistency Manager by calling the method `SorryILied(`*`lying_service`*`)`. When this method is invoked on the Consistency Manager, the Consistency Manager resets its own internal state and re-runs the state transfer protocol.

During normal operation, the Manager accepts incoming messages from the network and passes them to the Message Ordering System, described in Section 3.2.2, via the `add()` method. Any messages broadcast by non-neighboring regions are discarded at this point. Then the Manager invokes the Ordering system's `deliver()` method to check for any messages ready to be delivered to the application.

Messages are passed to the RVN application by passing them to the `msgReceived(`*`msg`*`)` handler. The application returns a list of response messages. If the node is leading, the responses are broadcast. Otherwise, they are simply added to an array `tx_log`. In order to mitigate the impact of message loss, every node listens for messages broadcast by the leader, and if a node overhears a message that is not in `tx_log`, it assumes that it missed an incoming message and tells the State Transfer Manager to update the RVN state. Additionally, if a node becomes leader after the previous leader has left, it broadcasts all of the messages in `tx_log`.

It's important to understand that even though the Consistency Manager includes a variety of features that add robustness in the presence of message loss, it does not provide any guarantees if message loss occurs. In fact, if message loss is allowed, then it is possible for an RVN to transition into a wildly incorrect state: For example, if a following node f misses every message broadcast by the leader l and some of the messages broadcast by neighboring RVNs and l leaves the region or crashes, then f will have a fair shot at becoming the new leader. Once f becomes leader, it will broadcast everything in `tx_log`. When other nodes in the region receive these messages, they will automatically resynchronize with l into an incorrect state.

3.2.2 Message Ordering

Although this implementation of the VN layer makes some very strong assumptions about local broadcast integrity, there are some aspects of reality that it cannot completely ignore. In particular, the possibility that messages arrive out of order has to be considered by even the most trivial real-world deployment because messages arrive an order-of-magnitude sooner over local interfaces than over remote interfaces. This disparity means that nodes tend to receive local broadcasts before receiving remote broadcasts, even if the remote message was broadcast first.

In order to deal with this problem, the VNE uses a simple total-ordering protocol [2] inspired by Lamport logical clocks [22]. The ordering protocol seeks to guarantee that every node will process messages in the same order given that they are received within d seconds.

Every message is timestamped with a triplet $\langle time, node\_id, sequence\_num \rangle$. Whenever a new message is broadcast, the node sequence number is incremented by one, and the *time* field is set to the maximum of the system time and the maximum received time plus one. Incoming messages are sorted by timestamp and delivered after d seconds.

3.2.3 Leadership Manager

The leadership management service [3] determines whether or not the VNE is actually leading the virtual node.

The current leadership management algorithm implemented in the VNE is a simple protocol based on heartbeat messages and leadership requests. It does not make particular effort to ensure correctness in the presence of significant message loss, but is intended to work well enough for practical testing. The imple-

[2]Source code for Ordering Service is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/vne/ordering.py`.

[3]Source code for the Leadership Manager is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/code/vne/services/beat-leader.py`.

mentation attempts to fulfill the following properties under the assumption of no message loss:

1. There is precisely one leader in the region at a time.
2. If a node *n* becomes leader of a region, then either
   (a) it successfully received the VN state from a prior leader, or
   (b) there are no nodes in the region which possess the VN state.

When a node enters a new VN region, it waits a random period of time, *w*. If it receives either a heartbeat or another leadership request during this period, it will begin following. Otherwise it will request leadership by broadcasting a ⟨LEADREQ, *region*, *application*⟩ message. If another node receives the request and already believes that it is leading the region, then it immediately broadcasts a denial ⟨DENY, *region*, *application*, *lead_time*⟩. The field *lead_time* contains the time that the node originally requested leadership, as measured by the node's local clock. Additionally, if the requesting node overhears any other requests, it will also broadcast a denial. After waiting a sufficient period of time, the requesting node will consider itself leader and begin broadcasting periodic ⟨HEARTBEAT, *region*, *application*, *lead_time*⟩ messages.

If two nodes both attempt to request leadership and subsequently both transmit DENY messages, the node that requested leadership first will win. This decision is made based on the contents of the *lead_time* field and ties are broken based on node names.

If the leading node crashes or leaves the region, then other nodes will stop receiving heartbeat messages. If a following node does not hear a heartbeat for more than `HB_TIMEOUT = ALLOWED_LOSS*`

`HEARTBEAT_INTERVAL` seconds, then it will transmit a leadership request after a short, randomized delay of up to `MAX_DELAY` seconds.

In order to fulfill the second property, nodes that do not have the current RVN state are forced to wait longer to request leadership than nodes that do have RVN state. The actual process of obtaining RVN state in a timely manner is detailed in Section 3.2.4. These delays ensure that if there are nodes in the region that have the RVN state, then they will acquire leadership.

There are two groups of nodes that do not have RVN state but might attempt to acquire leadership: nodes that have just entered the RVN region and nodes that have already entered the region and sent a request for the state. To prevent the first group from incorrectly acquiring leadership, the initial request delay *w* is set to be greater than `HB_TIMEOUT + MAX_DELAY`. This delay means that if there are any nodes already in the region, they will attempt to claim leadership before a new node will. In order to prevent any node from the second group from acquiring leadership, they are forced to wait an additional `MAX_DELAY` seconds after detecting that the leader left. This delay is governed by the `has_state` field, which is controlled by the Consistency Manager.

3.2.4 State Transfer Manager

Unless a node's Consistency Manager determines that it is leading an RVN region, then it will need to acquire a copy of the current RVN state. The Consistency Manager does this by calling `join()` on the State Transfer Manager [4]. The current protocol is intended to be as simple as possible: A node broadcasts a join request message and waits for a response with the current application state.

When a node wants a copy of the current state, it broadcasts a request $\langle$JOIN, *region*, *application*$\rangle$. The message is tagged with the current virtual node region, and the name of the current virtual node application. If the State Transfer Manager of the leading node hears the request, then it responds with a message $\langle$JOINACK, *region*, *application*, *state*, *last_msg*$\rangle$. The response is tagged with the VN region and name, the application state, and the timestamp of the last message that has been processed by the leader. The representation of the application state is application dependent.

Once the node receives a valid join response for its region, the transfer protocol returns *state* and *last_msg* to the Consistency Manager.

3.2.5 Application Interface

An RVN application essentially describes a state machine that is driven by the Consistency Manager. The programming interface reflects this view. The interface that the RVN application has to conform to is focused on three things: allowing the Consistency Manager to explicitly control the RVN state, to pass information about the current region to the RVN application, and to pass messages to the RVN application.

__init__(self, vn_map) is the application constructor. The application is given a copy of the virtual node map. At initialization, the RVN region can be considered to be `None`.

getState(self) must return the entire state of the application encoded as a binary string.

loadState(self, state) must set the application state to the specified value. It must accept the same format generated by `getState()`.

reset(self) is invoked to return the application to an initial state.

[4]Source code for the State Transfer Manager is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/vne/services/join-shim.py`.

regionChanged(self, region) is invoked when the host physical node changes regions. The intention of this call is simply to allow the RVN to store the current region in a convenient format. This strategy allows the RVN program to be self-contained and not have to rely on calling external APIs.

msgReceived(self, msg) is invoked whenever a new message has arrived for the virtual node. The return value is a list of response messages.

3.2.6 Client Interface

Clients are provided access to relevant VN information via their constructor. Each client is passed the name of their mobile node, a map of virtual regions, and a locator object that provides an interface to the physical position and the current region of the node.

The VN map encapsulates information about the layout of VN regions on the field. It provides access to a list of regions and can perform point location to determine which region a point is in. The locator object provides platform-independent access to a coordinate location.

If a client needs direct access to simulator internals, for example, the current mobile node object, it can access them by directly importing the simulator namespace. This type of direct access to simulator internals is clearly outside of any abstraction, however this kind of simulator "magic" is frequently useful. For example, it allows a client to look at the internal state of its physical node for debugging and testing purposes.

Client applications are allowed by the VN abstraction to send and receive messages to virtual nodes at will. Moreover, they might involve complicated user interfaces or use other communication networks simultaneously with virtual broadcast, for example a "gateway" client might have an out-of-band connection to the Internet. In order to allow a wide range of clients to take advantage of virtual nodes, the client API needs to be application and programming style independent.

All client implementations need to inherit from the base client class `VNClient`, which provides accessor functions for location information. Beyond that, the only requirement imposed on clients is that their constructor accept certain parameters and that they implement a `run` method.

__init__(self, name, vn_map, locator) is the client constructor. Clients are given a copy of the virtual node map, and a reference to a node `Locator` object.

run(self) defines the execution thread for of the client program.

One key piece of information that each client has access to is the mobile node's current location. The

VNE infrastructure makes this available via special `Locator` object, which has two properties, `position` and `region`.

Chapter 4

Using the Simulator

The goal of this chapter is to illustrate how to use the simulator to perform experiments with ad hoc networks. Where previous chapters discussed the simulator architecture and network models, this chapter will focus on the simulator's user interface. It will explain how to write scenario files, configure the simulator, generate mobility traces, visualize data, and use simulator toolchain.

4.1 Visualization

There are two workflows that I frequently use while performing simulations. One strategy is to identify interesting information prior to running the simulation and then tracking these aggregate statistics while the simulation runs. Once the simulation is over, the collected information is written to logs that can be directly analyzed. This workflow has two primary strengths: it doesn't usually require significant storage space and post-simulation analysis is fast. Yet, it's impossible to look at anything after the simulation except for the data that was explicitly recorded. Although this limitation is always a nuisance, it isn't necessarily critical if simulation is fast enough.

The other strategy is to record exhaustive information during the simulation and then use lots of custom scripts to extract data from these log files. This strategy is particularly useful when simulation is significantly slower than extraction, which turns out to be the case fairly often when performing RVN simulations with Python. However, it can be time consuming to write extraction scripts even with a good language and framework. Furthermore, it's easy to generate hundreds of megabytes, or even gigabytes, of message logs and traffic data in the normal course of running these simulations.

Both of these workflows have particular strengths. Yet, both of them make it very difficult to see the big

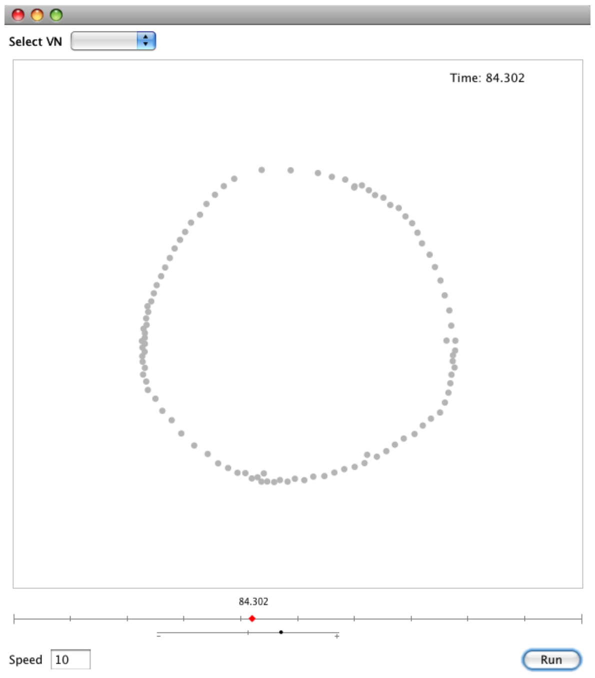


Figure 4-1: This visualization is showing nodes as they execute a simple circle-forming algorithm.

picture. Because the researcher is limited to looking at a fixed set of aggregate statistics in the first case and has to write custom scripts to extract information in the second, it's very easy to suffer from tunnel vision.

One good strategy for getting a broader view of a simulation is to use a graphical view. Visualizations have two distinct advantages over simple log files: They can very concisely represent a huge amount of information and they don't necessarily filter that information based on the user's preconceived notions. The latter trait is particularly important for debugging purposes. For example, if a handful of nodes "crashed" during the simulation due to a rare bug, it might easily go unnoticed by looking only at numeric results. However, if node status is represented visually, then the problem is trivial to observe.

These kinds of network visualization tools have been used before. For example, ns-2 comes with the NAM [23] network animation tool for visualizing ad hoc network simulations. The Python simulation framework developed in this thesis includes a custom visualizer written in Java and designed to show RVN and physical node states.

4.1.1 Implementation

The visualizer (the Viz) is designed to operate on recorded traces of the simulator execution.

Internally, the Viz maintains a set of objects representing the state of physical and virtual nodes. When

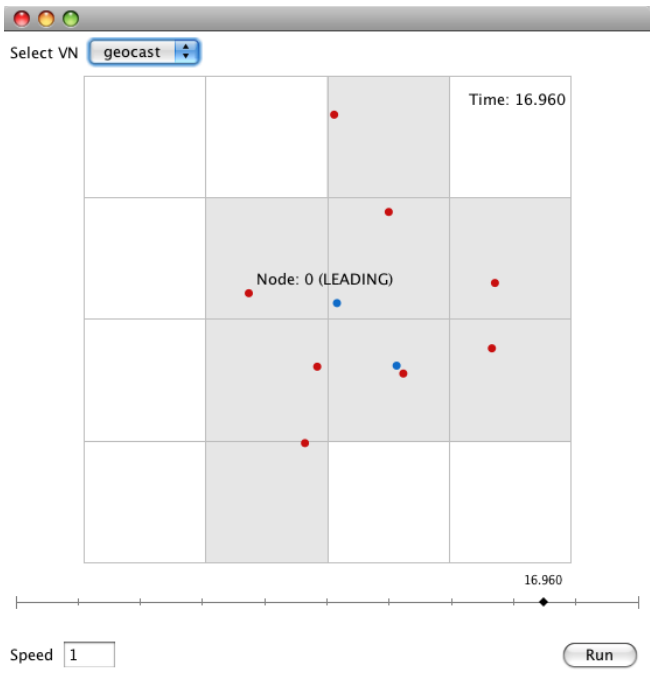


Figure 4-2: The visualization shows physical and virtual node state. Leading nodes are highlighted in blue, and active virtual nodes are shown in gray.

the log file is parsed, state changes are reified as `Event` objects that are inserted, in order, into a vector. During visualization, the Viz simply plays the event vector and updates the visualization after each update. All `Event` objects contain an `undo` method, which allows the Viz to be played in reverse or to seek to any position.

Information about node movement is also represented internally as a series of `Events`. The simulator is capable of embedding node position data into the log file, and the Viz can read these positions. However, this can lead to excessively large logs when used in long-running simulations. In order to deal with this, the Viz supports reading node mobility information directly from the same ns-2 traces that were used to run the simulation.

the Viz includes an embedded Python interpreter that accepts connections over a local socket. Although the current version of the viz doesn't implement a substantial amount of functionality in Python, future versions will allow the user use Python to script different visualization layers interactively.

4.1.2 Usage

the Viz is intended to be easy to use. To start, a user opens a log file generated by the simulator and optionally specifies a mobility trace. Once the log has been loaded, mobile node locations, states, and virtual node regions are plotted on the screen, as shown in Figure 4.1.2.

In the default configuration, mobile nodes are colorized according to their state:

Light Red nodes are in the process of requesting leadership of an RVN.

Dark Red nodes are currently leading an RVN.

Light Blue nodes are in the process of joining an RVN.

Dark Blue nodes have successfully joined an RVN but are not leading.

Gray nodes are not part of an RVN and have not started joining one.

The visualization also displays the state of every RVN:

Gray regions represent active RVNs. An active RVN has a node acting as leader and participating in RVN emulation.

White regions represent crashed RVNs. A crashed RVN is not capable of responding to messages or performing computation.

Although the visualization is oriented toward RVN simulations, it works for others as well. Figure 4.1.1 shows a visualization of a simple motion coordination algorithm.

4.2 Tutorial: Examining Virtual Node Overhead

One of the best ways to get a feel for a new programming environment is to look at an example. This chapter walks through the steps involved in running a basic simulation using the RVN Layer. Perhaps the simplest application of the simulator is to evaluate the message overhead associated with maintaining an RVN. Because the overhead is largely independent of the particular RVN application, measuring it doesn't require writing very much code. The experiment will set up a grid of RVNs with an empty RVN application and measure the number of physical node messages.

Assembling an RVN experiment involves roughly three steps:

- Write the RVN and client applications.
- Write a script to drive the simulation.

- Configure the details of the simulation and generate a mobility trace.

None of these jobs depend on each other, but this section starts by looking at the scenario file in order to paint a picture of how the pieces fit together.

4.2.1 Specifying a Scenario

Scenario files are Python scripts that drive the network simulator. The script gains access to the simulator by importing the appropriate modules. The core simulator tools are contained in `sim` and RVN specific methods are contained in `vnesim`. The simulator network stack is in `simsockets`.

A typical scenario has a standard structure: import the simulator modules, parse a configuration file into a Python dictionary, and then pass the configuration to the simulator. The scenario for this example, `scenario.py`, is shown in Listing 4.1. It runs a simulation with the same mobility trace 20 times, incrementing the number of participating nodes by two between each run. Each run is written to a different log file named after the number of nodes in the simulation.

Code Listing 4.1 (A Simulator Scenario File `scenario.py`).

```
1  import sim, vnesim
2
3  config = sim.parseConfig("vnodes.cfg")
4
5  for i in range(2, 41, 2): # 2, 4, 6, ..., 40
6      print "Run %d . . ." % i
7      sim.logging.file_name = "log-%d" % i
8      config['mobile nodes']['number'] = i
9
10     vnesim.go(config, 60) # Simulate for 60 seconds
11     sim.reset() # Reset for the next run
```

On line 1, the core simulator and RVN modules are imported. Then, on line 3, the simulator configuration is parsed. Although it is considered good practice to keep simulation parameters in a separate file, it isn't strictly required. Once parsed, the configuration is simply a nested series of dictionaries and lists.

The loop on line 5 varies the number of physical nodes in each simulation run. Line 7 explicitly sets the name of the log file used by the simulator. This statement works because the module level name `sim.logging` is aliased to the `logging` property on the default simulator instance. Line 8 uses the configuration dictionary to change the number of physical nodes in the simulation while leaving other parameters unchanged. The function `go`, invoked on line 8, runs an RVN simulation using the given configuration for 60 seconds.

This function is provided by the `vnesim` module and automatically creates the necessary `MobileNode` and RVN objects. The simulator is reset for the next run on line 11.

4.2.2 Configuring the Simulation

In order to avoid cluttering the Python code with simulation specific details, most of the details should be stored in an external file. The native format used by the simulator is YAML [24], a human-readable markup language. When a simulator call explicitly asks for the name of a configuration file, the file must be in YAML. However, most calls only depend on the parsed dictionary structure, so the user is ultimately free to choose his or her own format or to configure things inline.

Nonetheless, this example uses YAML. There are two principles that should make the YAML syntax clear: Items followed by colons are keys in a dictionary and items preceded by a hyphen are entries in a list. Numeric values are automatically parsed as Python numbers.

An RVN simulation coordinates several different systems: the simulator itself, the various physical nodes, the clients, the virtual nodes, and the VNE. Each of these modules has its own set configuration options. The combination of Listing 4.2, 4.3, 4.4, and 4.5 makes up the complete configuration file `vnodes.cfg`.

sim includes core simulator options, such as the name of log files and seed values for the random number generator. This section also includes configuration options for the physical model, for example, the maximum transmission radius and the transmission delay.

Code Listing 4.2 (Simulator Configuration).

```
sim:
  medium:
    recv lag:
      bcast: 0.02 # Remote receive lag (s)
      local: 0.001 # Local receive lag (s)
    filter:
      max_r: 150 # Maximum transmission distance (m)
```

mobile nodes configures the number of physical nodes in the simulation. This section is where the mobility trace and client applications are specified. The `client` option is specified as `<path/to/file>.<class name>`.

Code Listing 4.3 (Mobile Node Configuration).

```
mobile nodes:
  number: 0 # Number of mobile nodes
  client: # Path to client app
  paths: paths # Mobility trace file name
```

virtual nodes sets up the RVN Layer. Configuration is performed in two parts. First, all of the RVN applications in a simulation are listed. Then, the various RVN regions and corresponding RVN apps are listed. The region specifier is used to describe the geometric shape of the RVN region. The simulator supports a variety of shapes. This example uses the `grid` type to automatically create a four-by-four grid of RVNs that stretches from $(0, 200)$ to $(200, 200)$. Grid squares are automatically named appropriately.

Code Listing 4.4 (Virtual Node Configuration)**.**

```
virtual nodes:
  applications:
    - name: empty # Name of RVN app.
      location: empty_app.EmptyApp # Path to RVN app
  regions:
    - spec: grid [(0, 200), (200, 200), (4, 4)] # Region specifier
      neighbors: [] # List of neighboring regions
      name: # Region name
      apps: empty # List of RVNs in region
```

vne details of the RVN emulation algorithms can also be specified. Each VNE service (leadership management, consistency management, and state transfer) must be specified in this section.

Code Listing 4.5 (VNE Configuration)**.**

```
vne:
  services:
    - location: beat-leader.LeaderElect
      beat_period: 2
      max_delay: 2
      beat_miss_limit: 2
      claim_period: 2
    - location: join-shim.Join
      timeout: 10
    - location: consistency-shim.ConsistencyMgr
      ordering_delay: 0.01
```

4.2.3 Writing the RVN Application

One of the reasons for choosing this example is that it doesn't require a functional RVN application. Because RVN overhead is essentially independent of the actual application, an empty application serves the purpose as well as any other. The minimal application, call it `empty_app.py` is shown in Listing 4.6. Most of the methods are empty. However, `loadState` must return a string, and `msgReceived` must return a list.

Code Listing 4.6 (`empty_app.py`, the minimal RVN application)**.**

```
class EmptyApp:
    def __init__(self, region, vnmap, state=None, mnode=None):
        pass

    def getState(self):
        return ""

    def loadState(self, state):
        pass

    def reset(self):
        pass

    def regionChanged(self, region):
        pass

    def msgReceived(self, msg):
        return []
```

4.2.4 Running the Simulation

Running the simulation is simply a matter of running the scenario script. the simulator includes a Python wrapper, `sim.sh`, to ensure that module include paths are set correctly.

Code Listing 4.7 (Running the simulation)**.**

```
sim.sh scenario.py
```

This process creates 20 files named `log-2` through `log-40` that contain detailed traces of their respective simulation runs. In each log, every message transmission is recorded as a "sim <tab> tx" line in the log file. This format is intended to be easy to parse using standard text manipulation tools. For example, in order to measure the total number of broadcast messages in each simulation run, `grep` is sufficient:

Code Listing 4.8 (Counting messages with `grep`).

```
for i in ‘ls log*|sort +0.4 -n‘; do
  grep ’sim tx’ \$i | wc -l >> values;
done;
```

This chapter is intended to give the reader a taste of the simulator’s user interface. Detailed results from a much more thorough version of this experiment can be found in Section 5.1.5.

Chapter 5

Point to Point Routing in Ad-Hoc Networks

In this chapter, I examine how to use the RVN Layer to route messages in a multihop ad hoc network. I compare a simple RVN-based routing scheme to Ad hoc On-Demand Distance Vector (AODV) [13, 25] routing and a greedy geographic routing algorithm.

5.1 Routing with the RVN Layer

The routing services used to test the VNE are based on the algorithms developed in [3]: a virtual node to virtual node broadcast service (geocast), a client location tracking service, and a client-to-client routing service. Each of these three layers uses the others to provide a cohesive service. The client tracking service uses geocast to store client positions at a set of virtual nodes, and the client-to-client routing service uses the client tracking service to determine where to find a client.

5.1.1 Geocast

The Geocast [1] service sends messages from one virtual node to another. The service is based on greedy DFS and is illustrated in Figure 5-1. If an RVN receives a message destined for a different RVN, the service forwards the message to the neighboring RVN that has the shortest Euclidean distance between its center and the center of the destination region. If it does not receive an acknowledgement within `RETRY_TIME` seconds, it will re-send the message via the second shortest path. This procedure repeats itself until it has attempted `MAX_TRYS` times, at which point delivery fails. `MAX_TRYS` controls the total number of transmission attempts.

[1]Source code for the Geocast Service is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/apps/geocast/vtov_app.py`.

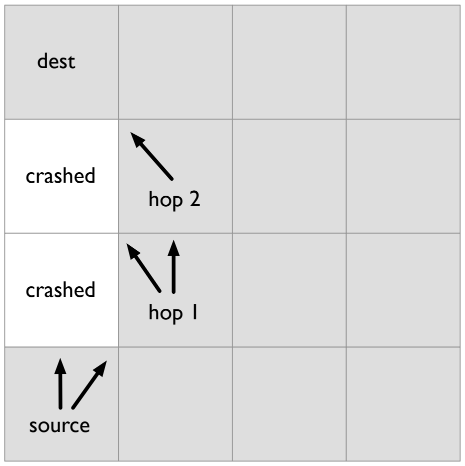


Figure 5-1: Routing with virtual nodes

Thus, for a grid of regions, it can range between one and eight: one will only attempt to transmit along the shortest path and eight will attempt to transmit through every neighboring region.

Once a message reaches the destination RVN, the destination broadcasts an acknowledgement message, FOUND, which is rebroadcast by every RVN that forwarded the original message. This scheme causes the acknowledgement to propagate back to the original RVN.

In order to avoid routing loops, the Geocast service maintains a table of every message that it has broadcast. When the Geocast service receives a message that it has already forwarded, it simply ignores the message. Old messages are removed from the table every `MSG_TIMEOUT` seconds.

`RETRY_TIME` is calculated as a function of the length of the shortest path between two virtual nodes: $2 * \texttt{HOP\_DELAY} * (region\_hops(src, dst) + 1)$. This approximates the time necessary to send a message to the destination and receive an acknowledgement. The added value 1 was chosen to add a fudge factor of one hop for both the forward route and the acknowledgement. For the purposes of these experiments, `MAX_TRYS` was set to three because it corresponds to one quadrant (for example, N, NE, and E) on a grid square.

5.1.2 Client Location Tracking

The Client Location Tracking Service [2] allows a client or RVN to query the recent location of another client. All clients periodically use the geocast service to send client locations to specific `NUM_HOMES` "home" virtual nodes whose names are calculated by hashing the client name into the domain of RVN identifiers. When

[2]Source code for the Client Location Tracking Service is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/apps/geocast/hls_app.py`.

another node wants to determine where a client is, it geocasts a query to the client's home locations, which geocast back a response.

The hashing code used by the Client Tracking Service is shown in Listing 5.1.

Code Listing 5.1.

```
rand = random.Random()
rand.seed(HASH_SEED)
homes = []
for i in xrange(PERM_COUNT):
  homes.append(shuffle(list(vnmap.regions), rand)[:NUM_HOMES])

def hash_client(self, client, i):
  try:
    return homes[int(client) % len(homes)][i % p_f]
  except ValueError:
    return homes[hash(client) % len(homes)][i % p_f]
```

The first five lines create an array `homes` of randomly generated permutations of RVN regions. Each list is truncated to the total number of home locations that each client will have. Then, `hash_client` simply indexes into the array using either the numerical value of the node, or Python's built in `hash` function.

5.1.3 Client-to-Client Routing

The Client-to-Client Routing Service [3] operates by combining Client Location and Geocast. When a client wants to send a message, it tells the local RVN. The RVN uses the Client Location Service to discover the destination client's current region. The RVN then geocasts the message to the destination region. Finally, the destination RVN delivers the message to the destination client. In order to maintain efficiency, the Routing Service caches results from the Client Location service for `ROUTE_CACHE` seconds.

These applications are well suited to evaluating the RVN layer. They are not trivial, but nor are they excessively complicated. They make good use of the layer's assumptions and features. For example, the geocast routing algorithm takes advantage of the fact that the geography of the virtual nodes is known and fixed. These applications also open the door to examining the performance of composite services with respect to different virtual node parameters.

5.1.4 Implementation Notes

The RVN routing scheme is implemented in the simulator as three different RVN Layers. This decision, which was made due to the structure of the simulator, means that each part of the routing service poten-

[3]Source code for the Client-to-Client Routing Service is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/apps/geocast/route_app.py`.

tially has different leaders and fails at different times. Although this design diverges from [3], it does not substantially change the performance of the algorithms: the same physical nodes are running the same VN applications.

One important implementation detail involves the translation of the routing algorithms in [3] from the VSA model to the RVN model. The most important difference between these models is that a VSA is a timed automaton: a VSA's state can evolve independently of message reception. Although this difference is less relevant in a routing algorithm – most actions are clearly driven by receiving a message – some activities like route expiry and route timeouts require this behavior. In order to approximate this ability, the implementation introduces a special timeout client that runs on all nodes. The timeout client periodically broadcasts messages of the form ⟨TIMEOUT, *time*⟩. RVNs trigger any time-sensitive activities based on these messages. Additionally, any RVN that requires an approximate clock can use the maximum timestamp from all received TIMEOUT messages.

5.1.5 Virtual Node Overhead and Performance

The overhead imposed by the RVN Layer is largely independent of the particular application. The message overhead imposed by the RVN layer is dominated by the component services of the VNE: leadership management and state transfer. Although some applications maintain more state than others, which leads to larger state transfer messages, message overhead is primarily dependent on the mobility pattern followed by physical nodes. The leadership management protocol incurs overhead in two ways: it periodically broadcasts heartbeat messages and a flurry of messages is involved in electing a new leader. Heartbeat messages are essentially a constant overhead per active RVN, however new leader elections happen more often in high mobility situations. The join protocol incurs overhead whenever a node joins a region, which is directly associated with how much nodes are moving around

The RVN layer also imposes overhead in terms of failure and performance penalties. RVNs can crash – and suffer state loss – if every node in the region leaves or crashes. Additionally, an RVN can temporarily pause in the case of a leadership change. These issues are also independent of the particular RVN and client applications and depend on node mobility.

In order to measure the overhead generated by the RVN layer, I ran a series of tests using a five region by five region, 400 m by 400 m grid. The test was run at three node densities: 25, 50, and 100 nodes. These levels correspond to one, two, and four average mobile nodes per RVN region. Node mobility was simulated using the random waypoint model. The physical broadcast radius was set to 250 m and the broadcast delay was set to 0.02 seconds. Mobility traces were generated using the setdest tool distributed with ns-2. For each density level, the waypoint pause time was varied from 100 to 900 seconds. Node velocity was evenly distributed between zero and 20 m/s for all experiments. Each test was run for 1000 seconds using five different, randomly generated mobility patterns and the mean of results was taken.

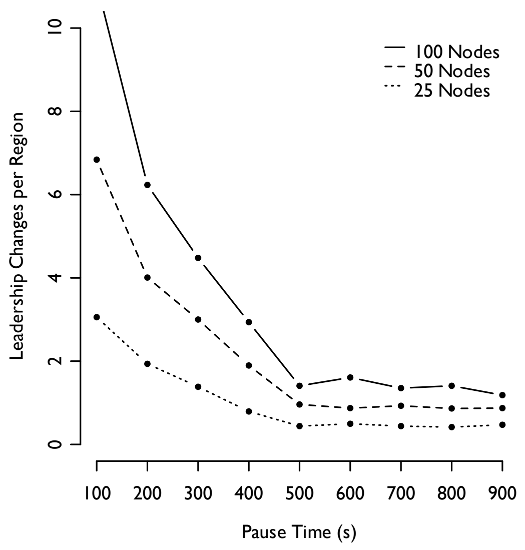
The tests examined both message overhead and failure rate. Message overhead was measured by counting the number of messages of various types that were transmitted per region. Failure statistics measure how often regions lost their state, how often they changed leaders, but retained state, and the percentage of time that the region is *stable*. A stable region is a region that is working normally, that is, it has a leader with state.

The RVN failure overhead results are summarized in Figure 5-2. Figure 5-2c summarizes the story: it's quite clear that the percent of time that an RVN spends capable of performing useful work is essentially independent of pause time. This means that even in high mobility situations, it's unlikely for an RVN to completely lose state. However, the stable time is strongly dependent on node density. This observation suggests that RVNs can work well, as long as there are enough physical nodes to effectively replicate the state.

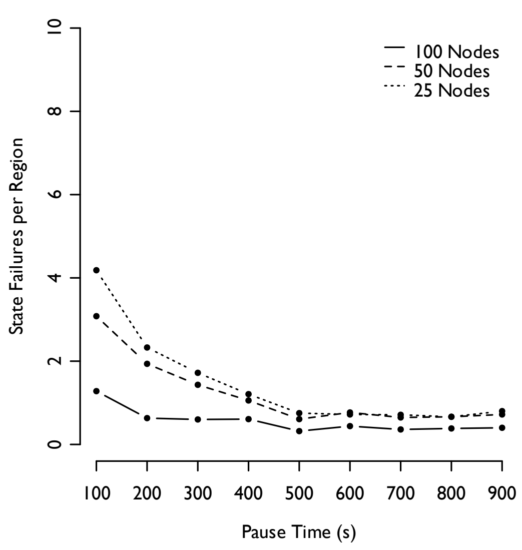
These plots have several interesting features. Figure 5-2a shows that the number of leadership changes per region increases with node density and Figure 5-2b shows the opposite: the number of state failures declines as node density increases. The fact that the number of state failures declines with increasing node density should be intuitive: If there are more physical nodes participating, then there is more redundancy and a lower chance that a region will ever be left completely empty. The fact that there are more regions that are continually occupied drives the corresponding increase in the total number of leadership changes. The net effect is shown in Figure 5-2c: As the node density increases, the amount of time that the RVN Layer is stable increases.

One counter-intuitive result in these plots is that the stable time, shown in Figure 5-2c, seems to be independent from the pause time. However, the 5-2d tells the real story: As the pause time increases, some RVNs become much more stable. However, because the physical nodes don't move as much, other RVNs spend the entire simulation in a failed state.

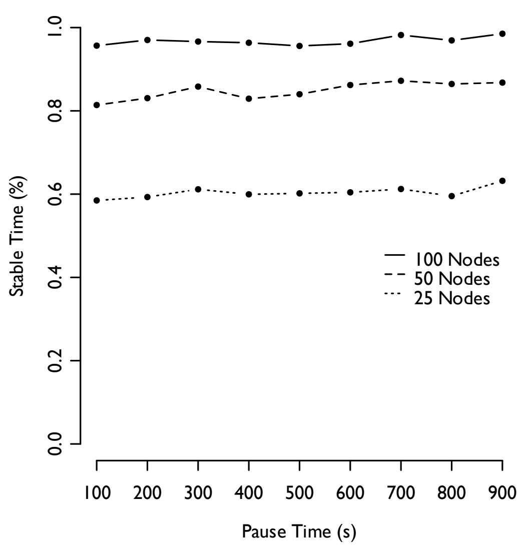
Additionally, there is a clear kink in Figures 5-2a and 5-2b at 500 seconds: in both graphs, the respective failure rate hits a low point and levels off. This kink is an artifact of the way that setdest generates mobility traces. Unlike some tools, setdest will wait for the entire pause time before starting node movement. Be-



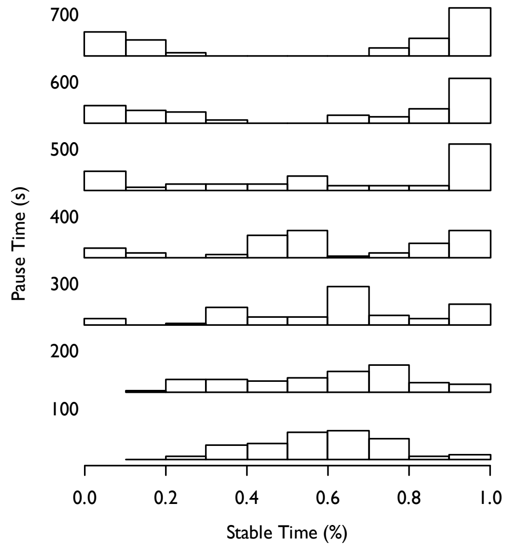
(a) RVN Leadership Changes



(b) RVN State Failures



(c) RVN Stable Time



(d) Histograms of Stable Time vs. Pause Time for 25 nodes

Figure 5-2: RVN Failure Overhead. These plots show the mean results from five simulator runs using independent mobility traces. Node velocity was evenly distributed between 0 and 20 m/s, and each test was run for 1000 seconds. Figure 5-2c shows the average percentage of time that regions were stable at various node densities. Figure 5-2d shows the corresponding frequency distribution for 25 nodes.

cause these simulations were run for 1000 seconds, this means that the nodes in any run with a pause time of 500 seconds or more change positions only once.

The RVN message overhead is illustrated in Figure 5-3. As should be expected, the heartbeat messages that are broadcast by the leadership management service are the most numerous. The number of heartbeat messages is strongly correlated to the node density: The higher the node density, the more stable the RVNs, which means that a leader is more likely to be broadcasting a heartbeat. The number of heartbeat messages is also negatively correlated to the pause time for the same reason: RVNs are slightly more stable in low mobility conditions.

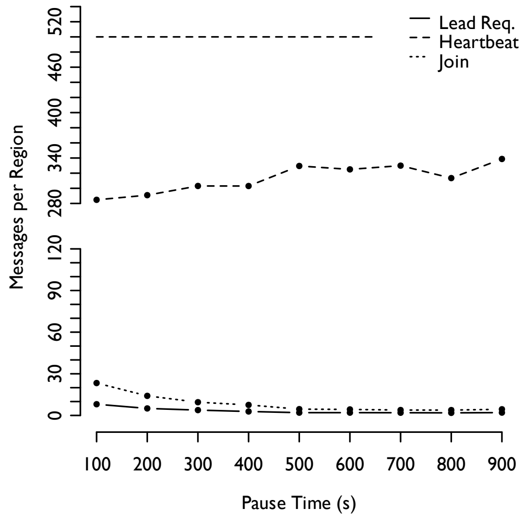
Similar observations apply to leadership requests and state request (join) messages. They scale almost linearly with node density because there are more region transitions, and they fall off rapidly as mobility decreases for the opposite reason. Note that join requests are much more common than leadership requests in high mobility and high density settings. This difference is caused because every single node that joins a region needs the region state, but the vast majority will not have to request leadership.

5.1.6 Routing Performance

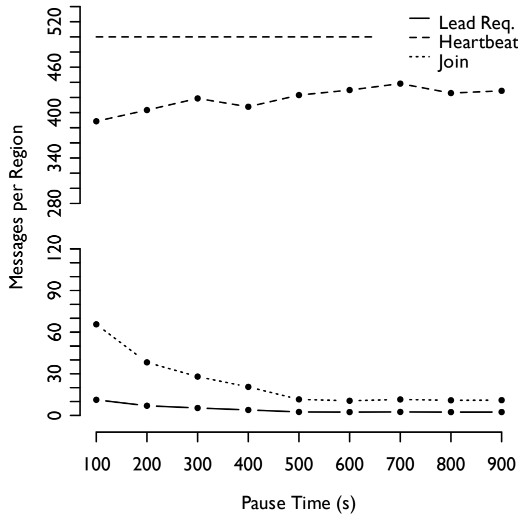
To test the performance of the RVN Geocast protocol, the following experiment was run: Ten virtual nodes were chosen as senders and ten as receivers. Each sending region attempted to geocast a message to the corresponding receiving region once every five seconds. If either the sending or the receiving region crashed during the simulation, a new region pair was chosen. The simulation was run for 1000 seconds with the same same mobility patterns, grid, and node parameters as in the RVN overhead experiments. Unlike the overhead experiments, these figures include the results of only a single run at each node density/mobility level.

The experiment results are summarized in Figure 5-4. The results focus on two metrics: packet delivery fraction and average message latency. The packet delivery fraction is the percentage of packets that were successfully routed from the source to the destination. The average latency reflects the average amount of time that it took to deliver the messages.

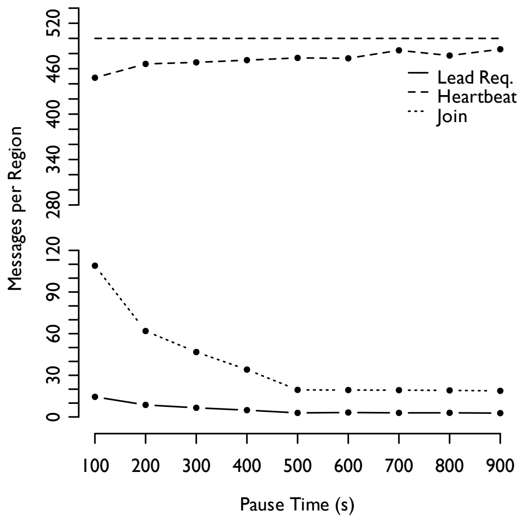
At low node density, the packet delivery fraction, shown in Figure 5-4a, was low and hovered around 70 percent. Additionally, at low node density the delivery fraction seems to be wildly unstable – jumping from 20 percent at 700 seconds to 80 percent at 800 seconds. The apparent instability is actually a high sensitivity to the initial node distribution. As shown in Figure 5-2d, as mobility decreases, there will be a relatively small number of highly stable virtual nodes at the expense of a number of completely inactive nodes. The



(a) 25 Nodes



(b) 50 Nodes



(c) 100 Nodes

Figure 5-3: RVN message overhead as a function of node density. For each pause time, these plots show the mean values from five simulator runs using independent mobility traces. Node velocity was evenly distributed between 0 and 20 m/s, and each test was run for 1000 seconds. The horizontal dotted line at 500 messages represents the maximum potential number of heartbeat messages per region.

ability to route between randomly chosen locations is strongly dependent on the particular location of active nodes. The effect is much less pronounced with high mobility: Although regions fail more often, the ability to route messages is much less dependent on initial conditions.

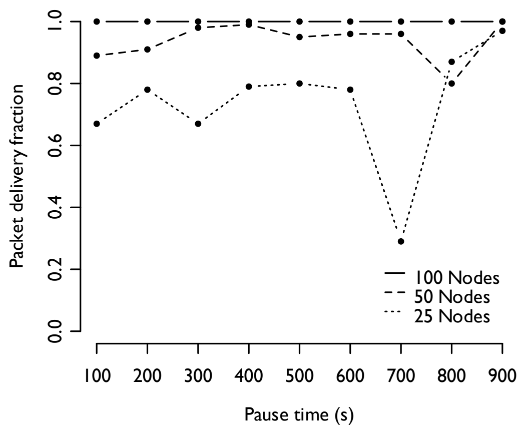
This sensitivity shows up in the average latency as well. In Figure 5-4b, there is a huge spike at 700 seconds. This spike is primarily driven by a handful of regions that were just barely connected: Messages eventually arrived but only after taking a convoluted route through much of the network. A similar, but much less pronounced problem causes the spike at 100 seconds in Figure 5-4c. Arguably, these messages should have timed out, which would improve the average latency figures at a small cost to the delivery fraction.

At moderate and high node density, the RVN Layer is much better behaved: When the Layer is essentially stable, it tends to take slightly over 0.1 seconds to route between nodes. This number corresponds to an average of three hops. One of the problems with using the RVN Layer for routing is that it pays, at the very least, a factor of two latency penalty because the message ordering system. Thus, when the transmission delay is set to 0.02 seconds, communication between RVNs takes at least 0.04 seconds.

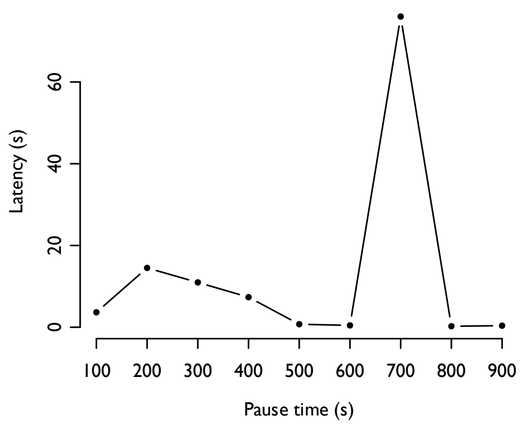
Measuring the performance of the Client-to-Client Routing Service was more difficult. One of the most problematic limitations of the Python simulator is its performance. Because simulating the Client-to-Client Service is several times more involved than simulating Geocast by itself, the measurements are necessarily limited. This experiment randomly chooses ten pairs of nodes. One node is designated the sender and the other node the receiver. The simulation waited 20 seconds for the RVN and Client Location Service to stabilize. Then, the sending nodes transmitted a message via the Client-to-Client Service twice a second. Measurements were taken for 15 seconds. This procedure was repeated for five node configurations at 25 and 50 total physical nodes. The intention of this experiment is to understand the steady state characteristics of the routing service at different node densities.

The results from the experiment are summarized in Figure 5-5. Much of the story from the Geocast analysis is is clearly visible in these plots. At low node density, most messages are delivered promptly, but many nodes have essentially zero connectivity. It's important to remember that the simulation time in this experiment is short enough that it filters the extremely high latency messages that were visible in the Geocast experiment. With a higher node density, nodes have much better connectivity and most messages are delivered.

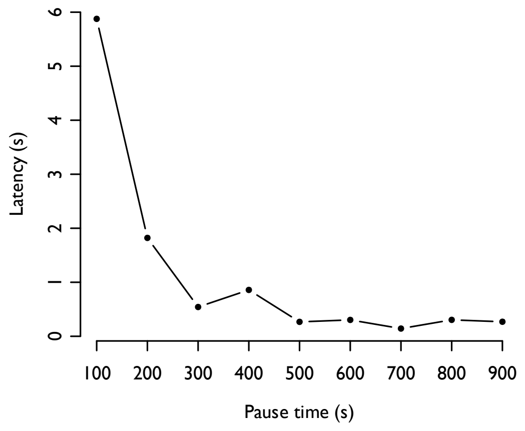
Notice that the average client-to-client latency is noticeably larger than the Geocast latency. This is primarily due to the cost of performing the client location prior to sending.



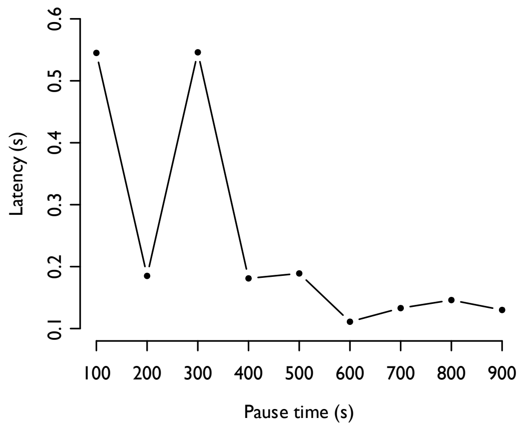
(a) Delivery Fraction



(b) Latency – 25 Nodes

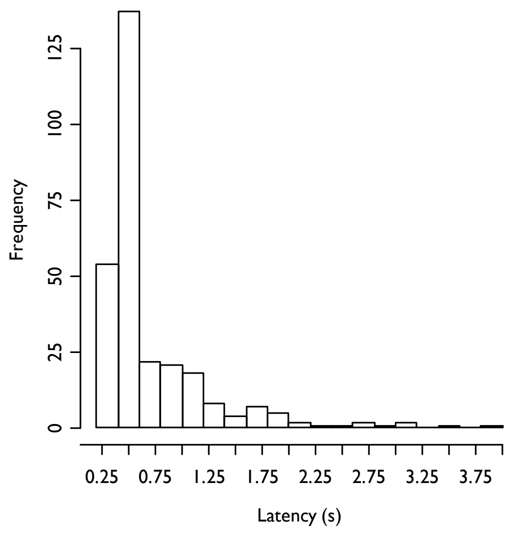


(c) Latency – 50 Nodes

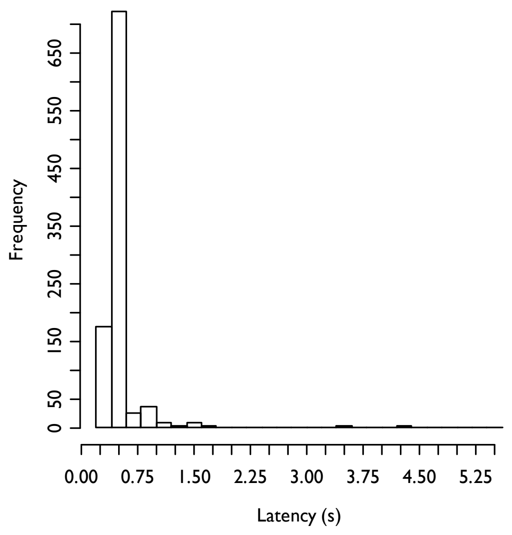


(d) Latency – 100 Nodes

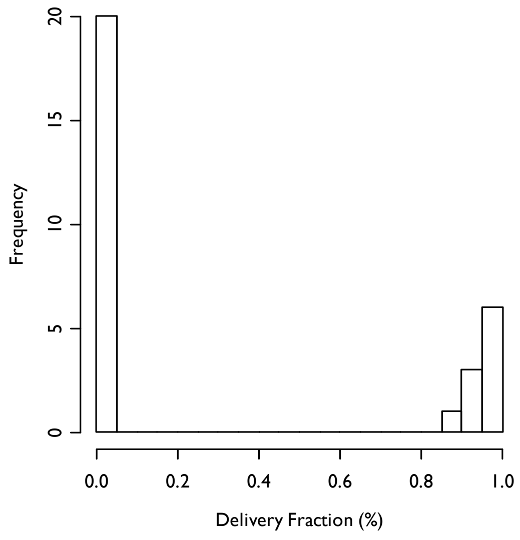
Figure 5-4: RVN Geocast Routing Performance. Each pause time corresponds to a single 1000 second, simulation run using a five by five region, 400 m by 400 m grid. Node velocity was distributed between 0 and 20 m/s. Broadcast radius was set to 250 m.



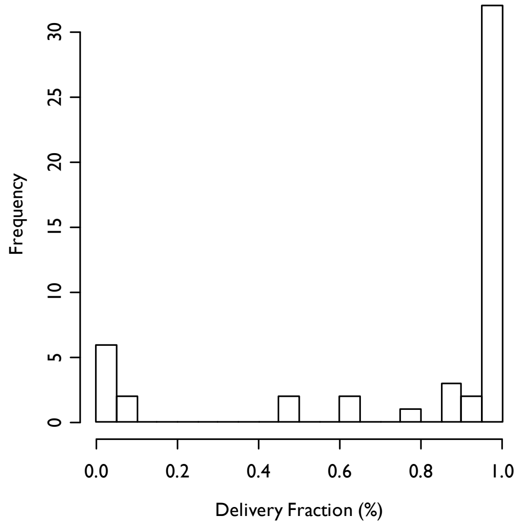
(a) Latency – 25 nodes



(b) Latency – 50 Nodes



(c) Delivery Fraction – 25 Nodes



(d) Delivery Fraction – 50 Nodes

Figure 5-5: RVN Client-to-Client Routing Performance. Each set of plots was generated from five, 15 second, simulation runs using a five by five region, 400 m by 400 m grid. Node velocity was distributed between 0 and 20 m/s. Broadcast radius was set to 250 m.

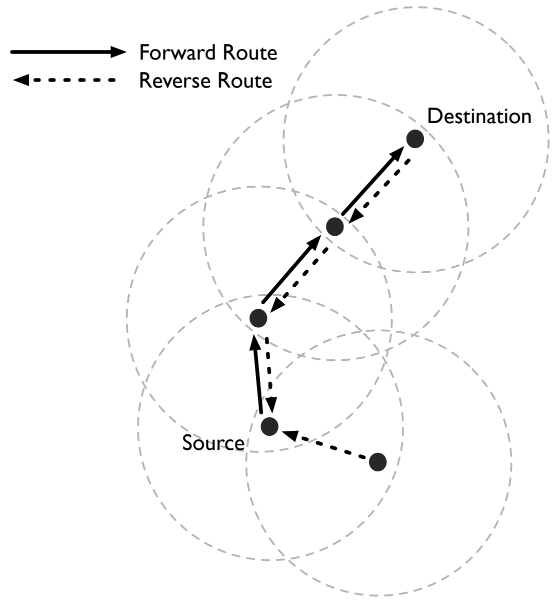


Figure 5-6: The Ad hoc On-Demand Distance Vector routing protocol

5.2 Ad hoc On-Demand Distance Vector

In order to contextualize the performance results from the RVN Geocast and Client-to-Client Routing Service, it's necessary to compare them to another routing protocol. This section considers the Ad hoc On-Demand Distance Vector (AODV) routing protocol [13, 25]. AODV is a well-known and respected routing protocol for MANETs. Unlike the other routing protocols examined in this chapter, AODV does not use any form of geographic information. It is an on-demand protocol, which means that it does not actively maintain its routing tables. Rather, a node must run a special route discovery algorithm when it needs to send a message.

The route discovery process, illustrated in Figure 5-6, uses a form of controlled flooding. It begins when a source node broadcasts a RREQ message. Every node that receives the request performs two tasks: it creates a reverse route to the source of the request, and it re-broadcasts the route request. When the request eventually reaches either the destination node or a node with a current route to the destination, a RREP message is sent back to requesting node via the established reverse route. This process of flooding the network in order to find a route is potentially costly. In order to control the process, AODV uses an expanding ring search. That is, it initially sends the RREQ only to its neighbors. Then it floods the RREQ to every node within two hops, and so on until it reaches the edge of the network.

One of the most important parts of a MANET routing protocol is route maintenance. How does the protocol discover when a route is broken, and what does it do to fix it? AODV is able to take advantage of a variety of different strategies for detecting broken links. The AODV standards document, RFC 3561 [25], specifies three that can be used individually or in combination: HELLO messages, link-layer feedback, and passive acknowledgements. HELLO messages are broadcast periodically by a node to inform neighboring nodes of its presence. Link-layer feedback takes advantage of the packet acknowledgements used by some MAC protocols. "Passive acknowledgement" involves observing the channel to see if a neighboring node forwards route requests when it should.

When a node running AODV discovers that one of its routes has failed, it sends an RERR message to every neighboring node that was using the failed route.

AODV tags route replies with a destination sequence number. The destination sequence number is generated by the route destination and included in all route replies. Every node increments its sequence number before originating a route discovery and before responding to an RREQ. The sequence number makes it extremely clear which routes are more recent than others and allows AODV to avoid routing loops.

5.2.1 Implementation Notes

In order to obtain results that are comparable to the RVN system, I implemented AODV [4] using the same Python simulator framework. The AODV implementation is based on RFC 3561 [25]. Although my implementation is mostly compliant with the RFC, it leaves out some features that are not relevant to these tests.

The RFC includes provisions for operating AODV over unidirectional links. The RFC also allows a node to attempt to repair a broken route locally. In this case, the node originates a new route request for the broken route rather than propagating a route error. Neither of these optimizations are included in my implementation. Additionally, it does not use link layer feedback or passive acknowledgements to detect link failure. Rather, it relies entirely on HELLO messages. This limitation does hurt performance in high mobility situations. However, it's not possible to implement properly because the simulator doesn't implement link layer acknowledgements.

My AODV implementation maintains the mapping between hardware addresses and IP addresses internally. It does not use a separate ARP protocol. This is a relatively minor change, considering that AODV necessarily receives a message from any node that it wants to unicast to, due to the way route discovery works. This change generally improves performance, because it avoids potential ARP resolution latency if the ARP cache falls out of the sync with the AODV route cache.

[4] Source code for my AODV implementation is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/apps/aodv/aodv.py`.

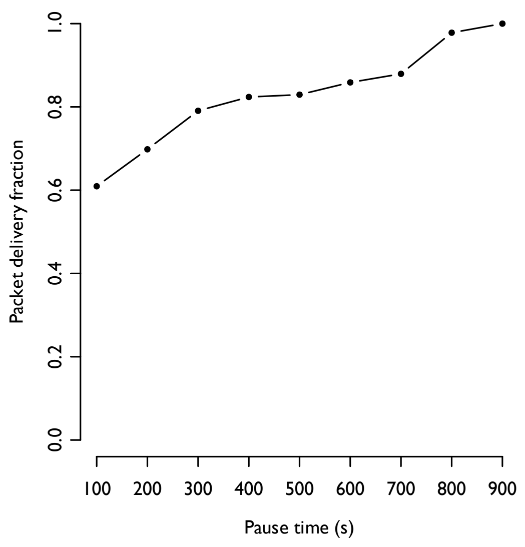
All message timeouts and broadcast rates are set to the default values specified in the RFC. In particular, HELLO messages are broadcast once a second, and route lifetimes are initialized to three seconds.

5.2.2 Performance

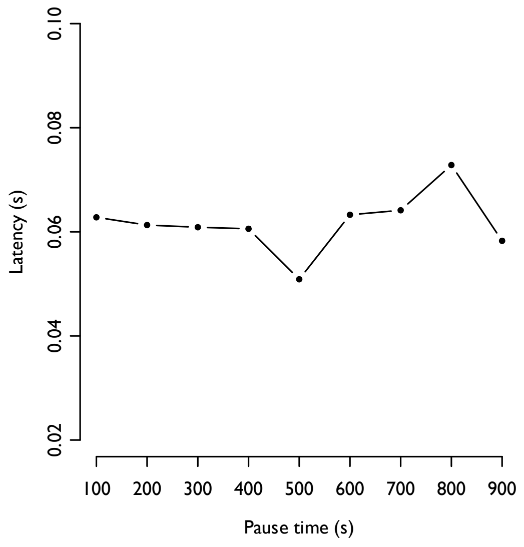
In order to provide a reference point for the results produced by this simulation model, I recreated experiments from a previously published paper [26]. These tests use a 1500 m x 300 m field. The broadcast range is set to 250 m, the transmission delay to 0.02 seconds, and speed is uniformly distributed between 0 and 20 m/s. There are 50 nodes, 10 of which transmit five packets per second to another node. Source and destination nodes are chosen uniformly at random. The simulation was run for 900 seconds and the pause time was varied between 100 and 900 seconds, which means that the nodes are entirely stationary at the maximum pause time.

The results of this test are illustrated in Figure 5-7, and essentially match the results found in [26]. AODV finds efficient routes and has very low latency as long as the channel is uncongested. Because this simulation does not model message collisions, the message latency shown in 5-7b is low. The message overhead measured in this experiment compares well with that in [26]. With low pause times, AODV is forced to broadcast roughly one routing message for every data message. As pause time increases and link changes decrease, message overhead approaches zero.

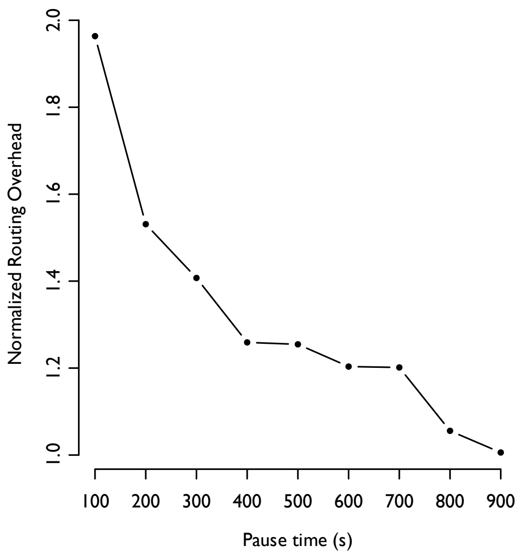
However, these results show a substantially degraded packet delivery rate at high node mobility, whereas the results in [26] show nearly perfect delivery at high node mobility and low channel congestion. Performance in these circumstances is primarily dependent on rapidly detecting and repairing broken links. The implementation of AODV in [26] uses link-layer feedback rather than HELLO messages to detect link breaks, which likely accounts for the performance disparity.



(a) AODV Delivery Fraction

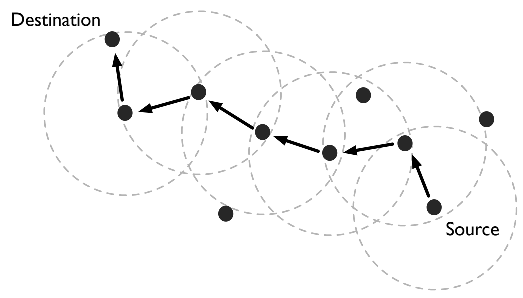


(b) AODV Delivery Latency



(c) AODV Routing Overhead (Excluding HELLO messages)

Figure 5-7: AODV Routing Performance. These tests were run for 900 seconds, with nodes moving in a 1500 m x 300 m field. The broadcast radius was set to 250 m. The transmission delay was set to 0.02 seconds.



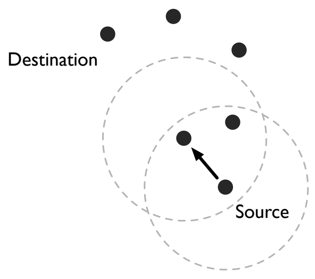


Figure 5-8: The greedy geographic routing algorithm always forwards messages to the neighbor nearest the destination, as in (a). However, if every neighboring node is farther from the destination than the current node, as shown in (b), the greedy heuristic fails.

5.3 Greedy Geographic Routing

One of the interesting aspects of using the RVN Layer for routing is that it implies partial knowledge of node positions. The routing system doesn't necessarily know where any individual physical node is, but it does know the location of RVN. AODV, in contrast, doesn't make any assumptions at all about geographical layout. This section examines the opposite extreme: routing with perfect knowledge of all node positions.

The basic greedy geographic routing algorithm, illustrated in Figure 5-8a, is extremely simple. If a node a wants to send a message to another node b, then a forwards the message to the neighbor that is closest to node b. If all of node a's neighbors are farther away from b than a, as illustrated in Figure 5-8b, then routing fails.

5.3.1 Implementation Notes

The greedy geographic protocol implementation [5] is extremely simplistic. If greedy routing fails, the implementation supports two options: drop the message or buffer the message for a configurable period of time. It does not support perimeter routing modes in the style of GPSR. This limitation doesn't turn out to be a problem in the experiments here. Furthermore, the implementation relies on simulator primitives in order to determine the location of the destination node and of all neighboring nodes. This implementation means that there isn't any measurable message overhead associated with maintaining the routing protocol.

Of the other routing protocols examined in this thesis, greedy geographic routing is most similar to the RVN geocast. However, it has distinctly more information available and essentially zero message overhead.

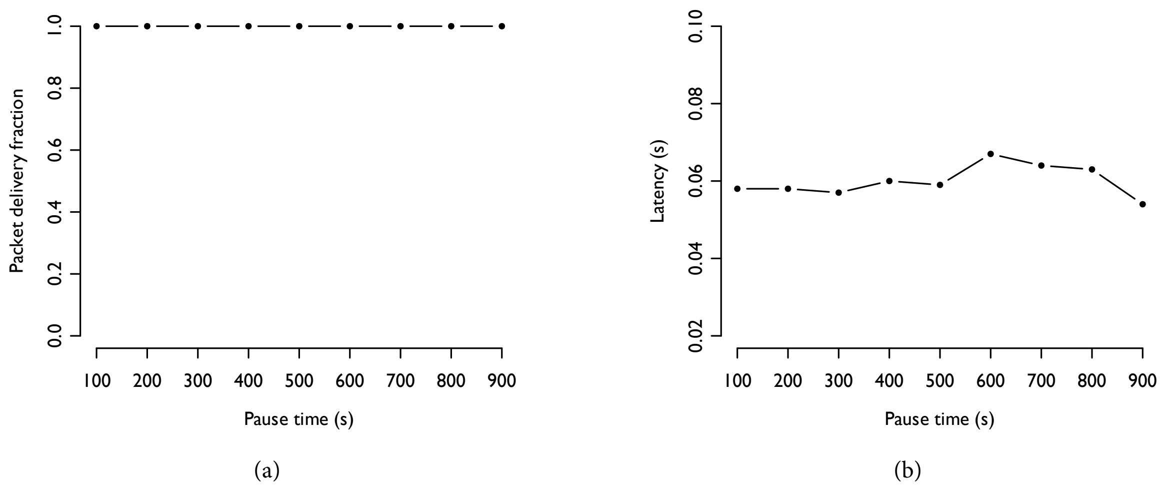


Figure 5-9: Greedy Geographic Routing Performance

5.3.2 Performance

Because this protocol relies on simulator primitives to determine the location of neighboring nodes, message overhead isn't an interesting metric because none exists. However, both the packet delivery fraction and average latency have some meaning. The packet delivery fraction will largely reflect the probability that a randomly chosen pair of nodes will have a valid greedy path between them. The average message latency will be a function of the average number of hops in that path.

In order to measure the performance of the greedy protocol, I used the same procedure as in measuring the performance of AODV and the overhead of RVNs . Using 25 physical nodes, 10 nodes were randomly designated senders and 10 were randomly designated receivers. Each node broadcast one message per second of simulation time. Mobility was modelled using the same random waypoint traces as in the RVN overhead experiments. Speed was uniformly between 0 and 20 m/s. The physical broadcast radius was set to 250 m and the broadcast delay was set to 0.02 seconds.

The results are summarized in Figure 5.3.2. The measurements show two things:

- If a reasonable number of nodes are evenly distributed in a square region, then a greedy path will almost certainly exist between any two of them.
- Given a 400 m x 400 m square and a 250 m broadcast radius, this path will typically be three hops long.

[5]Source code for the greedy geographic routing protocol is available at `https://carbide.mit.edu/trac/vne/browser/trunk/code/apps/georoute/georoute.py`.

The problem with these tests is that they are only looking at half of a routing protocol. Although the greedy route metric looks good given a square region of evenly distributed nodes, it ignores the very hard problem of efficiently distributing the location of every node throughout the entire network. This separation of concerns occurs cleanly in the Geocast protocol, because Geocast can rely on the Client Location Service. This greedy algorithm cannot use the same service, because it does not use the RVN Layer.

5.4 Performance Comparison

Of the routing protocols examined in this chapter, it seems clear that AODV performs the best. Unlike the RVN routing protocols, it maintains low latency and reliable packet delivery in low density, high mobility conditions. The implementation of AODV tested in this paper relies on HELLO messages to detect link breaks, and these messages contribute to routing overhead. However, they can be removed by using link-layer feedback for the same purpose. This change would simultaneously improve the measured delivery fraction for AODV in high mobility situations.

However, the RVN Layer routing protocols clearly demonstrate that the RVN Layer is a viable platform for writing routing applications. As long as the layer itself is stable, the RVN routing protocols tended to pay a small, multiplicative constant, penalty in terms of latency. Yet, the convenience that comes from having access to the abstraction layer makes this overhead acceptable.

Although the greedy routing protocol looks very nice on paper, it is not a complete protocol and relies on information that would not be available outside of a simulator. It is interesting to note that both the greedy protocol and AODV show an average latency of 0.06 seconds, corresponding to an average hop length of three. The RVN Geocast, while running in a very stable network, has an average latency close to 0.12 seconds. Considering that individual hops between RVNs are twice as expensive as normal, its routes tended to have roughly the same number of hops as other protocols.

Nonetheless, these results prompt more questions than they answer about routing with the RVN Layer. There are a variety of avenues to further examine the performance characteristics of RVN routing protocols and to compare them to conventional protocols. One of the key differences between RVN Geocast and AODV is that the Geocast protocol was originally designed to be a very safe, self-stabilizing algorithm that could recover from cases of extreme node failure. It uses end-to-end acknowledgements and allows every virtual node along the path to retry communication along new routes. AODV, in contrast, does neither of these things. One question to ask, is whether or not they add any benefit? It seems likely that they improve performance in the face of node failure and message loss. In order to test this, it would be interesting to measure the performance difference between Geocast and AODV in the presence of node failure, message loss, and other adversarial conditions.

It's also important to remember, and would be interesting to investigate, that the Geocast presented in this paper can be tuned differently. For example, it seems likely that a less careful approach would be much faster. For example, one virtual node might pass messages to the neighboring node that is closest to the destination, receive an immediate acknowledgement, and then wash its hands of the matter entirely. This send-and-forget approach has much more in common with AODV than the current Geocast implementation.

One of the other gaps in the set of experiments is in terms of mobility. Although the experiments here tested a very wide range of pause times, none of the experiments isolated node velocity. Additionally, all of the pause times were relatively large compared to to the broadcast message delay. It would be interesting to see a comparison of Geocast to AODV with the pause time set to zero, that is, with constant node movement. This experiment would compare the speed at which AODV can build and repair routes to the speed with which RVNs can deal with leadership changes and state loss.

This implementation of Geocast has other quirks that should be investigated. One reason that the Geocast latency appeared to be so large, is that it didn't impose timeouts on a per-message basis. In some cases, messages managed to spend a very long time bouncing around the network. They were eventually delivered, but only after the network had rearranged itself. It would be interesting to see what the overall latency looks like if messages were forced to time out after a relatively short period of time.

The precise source of the Client-to-Client Service's performance problems also needs to be investigated in greater detail. One potential reason for its problems is that it depends on the Client Location Service to determine the location of the destination node. However, the Client Location Service necessarily queries the client's home region, even if the client itself is relatively nearby. AODV doesn't suffer from this problem. The precise performance penalty caused by the extra delay should be measured.

It would also be interesting to quantify the difference between the way that AODV and the RVN Client-to-Client Service reuse routing table information. Both routing protocols require a route discovery phase – Geocast uses the Client Location Service and AODV floods RREQ messages – however the information that they discover is valid for different periods of time. The results of a Client Location lookup are valid until the client moves. However, AODV's routing table information can break whenever any intermediate node moves. AODV includes some optimizations to mitigate this effect. For example, it allows a node to repair a broken link locally rather than propagating the error throughout the rest of the network. Yet, it is not entirely clear how these differences manifest in terms of performance.

Chapter 6

Conclusion

This thesis has presented a definition and an implementation of the Reactive Virtual Node Layer, a simulator for mobile ad hoc networks, and set of simulation results evaluating the performance of several MANET routing protocols. In this chapter, I review the contributions of this thesis and conclude by suggesting a variety of avenues for future research.

6.1 Contributions

This thesis makes several contributions to the field of ad hoc network simulation and MANET routing. In this thesis, I have:

- Presented a new network simulator that is written in Python and specialized for mobile ad hoc networks.

  This thesis introduced a new tool for simulating ad hoc networks. Unlike existing solutions, this tool is primarily implemented in a highly dynamic language, which makes the simulator easy to develop and removes the impedance mismatch between simulated software and the simulator itself. Although there is a substantial amount of work left to do before the simulator is useful for large scale, physically accurate simulations, it is immediately useful for prototyping routing algorithms and for performing relatively small scale simulation.

- Developed a new visualization tool for simulated ad hoc networks

  Most MANET simulation focuses on specific, quantitative measures of routing efficiency. Yet, purely quantitative measurements of complicated systems often risk missing the forest for the trees. This thesis presented a new visualization framework for understanding behavior in mobile ad hoc networks.

- Defined the Reactive Virtual Node Layer

  This thesis presented a and defined a reactive virtual infrastructure abstraction, the Reactive Virtual Node Layer.

- Provided a working implementation of the RVN Layer for simulation

  One of the key contributions of this thesis is an actual implementation of the RVN Layer abstraction. Although the implementation runs in simulation only, it opens the door to more practical research with the abstraction.

- Compared the performance of three MANET routing protocols with simulation

  This thesis uses the RVN Layer implementation to simulate a VN routing algorithm. In order to better understand the performance figures, they are compared with two conventional routing schemes.

6.2 Evaluation

Although the Python simulator framework that is presented in this thesis is pleasant to use compared to conventional network simulators and is very useful for prototyping network protocols, it has some significant problems.

The single most significant failure is performance. It simply isn't possible to gloss over the fact that its performance is dismal. Even though switching between tasklets in Stackless Python is relatively cheap compared to switching between real processes or kernel threads, the simulator implementation simply does it too often. If this simulator is ever going to become a viable platform for anything but prototyping, it will have to gain a better ability to replace context switches with simple method calls. Once that happens, the constant factor slowdown due to using an interpreted language can be whittled away with careful optimization.

Additionally, the original basis for analyzing simulation results, detailed log files, turned out not to work well; the context that would otherwise be available due to the program structure is missing from log files, which means that it's necessary to write large, tedious programs that carefully recreate the missing context. A more effective approach is to write a single statistics-collecting object that offers a simple interface to the rest of the code. While the simulation is running, code can easily invoke methods on the statistics object. Once the simulation is complete, the entire object should be serialized to a file. Analysis is much easier because the data remains in its most useful form.

6.3 Future Work

This section proposes a number of potential avenues for extending the work that was presented in this thesis.

6.3.1 Simulator Enhancements

One of the serious limitations of the Python VNE implementation and the Python simulation framework is simulation fidelity. The Python simulator makes development relatively easy, however it does not implement detailed radio or channel models. In order to overcome some of these limitations, my work was recently adapted by Jiang Wu. He implemented the VN framework using the ns-2 simulator [27] to evaluate virtual nodes for solving address allocation in ad-hoc networks.

Nonetheless, it would be extremely beneficial to combine the ease of use and development that comes from a pure Python simulator with the fidelity of detailed radio and channel models. These changes might involve a substantial amount of work on two fronts: basic model implementation and performance optimization. Python is a fantastic language for rapidly developing software, and implementing a simple mathematical model of radio propagation should be straightforward. However, radio and channel models are necessarily triggered frequently and are performance sensitive.

There are several potential solutions to this problem. One option is to simply wait for better Python interpreters. Multiple projects are attempting to, for example, port the Python language to the Java Virtual Machine or implement optimizing JIT compilers for the language. However, these projects are probably going to take several years to mature significantly. Another option would be to implement the models as compiled extensions. This second option has its own limitations. Even though Python extensions can be quite fast, the penalty from calling out to the extension accumulates over the course of a long simulation.

6.3.2 Visualizing Distributed Systems

Although the current incarnation of the Viz is very useful for debugging virtual node applications and for observing node behavior, its utility for visualizing arbitrary MANET software is limited. The viz doesn't really support custom graphic layers, and there isn't any support for adding new sources of data.

There are a variety of features that I think would substantially improve the visualization. There should be an easy to use system for writing custom visualizations that run in same context and can take advantage of the RVN log file parser. Furthermore, it would be nice if these custom layers could by written in Python. Custom visualization layers should, presumably, be able to implement a user interface for dynamically interacting with them as well.

The visualization display itself could also use some improvements. One of the most important dimensions in a process visualization is time. However, the Viz is only able to show a single moment in time at once. One interesting avenue of development would be to improve the spatial mapping of time in MANET visualization. For example, adding a small multiples display that shows the network at time t, $t-5$, $t-10$, etc. would make it easier to see the evolution of the network over time.

6.3.3 MANET Routing

Although it seems clear that the RVN Layer is potentially useful for routing in MANETs, there is a substantial amount of work that remains to be done. In addition to the experiments and protocol modifications suggested in Section 5.4, there are at least two other avenues for future research:

6.3.3.1 Comparing Virtual Node Abstractions

This thesis examined three different protocols for routing in MANETs. However, it focused on only one routing protocol that used the RVN Layer. Moreover, the protocol was originally designed to run on VSAs. It would be interesting to see if a routing service that was designed from the ground up for an RVN could achieve better performance.

The implementation of the VN routing protocol used the Python RVN simulator also presented in this thesis. One important question is whether or not the protocol would perform better if it was implemented on a VSA, and if so, to what degree?

6.3.3.2 Comparing Routing Protocols

Point-to-point routing services can be cleanly decomposed into three tasks: locating the destination, establishing a route to the destination, and forwarding messages to the destination. AODV combines the first two tasks, and RVN Geocast combines the last two. Thus, there are other combinations that should be tested. In particular it would be interesting to consider an RVN routing service that combines client location and route establishment into a single phase.

Another open question stands in respect to understanding the interaction between traditional, node-based protocols and virtual node systems. Does it make sense to implement part of a system using RVNs, run a more traditional protocol on the client nodes, and let the traditional protocol query the VN Layer? If one of the primary strengths of RVNs is that they are easy to program, then this strategy might be a reasonable approach to prototyping distributed systems.

One simple protocol to test this approach with is the greedy geographic routing protocol: What happens when virtual nodes are used to run a location service and a simple, physical node-based protocol is used to route messages? It seems likely that this combination could slightly increase routing performance compared to a purely RVN-based algorithm, but it may come at the cost of lower reliability.

Bibliography

[1] S. Dolev, S. Gilbert, N. Lynch, E. Schiller, A. Shvartsman, and J. Welch, “Virtual mobile nodes for mobile ad hoc networks,” in *18th International Symposium on Distributed Computing (DISC04)*, Oct. 2004.

[2] S. Dolev, S. Gilbert, E. Schiller, A. Shvartsman, and J. Welch, “Autonomous virtual mobile nodes,” in *Third Annual ACM/SIGMOBILE International Workshop on Foundation of Mobile Computing (DIAL-M-POMC 2005)*, Cologne, Germany, Sep. 2005.

[3] S. Dolev, L. Lahiani, N. Lynch, and T. Nolte, “Self-stabilizing mobile node location management and message routing,” in *Seventh International Symposium on Self-Stabilizing Systems (SSS 2005)*, Barcelona, Spain, Oct. 2005.

[4] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte, “Timed virtual stationary automata for mobile networks,” in *9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, Pisa, Italy, Dec. 2005.

[5] Friefunk. [Online]. Available: http://start.freifunk.net

[6] FunkFeuer. [Online]. Available: http://www.funkfeuer.at/

[7] B. A. Chambers, “The grid roofnet: a rooftop ad hoc wireless network,” Master’s thesis, Massachusetts Institute of Technology, June 2002.

[8] Meraki Networks Inc. [Online]. Available: http://www.meraki.com

[9] Metrix Communication LLC. [Online]. Available: http://www.metrix.net

[10] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch, “Geoquorums: Implementing atomic memory in ad hoc networks,” in *17th International Symposium on Distributed Computing (DISC 2003)*,

ser. Lecture Notes in Computer Science, vol. 2848. Sorrento, Italy: Springer-Verlag, Oct. 2003, pp. 206–320.

[11] M. Brown, S. Gilbert, N. Lynch, C. Newport, T. Nolte, and M. Spindel, "The virtual node layer: A programming abstraction for wireless sensor networks," in *Proceedings of the the International Workshop on Wireless Sensor Network Architecture (WWSNA)*, Cambridge, MA, Apr. 2007, to appear.

[12] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers," in *Proceedings of the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications*. London, United Kingdom: ACM, 1994, pp. 234–244.

[13] C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing," in *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, New Orleans, LA, USA, February 25–26 1999.

[14] Z. Haas, "A new routing protocol for the reconfigurable wireless networks," in *Proceedings of 6th IEEE International Conference on Universal Personal Communications, IEEE ICUPC'97*, vol. 2. San Diego, California: IEEE, Oct. 1997, pp. 562–566.

[15] D. B. Johnson, D. A. Maltz, and J. Broch, *DSR: the dynamic source routing protocol for multihop wireless ad hoc networks*. Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 139–172.

[16] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot, "Optimized link state routing protocol for ad hoc networks," in *Proceedings of the 5th IEEE Multi Topic Conference, IEEE INMIC'01*, 2001, pp. 62–68.

[17] B. Karp and H. T. Kung, "GPSR: greedy perimeter stateless routing for wireless networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking, MobiCom 2000*. Boston, Massachusetts, United States: ACM, 2000, pp. 243–254.

[18] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, "The theory of timed i/o automata," *Synthesis Lectures on Computer Science*, 2006.

[19] (2008) Stackless python. [Online]. Available: http://www.stackless.com

[20] R. Brown, *Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem*. ACM Press, 1988, vol. 31.

[21] G. Yan and S. Eidenbenz, “Sluggish calendar queues for network simulation,” in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2006. MASCOTS 2006. 14th IEEE International Symposium on*, 2006, pp. 127– 136.

[22] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, pp. 558–565, 1978.

[23] D. Estrin, M. Handley, J. Heidemann, S. McCanne, Y. Xu, and H. Yu, “Network visualization with nam, the vint network animator,” *Computer*, vol. 33, pp. 63–68, 2000.

[24] O. Ben-Kiki, C. Evans, and B. Ingerson. (2005) YAML specification. [Online]. Available: http://www.yaml.org/spec/

[25] C. Perkins, E. Belding-Royer, and S. Das, “Ad hoc on-demand distance vector (aodv) routing,” Jul. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3561.txt

[26] C. Perkins, E. Royer, S. Das, and M. Marina, “Performance comparison of two on-demand routing protocols for ad hoc networks,” *Personal Communications, IEEE [see also IEEE Wireless Communications]*, vol. 8, pp. 16–28, 2001.

[27] The network simulator ns-2. [Online]. Available: http://www.isi.edu/nsnam/ns/

[28] D. P. Bertsekas and R. G. Gallager, *Distributed Asynchronous Bellman-Ford Algorithm*. Englewood Cliffs: Prentice Hall, 1987, pp. 325–333.