Virtual Ring Routing: Network Routing Inspired by DHTs

Matthew Caesar*[2], Miguel Castro[1], Edmund B. Nightingale*[3], Greg O'Shea[1], Antony Rowstron[1]

[1] Microsoft Research
Cambridge, UK

[2] University of California Berkeley
Berkeley, USA

[3] University of Michigan
Ann Arbor, USA

mcastro,gregos,antr@microsoft.com mccaesar@cs.berkeley.edu enightin@eecs.umich.edu

ABSTRACT

This paper presents Virtual Ring Routing (VRR), a new network routing protocol that occupies a unique point in the design space. VRR is inspired by overlay routing algorithms in Distributed Hash Tables (DHTs) but it does not rely on an underlying network routing protocol. It is implemented directly on top of the link layer. VRR provides both traditional point-to-point network routing and DHT routing to the node responsible for a hash table key.

VRR can be used with any link layer technology but this paper describes a design and several implementations of VRR that are tuned for wireless networks. We evaluate the performance of VRR using simulations and measurements from a sensor network and an 802.11a testbed. The experimental results show that VRR provides robust performance across a wide range of environments and workloads. It performs comparably to, or better than, the best wireless routing protocol in each experiment. VRR performs well because of its unique features: it does not require network flooding or translation between fixed identifiers and location-dependent addresses.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Algorithms, Measurement, Performance, Reliability

Keywords

Network Routing, Distributed Hash Table, Wireless

1. INTRODUCTION

This paper presents Virtual Ring Routing (VRR), a new network routing protocol with a unique design. The design is inspired by Distributed Hash Table (DHT) overlays (e.g.,[38, 40, 39, 44]) but VRR is a network routing protocol. Whereas DHTs assume an underlying network routing protocol that provides connectivity between all pairs of nodes, VRR is implemented directly on top of the link layer. VRR provides not only traditional point-to-point network routing but also DHT functionality: it balances the load of managing hash-table keys across nodes and routes messages sent to a key to the node responsible for managing the key.

VRR is also unique because it never floods the network and uses only location independent identifiers to route. Nodes are organized into a virtual ring ordered by their identifiers and each node maintains a small number of routing paths to its neighbors in the ring. The nodes along a path store the next hop towards each path endpoint in a routing table. VRR uses these routing tables to route packets between any pair of nodes in the network: a packet is forwarded to the next hop towards the path endpoint whose identifier is numerically closest to the destination. The paths between virtual ring neighbors are setup using this algorithm.

VRR can route over any link layer technology but this paper focuses on wireless ad hoc environments. We believe that DHT functionality is particularly useful in these environments because it can be used to implement scalable network services in the absence of servers. Furthermore, VRR addresses some performance issues with previous wireless routing protocols.

VRR performs well across a wide range of environments and workloads because it does not flood and it does not use location-dependent addresses. Proactive wireless routing protocols flood on topology changes (e.g., [34, 6]) and reactive protocols flood to discover routes (e.g., [22, 35]). Hybrid protocols perform scoped floods on topology changes and flood to discover routes to nodes in other regions of the network (e.g., [18, 37]). Previous protocols that do not flood use location-dependent addresses to route (e.g., [5, 33, 24, 26, 15]), which has some disadvantages. Location-dependent addresses can change with mobility and, in some protocols, with congestion and failures [5, 33, 15]. These changes can result in losses and increased congestion, and usually require mechanisms to lookup the location of a node given a fixed identifier [28].

We compared VRR against a number of representative wireless routing protocols using both simulations and measurements from real implementations running on a sensor network and an 802.11a Windows PC testbed. The Windows implementation modifies the Mesh Connectivity Layer (MCL) [10] to support VRR. The VRR network appears to an unmodified TCP/IP protocol stack as a single virtual link. Therefore, all IP-based applications can be run over the VRR network without modification.

The experiments show that VRR provides robust performance across a wide range of environments and workloads. It performs comparably to, or better than, the best routing protocol in each experiment.

The paper is organized as follows. Section 2 presents an overview. Section 3 describes VRR in detail. We evaluate performance in Section 4. Section 5 describes related work and Section 6 concludes.

*Work done during an internship at Microsoft Research Cambridge.

SIGCOMM'06, September 11–15, 2006, Pisa, Italy.

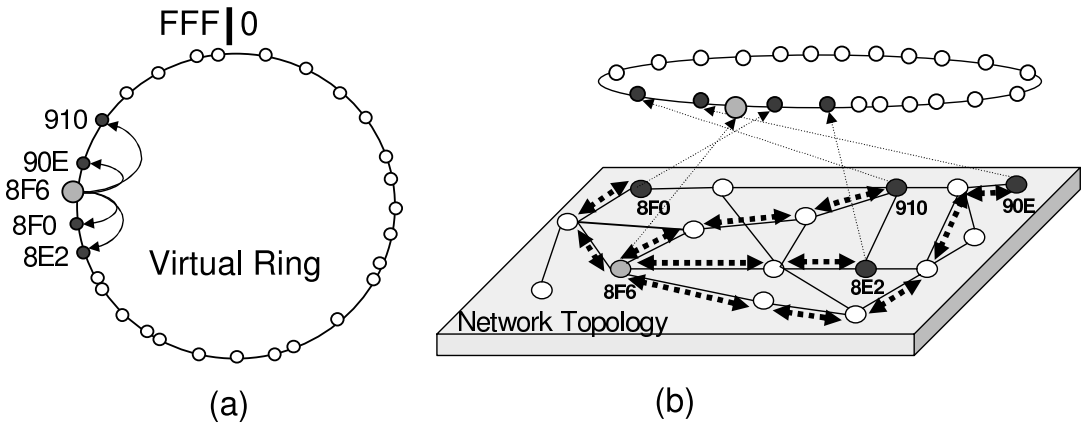


Figure 1: Relationship between the virtual ring and the physical network topology.

2. OVERVIEW

VRR uses random unsigned integers to identify nodes, and organizes the nodes into a *virtual ring* in order of increasing identifier (with wrapping around zero). Node identifiers are fixed, unique, and location independent. To maintain the integrity of the virtual ring with node and link failures, each node maintains a virtual neighbor set (or *vset*) of cardinality r containing the node identifiers of the $r/2$ closest neighbors clockwise in the virtual ring and the $r/2$ closest neighbors counter clockwise.

Each node also maintains a physical neighbor set (or *pset*) with the identifiers of nodes that it can communicate with at the link layer. Since link quality can vary widely in wireless environments, it is important for nodes to estimate the quality of wireless links to candidate physical neighbors. A node only adds a neighbor to the pset if the quality of the links to and from that neighbor is above a threshold. In addition, VRR nodes can take link quality into account when making forwarding decisions. Sections 4.2 and 4.3 describe implementations that estimate link quality using packet loss and bandwidth metrics but it is possible to use other metrics.

Figure 1(a) shows an example virtual ring with a 12-bit identifier space (with identifiers in base 16). It also shows the vset of the node with identifier $8F6$ with $r = 4$.

VRR sets up and maintains routing paths between a node and each of its virtual neighbors. These are called *vset-paths*. Since node identifiers are random and location independent, the virtual neighbors of a node will be randomly distributed across the physical network. So vset-paths are multi-hop in most cases. They are also bidirectional because membership in the vset is symmetrical (if node x is in the vset of node y then node y is in the vset of x).

The routing information for a vset-path is stored in the *routing tables* of the nodes along the path. Each node maintains a routing table with information about the vset-paths to its virtual neighbors and other vset-paths that are routed through the node. A routing table entry identifies the two vset-path endpoints and the next hop towards each endpoint. This information is maintained proactively, i.e., it is maintained even when there is no traffic along the path.

Figure 1(b) shows the mapping between the virtual ring and the physical network topology and it shows the vset-paths between node $8F6$ and its virtual neighbors.

VRR does not setup or maintain paths between nodes that are not virtual neighbors because vset-paths can be used to route packets between any pair of nodes. VRR nodes route packets to destination identifiers by forwarding them to the next hop towards the path endpoint whose identifier is numerically closest to the destination identifier from among all the endpoints in their routing table.

If there is a correct vset-path between each node and its virtual neighbors, VRR can route between any pair of nodes by following the vset-paths between neighboring nodes along the ring. But VRR does better because each node uses not only the vset-paths to its virtual neighbors but also vset-paths between other nodes that happen to be routed through it. The following approximate analysis provides some intuition into how this works. If each node maintains r vset-paths to its virtual neighbors and the average path length is p, the total number of routing table entries in an n node network is nrp. Therefore, each node will have on average rp entries for vset-paths in its routing table: r entries for the paths to its virtual neighbors and $r(p-1)$ additional entries for vset-paths through the node. If we assume that these additional vset-paths end at nodes that are selected randomly and uniformly, the probability that a random node has a path to a random destination is $O(rp/n)$. Therefore, a packet is expected to reach a node that has a vset-path to the destination after visiting $O(n/(rp))$ nodes, which will add only a constant stretch if p grows with $\sqrt{n}$ (as in wireless ad hoc networks).

VRR provides not only point-to-point network routing between two nodes but also a distributed hash table (DHT) [40, 38, 44, 39]. VRR routes messages sent to numerical keys to the node whose identifier is numerically closest to the key. These keys can identify application objects instead of VRR nodes. We believe that DHT functionality is particularly useful in wireless ad hoc scenarios where there may be no servers to coordinate nodes but we do not explore specific applications in this paper. VRR could easily support peer-to-peer applications like directories, instant messaging, cooperative caching, and cooperative storage.

VRR does not impose any structure on node identifiers. It only requires that they be unique and totally ordered. Therefore, node identifiers can be generated in different ways to suit specific purposes. For example, an identifier could be the 160-bit SHA-1 hash of a node's public key to facilitate secure communication [30], or a randomly selected 32-bit integer to provide backwards compatibility with IPv4 addresses. It is also possible to use certified node identifiers as described in [4] to prevent Sybil attacks [9].

VRR does not use flooding and it uses only location independent identifiers to route. All control and data packets are routed as described above without any translation to location based addresses. In particular, control messages to setup new vset-paths are routed using the existing vset-paths. Additionally, VRR can usually route around failed paths without requiring them to be repaired because there are usually many routes between each pair of nodes. These features allow VRR to offer robust performance across a wide range of environments and workloads.

3. VIRTUAL RING ROUTING

This section presents VRR in detail. It starts by describing the routing state maintained by nodes and how it is used to forward messages. Then it describes how this state is maintained when nodes join and when nodes or links fail.

3.1 Forwarding

Each node maintains a routing table with an entry for every vset-path that includes the node. Each entry contains the identifiers of the two endpoints of the path, the identifier of the physical neighbor to be used as the next hop towards each endpoint, and a vset-path identifier. The first endpoint identifier in an entry is always the identifier of the node that initiated the vset-path setup.

Figure 2 continues the example from Figure 1 by showing the routing table of node 8F6. The first four entries in the table are for the vset-paths from the node to its four virtual ring neighbors.

Since node 8F6 is an endpoint in these paths, the identifier of the next hop towards the node is null. The 5th and 6th entries in the table are for two vset-paths that are routed through node 8F6. VRR maintains the invariant that the next_A and next_B fields in a node's routing table entries are in the pset of the node.

endpoint_A	endpoint_B	next_A	next_B	path id
8F0	8F6	20E	null	03
8E2	8F6	F01	null	2F
8F6	90E	null	7E2	1E
910	8F6	F01	null	2F
35F	37A	20E	7E2	12
A01	A10	F01	FC1	F0
8F6	20E	null	20E	FF
8F6	F01	null	F01	FF
8F6	7E2	null	7E2	FF
8F6	FC1	null	FC1	FF

Figure 2: Sample routing table for the node with identifier 8F6. The first four entries are vset-paths to 8F6's virtual neighbors, the fifth and sixth entries are for vset-paths that happen to be routed through 8F6, and the last four are paths to 8F6's physical neighbors.

VRR also inserts one-hop paths to physical neighbors in the routing table to simplify routing. These are the last four entries in the table and have the special path identifier FF.

The routing table in the example also shows that vset-path identifiers are not necessarily distinct. The vset-path identifier is assigned by endpoint_A, which is the node that initiates the path setup, such that each vset-path is uniquely identified by the pair $\langle$path id,$\text{endpoint}_A\rangle$. Vset-path identifiers can be small; each node originates at most one vset-path to each of its r virtual ring neighbors and nodes can reuse the identifiers of torn down paths after a probation period to ensure that there are no routing table entries with those identifiers.

```
NextHop(rt, dst)
    endpoint := closest id to dst from Endpoints(rt)
    if (endpoint == me)
        return null
    return next hop towards endpoint in rt
```

Figure 3: VRR's forwarding algorithm. The identifier of the local node is *me* and *rt* is its routing table.

The forwarding algorithm used by VRR is very simple — VRR picks the node with the identifier closest to the destination from the routing table and forwards the message towards that node. The packet is delivered to the node with the identifier closest to the destination in the network. This is shown in Figure 3. When a node receives a packet destined to the node with identifier *dst*, it sets *endpoint* to the node identifier numerically closest to *dst* from among all the endpoint identifiers in the routing table, *rt*. If *endpoint* is the identifier of the local node, the function returns null and the packet is delivered locally. Otherwise, the next hop to reach *endpoint* is retrieved from the routing table and the packet is sent to that node. If there are several alternative paths to reach *endpoint* in the routing table, the algorithm uses one of the entries with the highest value of $\langle$path id,$\text{endpoint}_A\rangle$ to compute the next hop. This favors one-hop paths when present.

3.2 Node joins

When a node joins the VRR network, it initializes its pset and vset and it sets up vset-paths to its virtual neighbors. It finds its virtual neighbors by routing a message to its own identifier. This is all done without flooding the network by using existing vset-paths to route messages.

The joining node starts by looking for physical neighbors that are already active in the network and, therefore, can be used as proxies to route messages to others. It finds a proxy by sending and listening to *hello* messages that VRR nodes broadcast to physical neighbors periodically. These messages are also used to initialize the pset of the joining node as will be discussed in Section 3.3.

After finding a proxy, the joining node sends a *setup_req* message to its own identifier, x, through the proxy. This message is routed using the forwarding algorithm to the node whose identifier, y, is closest to x. Node y is one of the immediate virtual neighbors of the joining node in the virtual ring and it knows the identities of the other virtual neighbors of x.

Node y replies with a *setup* message that is routed back to the joining node through the proxy and it also adds x to its vset. This message sets up the vset-path between node y and the joining node by updating the routing tables of the nodes it visits. The joining node adds y to its vset when it receives the message.

The *setup* message also includes y's vset. The joining node uses the received vset to initialize its own; it sends *setup_req* messages to the identifiers of its other virtual neighbors. The joining node adds these neighbors to its vset when it receives *setup* messages from them. This completes all routing state initialization and the node becomes *active*.

Figure 4 shows pseudo code that describes in more detail the initialization of routing state. It introduces two additional message types: *setup_fail* messages are sent in reply to *setup_req* messages to indicate refusal to setup a vset-path to the source, and *teardown* messages are used to remove entries for vset-paths from the routing tables along the path.

A node replies to a *setup_req* message from x with *setup_fail* when it does not add x to its *vset*. This can happen when there are concurrent joins and the node learns about identifiers closer to its own than x. This message provides x with new destinations to send *setup_req* messages to.

Like *setup* messages, *setup_fail* messages are routed back to x through the proxy: they are routed towards the identifier of the proxy until they reach one of x's physical neighbors that sends it to x. This works because x picks a proxy that is a physical neighbor and this can shorten the path if the message visits another physical neighbor of x before reaching the proxy.

The setup of vset-paths may be aborted due to failures or concurrent setups as shown in the Receive function for *setup* in Figure 4. VRR aborts a vset-path setup by calling TearDownPath to remove all entries for the path from the routing tables of all nodes that may have been visited by the *setup* message. The first call to TearDownPath happens when the node receives the message from a node that is not in its *pset* or it already has the entry for the path

```
Receive(⟨setup_req, src, dst, proxy, vset'⟩, sender)
    nh := NextHopExclude(rt, dst, src)
    if (nh ≠ null)
        Send ⟨setup_req, src, dst, proxy, vset'⟩ to nh
    else
        ovset := vset;   added := Add(vset, src, vset')
        if (added)
            Send ⟨setup, me, src, NewPid(),proxy, ovset⟩ to me
        else
            Send ⟨setup_fail, me, src, proxy, ovset⟩ to me

Receive(⟨setup, src, dst, pid, proxy, vset'⟩, sender)
    nh := (dst ∈ pset) ? dst : NextHop(rt, proxy)
    added := Add(rt,⟨src, dst, sender, nh, pid⟩)
    if (¬added ∨ sender ∉ pset)
        TearDownPath(⟨pid, src⟩, sender)
    else if (nh ≠ null)
        Send ⟨setup, src, dst, pid, proxy, vset'⟩ to nh
    else if (dst = me)
        added := Add(vset, src, vset')
        if (¬added)
            TearDownPath(⟨pid, src⟩, null)
    else
        TearDownPath(⟨pid, src⟩, null)

Receive(⟨setup_fail, src, dst, proxy, vset'⟩, sender)
    nh := (dst ∈ pset) ? dst : NextHop(rt, proxy)
    if (nh ≠ null)
        Send ⟨setup_fail, src, dst, proxy, vset'⟩ to nh
    else if (dst = me)
        Add(vset, null, vset'∪{src})

Receive(⟨teardown, ⟨pid, e_a⟩, vset'⟩, sender)
    ⟨e_a, e_b, n_a, n_b, pid⟩ := Remove(rt, ⟨pid, e_a⟩)
    next := (sender = n_a) ? n_b : n_a
    if (next ≠ null)
        Send ⟨teardown, ⟨pid, e_a⟩, vset'⟩ to next
    else
        e := (sender = n_a) ? e_b : e_a
        Remove(vset, e)
        if (vset' ≠ null)
            Add(vset, null, vset')
        else
            proxy := PickRandomActive(pset)
            Send ⟨setup_req, me, e, proxy, vset⟩ to proxy

Add(vset, src, vset')
    for each (id ∈ vset')
        if (ShouldAdd(vset, id))
            proxy := PickRandomActive(pset)
            Send ⟨setup_req, me, id, proxy, vset⟩ to proxy
    if (src ≠ null ∧ ShouldAdd(vset,src))
        add src to vset and any nodes removed to rem
        for each (id ∈ rem) TearDownPathTo(id)
        return true;
    return false;

TearDownPath(⟨pid, e_a⟩, sender)
    ⟨e_a, e_b, n_a, n_b, pid⟩ := Remove(rt, ⟨pid, e_a⟩)
    for each (n ∈ {n_a, n_b, sender})
        if (n ≠ null ∧ n ∈ pset)
            vset' := (sender ≠ null) ? vset : null
            Send ⟨teardown, ⟨pid, e_a⟩, vset'⟩ to n
```

Figure 4: Pseudo code for VRR. The identifier of the local node is *me*, its virtual neighbor set is *vset*, its physical neighbor set is *pset*, and its routing table is *rt*. The functions that are not defined in the figure work as follows: NextHopExclude is identical to NextHop except that the last argument is excluded from the next hop computation to prevent the message from being routed back to the source; NewPid() returns a new path identifier that is not in use by the local node; Add(*rt*, $\langle e_a, e_b, n_a, n_b, pid\rangle$) adds the entry to the routing table unless there is already an entry with the same $\langle pid, e_a\rangle$; Remove(*rt*, $\langle pid, e_a\rangle$) removes and returns the entry identified by $\langle pid, e_a\rangle$ from the routing table; PickRandomActive(*pset*) returns a random physical neighbor that is active; Remove(*vset, id*) removes node *id* from the *vset*; ShouldAdd(*vset, id*) sorts the identifiers in *vset*∪{*id, me*} and returns true if *id* should be in the *vset*; and TearDownPathTo(*id*) is similar to TearDownPath but it tears down all vset-paths that have *id* as an endpoint.

being setup in the routing table. These loops are rare but can occur when vset-paths are being concurrently setup or torn down. Calling TearDownPath provides a clean and simple solution to deal with these infrequent loops. The other calls to TearDownPath happen if the message is delivered to the wrong node or a node that is no longer a virtual neighbor of the source. They can happen with failures and concurrent joins.

The Add function in Figure 4 is used to add members to a node's *vset*. A node only adds new members when it receives a *setup* or *setup_req* message from them to prevent convergence problems due to the addition of failed members. When nodes remove a member from their *vset* to make room for a new member, they teardown any vset-path to the removed member to inform it that it is no longer in the vset and to garbage collect redundant routing state.

To deal with concurrent joins, all messages in Figure 4 have a *vset'* field that contains the identifiers of the nodes in the *vset* of the source. When a node x receives a message, it invokes the Add function that sends *setup_req* messages to all identifiers in *vset'* that should be added to the local *vset*. This allows nodes to exchange their local views of the virtual ring until their views converge and the appropriate vset-paths are setup.

If a node cannot find an active neighbor to use as a proxy to join the network, it creates a new ring by making itself active after a timeout. This can partition the network into multiple rings. These rings are merged using the mechanism described in Section 3.3.4.

3.3 Node and link failures

VRR must detect failures and repair routing state in a timely fashion to ensure virtual ring consistency. To avoid the overhead of end-to-end probes or end-to-end heartbeats, VRR maintains hard routing state for vset-paths and it detects both node and path failures using only direct communication between physical neighbors. This section describes how we detect failures and repair routing state.

3.3.1 Symmetric failure detection

Most routing protocols use soft state (e.g., [22, 35, 18, 37]) because it is easier to maintain soft state consistent than hard state. We introduce a simple technique that makes it easy to maintain hard state consistent. We call this technique *symmetric failure detection*. It guarantees that if a node x marks a neighbor y faulty, y will also mark x faulty. We use this to ensure that routing state is correctly removed from the network on failures, and to implement reliable node and path failure notifications. For example, if x tears down a vset-path because it suspects the next hop y is faulty, symmetric failure detection ensures that y will teardown the other half of the vset-path and that both path endpoints will learn about the failure. This is related to the technique presented in [12] to detect failures in an overlay network.

To implement symmetric failure detection, each node x monitors the ability to communicate with its physical neighbors by broadcasting *hello* messages every T_h seconds. x also remembers the

nodes from which it has heard a *hello* during the last $2kT_h$ seconds. Typical values are $k = 4$ and $T_h = 1$. These nodes can be in one of three states: *linked* if the node received hellos from x and x received hellos from the node, *pending* if x received hellos from the node but does not know if the node received hellos from x, and *failed* if the link to the node has been marked faulty. Other nodes are in the *unknown* state. The pset of x is the set of nodes in the linked state and x also tracks whether these nodes are active. It is possible to further restrict pset membership by imposing minimum thresholds on link quality (as described in Sections 4.2 and 4.3) but we ignore this to simplify the exposition.

The *hello* messages include three sets to classify the nodes according to their states: a set with nodes that are linked and active, one with nodes that are linked but not active, and another with pending nodes. When node x receives a *hello* message from node y, it compares its state in the hello message with its local state for y. Then, it updates y's local state according to the state transition diagram shown in Figure 5. The edges in the diagram correspond to x's state in y's hello message, for example, a state of missing indicates that x does not appear in the message. Additionally, x marks y as *failed* if it does not receive *hello* message from y for kT_h seconds and it removes y from the set of failed nodes if it does not receive a *hello* for $2kT_h$ seconds. These state transitions ensure that a node can send and receive messages from nodes in the pset (linked state) and that failure detection is symmetric.

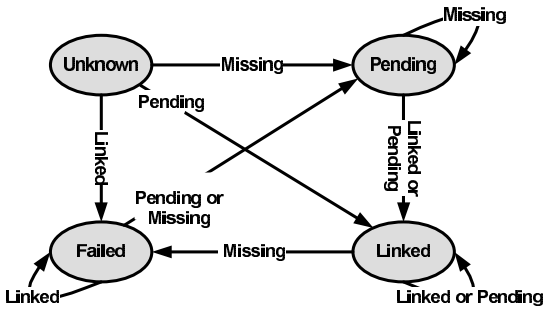


Figure 5: State transitions for physical neighbors when a *hello* message is received.

A *hello* message also indicates whether the sender is active or not. Whenever x determines that a physical neighbor z is both linked and active, it inserts a physical neighbor path to z in the routing table. From the information in *hello* packets from z, x is also able to determine the neighbors of z that are active and can be reached in two hops via z. As an optimization, these two-hop paths are also added to the routing table.

VRR also detects node and link failures by using per-hop acknowledgments and retransmissions for all messages except *hellos*. A node x marks a physical neighbor y failed when it does not receive acknowledgements for packets sent to y after some number of retransmissions.

3.3.2 Failure repair

When a node x marks a node y *failed*, it initiates the teardown of any vset-paths in its routing table that have y as a next hop. It does this by calling TeardownPath(p, null) (as defined in Figure 4) for each identifier p of a failed vset-path. Additionally, x removes any one- and two-hop paths through y from its routing table.

To ensure consistent routing state with concurrent failures, teardown messages are acknowledged and retransmitted. If a physical neighbor fails to acknowledge a teardown for a vset-path, it is marked *failed*, which triggers the sending of additional teardown messages that complete the teardown of the vset-path. This also provides a robust mechanism to abort incorrect vset-path operations when local consistency checks fail. This abort mechanism is used to handle several complex but infrequent corner cases, for example, to teardown a vset-path when a setup message loops.

Nodes repair vset-paths to their virtual neighbors when the paths fail. When a node receives a teardown message for a vset-path for which it is an endpoint, it removes the other endpoint from its *vset* and sends a *setup_req* message to that node, as shown in Figure 4. This message is retransmitted up to a maximum number of times (for example, five), which usually is sufficient to setup a new vset-path to the same neighbor if it is alive and it is reachable. When the neighbor is dead or unreachable, VRR delivers the *setup_req* message to the node with identifier closest to the dead virtual neighbor. If this node is the appropriate replacement neighbor, it replies with a *setup* message. Otherwise, it replies with a *setup_fail* message that almost always includes the identity of the replacement neighbor. When this mechanism fails to setup a vset-path to a replacement neighbor, VRR repeats the join procedure but this is rare even for small vset sizes (e.g., $r = 4$).

3.3.3 Local vset-path repair

Tearing down the full vset-path when a link fails can be unnecessary. Instead, VRR can perform *local repair* by replacing only the link that failed by an alternative route when possible. Local repair has been used before, for example, DSR [22] uses the route cache to find alternative routes on link failures but it communicates the new route back to the source.

VRR's vset-path repair mechanism is truly local — it only involves the nodes around the failed link. Since VRR does not rely on source routes or end-to-end path metrics like hop count, it can perform local repair without communicating with any of the endpoints. Therefore, the cost of local repair is constant in VRR whereas the cost of repair in previous mechanisms grows with the path length.

To support local repair, we extend the state stored in the routing table for each vset-path. When setting up a vset-path, each node stores both a $next_A$ and a $nextnext_A$ field for the path. They record the identifiers of the first and second hops towards the endpoint that originates the vset-path setup message (endpoint$_A$). This is achieved simply by adding a *prev* field to setup messages to record the node visited by the message before the sender.

The idea behind local repair is simple: when a node x detects a failed link to a node y, it determines the set of all vset-paths in the routing table that use y as their next hop. For each vset-path where $next_A = y$, x searches for an alternate $next_A$ that can bypass the failed link. If endpoint$_A$ is a physical neighbor, x patches the vset-path directly to endpoint$_A$. Otherwise, if $nextnext_A$ is a physical neighbor, x patches the vset-path to $nextnext_A$. These two cases are checked first because they allow x to repair and shorten the vset-path at the same time. If these checks fail, x searches for a physical neighbor with a link to $nextnext_A$. If there is such a neighbor w, the path can be repaired by replacing the failed link with a link from x to w and another link from w to $nextnext_A$. This transformation does not increase the path length, and finding w requires only a local search in the routing table because routing tables have paths to nodes within a 2-hop radius.

For the vset-paths where $next_B = y$, x simply delays the teardown by a period of $(k+1)T_h + \delta t$ seconds, which is the expected time for the node on the other side of the link to detect the failure plus some time to complete repair. If during that period it receives a message from a node wishing to repair the entry, it cancels the teardown. Otherwise, after the period it tears down the path.

The local repair algorithm uses simple local consistency checks that are conservative and trigger teardowns when they fail. It ensures that either the path is successfully repaired or torn down.

3.3.4 Partitions

Node and link failures may partition the network. When this happens, the algorithm ensures that nodes form separate rings. Typically, they form one ring in each partition, which enables the nodes in a partition to communicate with each other. However, the algorithm that we described so far is not sufficient to ensure that these rings converge to a single ring when the partition heals. This section describes a mechanism to ensure this.

The mechanism picks a *representative* from each separate ring and uses *hello* messages to maintain routes from each node to each representative. These routes are not vset-paths but they are inserted in the routing table like routes to one- and two-hop neighbors. When an active node learns about a representative that should be in its vset, it sends a *setup* message to that representative and adds the representative's identifier to its vset. The routes to representatives ensure that this message can be routed across partitioned rings. Receiving the *setup* triggers the vset stabilization mechanism described in Figure 4, which ensures that the separate rings are merged into one in the absence of further node and link failures. If the merge fails, the nodes form separate rings and the process is repeated.

The representative for a ring is the node whose identifier is closest to zero in the ring. Each node can determine locally whether it is a representative by inspecting its vset. VRR uses a mechanism similar to DSDV [34] to maintain routes between each node and a representative. It piggybacks updates to these routes in every *hello* message. The updates have a sequence number that is incremented by the representative before each *hello* to prevent loops. Nodes stop sending route updates for a representative if they do not receive an update with a fresh sequence number for more than kT_h seconds.

To keep the overhead low, nodes only send route updates for the two representatives whose identifiers are closest to zero from among those they have fresh routes to. This is sufficient to merge two rings at a time and it ensures that the overhead is constant. The partition repair mechanism does not add additional messages and only adds a small amount of data to *hello* messages. In contrast, route update messages in DSDV [34] have size $O(n)$. Additionally, we eliminate unnecessary messages by having nodes send a *setup* to a representative only when they receive route updates for two representatives in a *hello* message and only to the representative farthest away from zero.

4. EVALUATION

We evaluated VRR using both simulations on *ns-2* [1] and measurements of two prototypes running on different testbeds: a 67-node sensor network [20] and a 30-node 802.11a network of Windows PCs. The simulations compared the performance of VRR with DSR [22], AODV [35], and DSDV [34], which are representative wireless routing protocols with well tuned implementations in *ns-2*. DSR and AODV are reactive protocols and DSDV is a proactive protocol. We compared the performance of the sensor network prototype with BVR [15] and the performance of the Windows prototype with MR-LQSR [10, 11]. BVR is representative of the state of the art in protocols that route using location-dependent addresses, and MR-LQSR is representative of the state of the art in wireless mesh routing protocols.

We ran a large number of experiments. The results show that VRR performs well across all the experiments. Other protocols tend to perform well on some experiments but poorly on others.

4.1 Simulations

The simulation experiments ran on *ns*-2.27 using the wireless extensions developed by the CMU Monarch project [3]. They simulated an 802.11b wireless network running at 11Mbps. We ran a large set of experiments to explore the impact of different workload and environmental parameters on the performance of the routing protocols; we varied the rate of mobility, the traffic load offered by each node, the number of nodes, and the lifetime of network flows.

4.1.1 Protocols

We compared the performance of VRR, DSR, AODV and DSDV. This VRR implementation supports the local repair optimization. It was configured to use 4-byte node identifiers, a vset size of four ($r = 4$), and a hello period of one second ($T_h = 1s$). We used the default parameters for the other protocols, which are carefully tuned to this simulation environment. The four routing protocols unicast data packets, use link failure notifications, and do not use RTS/CTS. We modified all the routing protocols not to use the ARP protocol to translate the addresses of physical neighbors, because this can introduce significant delays that obscure the performance differences between the protocols. It is easy to populate ARP caches with the MAC addresses of physical neighbors using VRR's hello messages.

4.1.2 Experimental setup

Our experimental setup is very similar to the one used in [3] to facilitate comparison with previous work. The base configuration simulates 50 mobile nodes randomly distributed over a 1500m $\times$ 300m plane as in [3]. We vary the number of nodes from 25 to 200 and adjust the plane dimensions to keep the density of nodes per square meter constant and to preserve the aspect ratio. For example, we ran 200-node simulations in a 3000m $\times$ 600m plane.

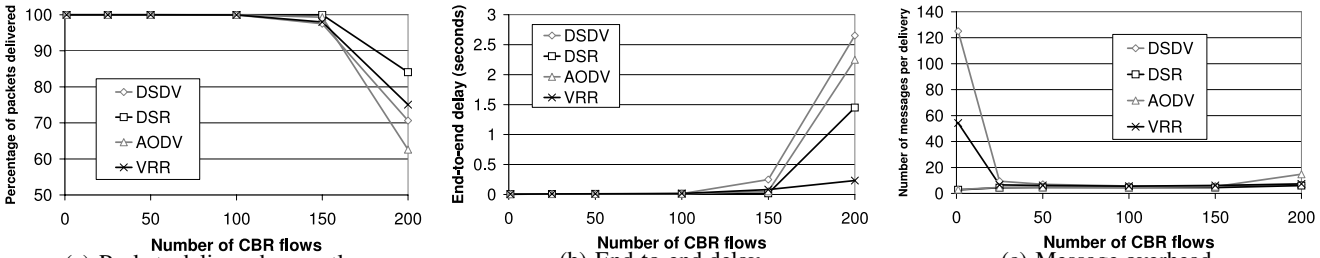
We ran experiments with and without mobility. The mobility patterns were generated using the random trip mobility model [2] that fixes the slow convergence problems [42] of the original random waypoint model [23]. In this model, each node selects a destination coordinate uniformly at random within the plane and moves towards that coordinate at constant speed. When it reaches the destination, the node selects a new destination and speed without pausing. We show results for two scenarios that represent extremes: the *static scenario* with no movement and the *20m/s mobility scenario* with fast movement. Nodes select speeds uniformly at random from the interval (0,20] m/s in the mobility scenario.

All the experiments ran for 1900 seconds with measurements taken only during the last 900 seconds. The initial 1000 seconds were used to ensure that the routing protocols reached steady state. The results that we present do not include the overhead to initialize routing information when the network starts. For VRR, this overhead was small and routing state converged fast. For example when starting a static network with 200 nodes, the average number of control messages per node (excluding hello messages) was 110.4 and all nodes were active after 24.3 seconds. In contrast, a single flood of the network requires 200 messages.

The experiments used a variable number of UDP constant bit rate sources (CBR) as in [3]. In the default configuration each node picks a random destination and starts sending 100 byte packets to that destination at a random time in the interval [1000,1180] seconds and at the rate of one per second [3]. We also ran experiments varying the number of CBRs and their lifetime.

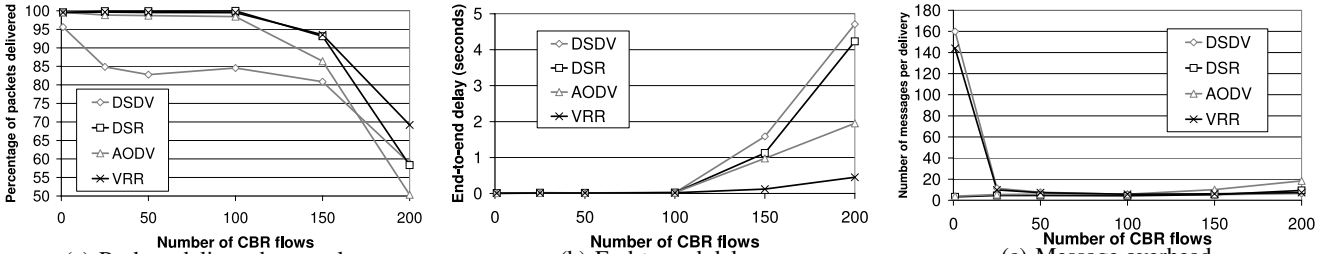
4.1.3 Evaluation metrics

We measured the fraction of CBR packets delivered correctly, the end-to-end delay for these packets, and the number of router-level messages per correct delivery (i.e., the number of messages passed down to the MAC divided by the number of CBR packets delivered correctly). Each experiment ran five times with different seeds and we present the average result for each metric. We used the same traffic, topology, and mobility patterns for all protocols.



(a) Packets delivered correctly (b) End-to-end delay (c) Message overhead

Figure 6: Performance with increasing number of CBR flows in the static scenario.



(a) Packets delivered correctly (b) End-to-end delay (c) Message overhead

Figure 7: Performance with increasing number of CBR flows in the 20m/s mobility scenario.

4.1.4 Performance with increasing traffic load

The first set of experiments compared the performance of the routing protocols with increasing traffic load while keeping the size of the network constant at 100 nodes. We varied the total number of CBR flows between 1 and 200. With less than 100 flows, we selected sources randomly such that each node sourced at most one flow. With 100 or more flows, each node sourced at least one flow and the sources for the additional flows were selected randomly such that no node sourced more than two flows. The destination of each CBR flow was selected randomly. Figures 6 and 7 show the results for the static and mobility scenarios, respectively.

In the static scenario, all protocols achieve nearly perfect delivery ratios and low delays with 100 flows or less. As the number of flows increases, the delivery ratio drops and delays increase due to congestion. The delays of DSR, DSDV, and AODV increase dramatically because they queue packets while they repair routes that fail due to congestion. This strategy improves delivery ratios but it results in high delays. Additionally, packets spend more time in the interface queues because of collision avoidance and packet retransmissions at the MAC layer. The delivery ratio decreases because packets are dropped when the router or interface queues fill up.

VRR achieves low delay because it never queues packets waiting for routes and it can achieve good delivery ratio because it can route around failed links most of the time.

Figure 6(c) shows a high overhead per delivery for DSDV and VRR with one CBR flow. This is because both protocols send control messages periodically between physical neighbors. DSR and AODV do not send these messages. In real wireless environments, periodic messages are required to estimate link quality [10].

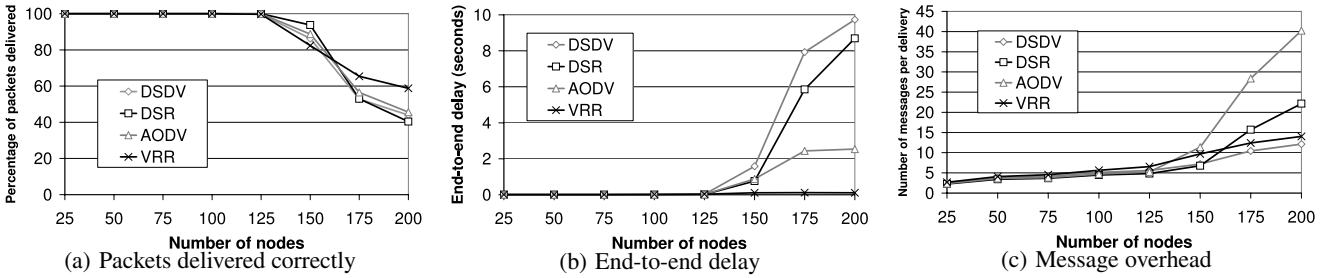
The results for the mobility scenario in Figure 7 show similar trends. The difference is that routes fail not only because of congestion as the number of flows increases but also because nodes move. For DSR, DSDV, and AODV, this results in more packets queued waiting for routes and even higher delays. DSDV achieves low delivery ratios even without congestion because routing tables are not sufficiently up to date. VRR achieves the highest delivery ratios and lowest delays because of the reasons mentioned before. Additionally, it tends to move vset-path links from fast moving nodes to slow moving nodes because links that do not fail do not change and those that fail are moved to new nodes. This simple mechanism to learn good routes is similar to the one used in DAR to perform dynamic routing in circuit-switched networks [25].

4.1.5 Performance with increasing network size

The second set of experiments evaluates the performance of the routing protocols as the number of nodes increases while keeping the traffic load offered by each node constant. We varied the number of nodes from 25 to 200 and each node sourced CBR traffic to a single random destination. Figures 8 and 9 show the results for the static and mobility scenarios, respectively.

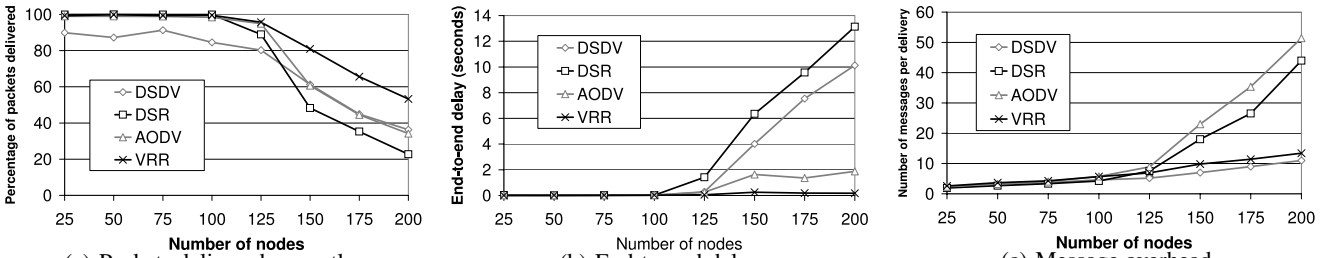
Figure 8 shows that all protocols achieve high delivery ratios and low delays with 125 nodes or less. With more nodes, delivery ratios drop and delays increase due to congestion. The delays of DSR, AODV, and DSDV grow high with more than 125 nodes. As in the previous set of experiments, the delays grow because these protocols queue packets waiting for routes that failed and also because packets spend more time in interface queues. Increasing the network size aggravates the problem because the message overhead to repair routes grows and longer routes are more likely to fail. The large increase in messages per correct delivery for DSR and AODV in Figure 8(c) illustrates this problem. These protocols incur a high overhead to repair a failed route because they use flooding. DSDV has a low message overhead because it uses damping to reduce the number of control messages and aggregates several routing table updates in a single control message. However, this results in large control messages and less consistent routing tables.

The results with mobility are similar but there are more route failures because nodes move. In both scenarios, VRR achieves low delays for all network sizes with good delivery ratios. It can do this for the reasons mentioned in the previous section and because it can repair routes with lower overhead than the other protocols.



(a) Packets delivered correctly (b) End-to-end delay (c) Message overhead

Figure 8: Performance with increasing network size in the static scenario.



(a) Packets delivered correctly (b) End-to-end delay (c) Message overhead

Figure 9: Performance with increasing network size in the 20 m/s mobility scenario.

4.1.6 Performance with short-lived flows

The final set of simulations compared protocol performance with short-lived flows. They used the same experimental setting as the previous set, except that nodes chose a random destination for each packet (instead of always sending packets to the same destination). Figures 10 and 11 show the results without and with mobility.

The results for DSDV and VRR are very similar to those obtained with long-lived flows because these protocols do not discover routes on demand. DSDV maintains routes between all pairs of nodes proactively, and VRR maintains routes between virtual ring neighbors that can be used to route between any pair of nodes. DSR and AODV perform badly in this scenario because they discover routes on demand by flooding the network. They cannot amortize the cost of discovery over many data packets because flows are short lived. Figures 10(c) and 11(c) show that the overhead per delivery in DSR and AODV grows quadratically with the number of nodes. Traffic patterns with short-lived flows to random destinations are likely for applications running on DHTs. Therefore, layering existing DHTs on top of reactive protocols is unlikely to work well and proactive protocols perform poorly with mobility.

4.1.7 Stretch

We also measured VRR's stretch, that is, the average ratio between the number of hops traversed by a message and the length of the shortest path between source and destination. We used the same experimental setting except that the packet rate was decreased to 0.1 packets per second to ensure a high delivery ratio for all network sizes. Figure 12(a) shows the stretch for different network sizes, Figure 12(b) shows the stretch distribution for different shortest path lengths between source and destination, and Figure 12(c) shows the distribution of shortest path lengths.

The stretch increases with the network size but it stays below 40% up to 200 nodes. Our rough analysis predicted constant stretch but it ignored the use of routes to one- and two-hop neighbors when forwarding packets. This optimization reduces stretch but its impact decreases with the network size.

VRR preserves locality of communication, which is important to achieve scalability. As shown in Figure 12(b), there is no stretch when the distance between source and destination is less than three hops, and the stretch is relatively independent of the distance in other cases. If VRR did not preserve communication locality, the average number of hops to deliver a message would be independent of the distance between the source and the destination. For example, the stretch when the source and destination are three hops apart would be 2.67 (because the average number of hops to deliver a message is 8.01). Since VRR preserves locality of communication, this stretch is only 1.57.

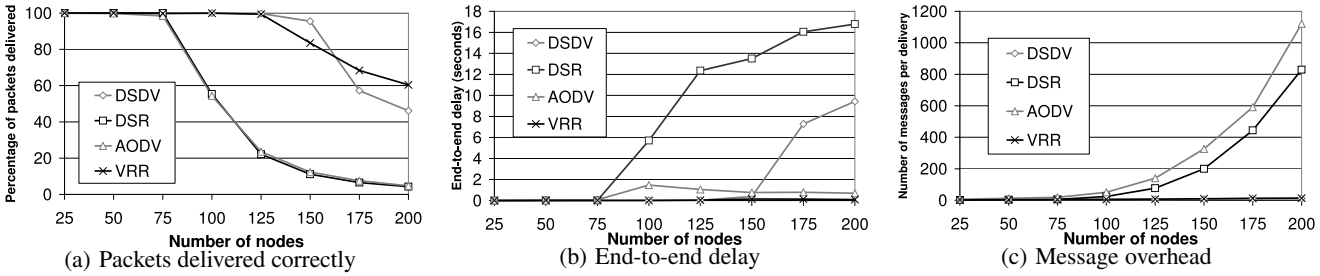
4.2 Sensor network testbed

We compared the performance of VRR and BVR [15] on a sensor network testbed with 67 mica2dot [20] motes distributed over a single floor of the U.C. Berkeley computer science building. BVR is representative of the state of the art in coordinate-based routing and it has an implementation that runs on mica2dot motes.

4.2.1 Protocols

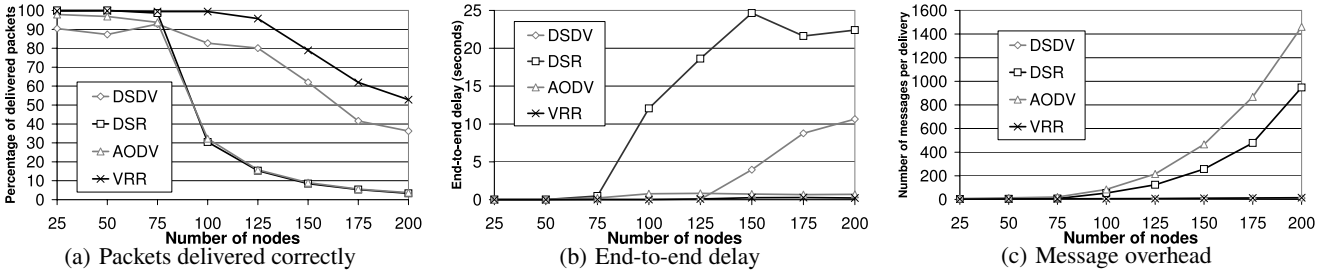
We implemented VRR on mica2dot motes running TinyOS [27]. The implementation was written in nesC [17] and it was configured to use 1-byte node identifiers, a vset size of four ($r = 4$), and a hello period of 10 seconds ($T_h = 10s$). The hello period is large because the data rate of the mica2dot radios is only 19.2Kb/s. This implementation does not support local repair because this optimization provides little benefit in static networks.

We used the BVR implementation described in [15] with a small number of performance improvements [14]. Each BVR node has both a unique identifier and a coordinate that reflects its current location in the network. Coordinates are a vector with the distances in hops to a set of beacons. BVR forwards packets greedily to the neighbor whose coordinate is closest to the destination. When greedy forwarding fails, the packet is sent towards the beacon closest to the destination. If the packet reaches the beacon, it is flooded with scope equal to the distance between the destination and the



(a) Packets delivered correctly (b) End-to-end delay (c) Message overhead

Figure 10: Performance with short-lived CBR flows in the static scenario.



(a) Packets delivered correctly (b) End-to-end delay (c) Message overhead

Figure 11: Performance with short-lived CBR flows in the 20m/s mobility scenario.

beacon. BVR ran with the parameters in [15] and with eight randomly placed beacons. We experimented with different numbers of beacons and chose eight because it provided the best performance.

Both protocols use a link quality estimator based on the algorithm proposed in [41] with parameters tuned using empirical data gathered from the testbed. The network diameter with BVR's estimator is 7. VRR nodes use this estimator to select the members of their physical neighbor set; only links with quality above a threshold are selected.

The mica2dot radios send fixed size packets with a data payload of 28 bytes. VRR adds a header with the identifier of the destination mote. BVR adds a header with the identifier of the destination mote, the coordinate of the destination, and a vector representing the minimum distance observed. With 8-bit identifiers, 4-bit distances, and 8 beacons, BVR's header uses 32% of the payload. VRR's header uses less than 4% of the payload. It is possible to reduce BVR's overhead but this is likely to reduce delivery ratios. The experiments that we ran do not penalize BVR for this overhead: they send packets at the same rate for both protocols and count the fraction of packets delivered.

BVR does not implement a service to map between unique identifiers and the current coordinates of a node. The experiments used the testbed's wired control network to obtain the current coordinates of destination nodes (as in [15]). Running a mapping service would likely decrease the routing performance of BVR and it will be necessary for some applications.

4.2.2 Experiments

The experimental results are averaged over five runs and the motes acting as beacons were chosen randomly each run.

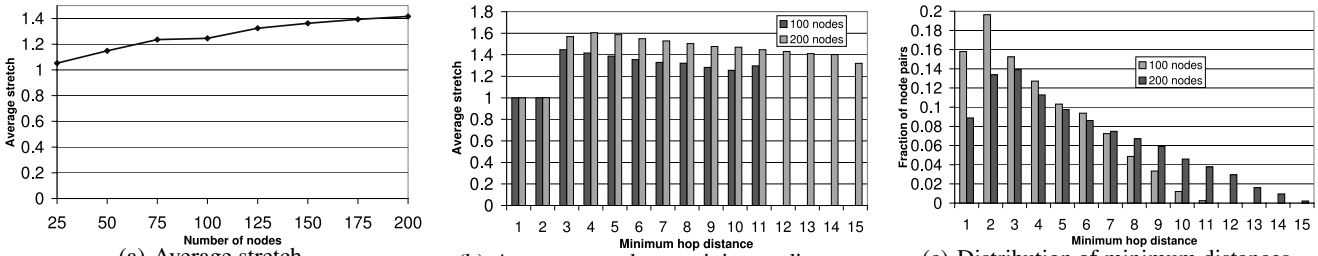
The first experiment measured the fraction of data packets delivered successfully with increasing traffic (as in [15]). There was a 15 minute warmup phase without traffic. For the next five minutes, we selected a new source and destination at random every second and the source sent a data packet to the destination. Afterwards, the send rate was increased by one every 120 seconds up to 8 packets per second. Figure 13(a) shows the delivery ratios for VRR and BVR. BVR achieves nearly perfect delivery ratio with a send rate of one packet per second but the ratio drops as the send rate increases. VRR's delivery ratio is nearly perfect throughout the experiment.

The second experiment measured routing overhead. After a 15 minute warmup phase, we routed 1000 data packets between randomly selected source and destination motes at the rate of one per second. For each data packet that was delivered successfully, we counted the number of data packet transmissions. Figure 13(b) shows a CDF of these transmission counts. The median values for the two systems are similar but the maximum transmission count is 10 for VRR and 71 for BVR. This is because of the overhead incurred by BVR when greedy forwarding fails. Even in static networks without packet losses, greedy forwarding may fail between some pairs of nodes. This problem is not specific to BVR; recovering from greedy forwarding failures is known to introduce overhead in other coordinate-based routing protocols [26].

The final experiment measured the fraction of packets delivered successfully with artificially induced mote failures. After an initial warmup phase of 15 minutes, five random source/destination pairs were chosen every second from the set of live motes and a packet was sent between each pair. After 400 seconds, we killed 10% of the motes at random. We ensured that the 7 motes killed were not BVR beacons because the current BVR implementation does not support recovery from beacon failures. Beacon failures would likely have a more dramatic impact on performance. Figure 13(c) shows the results.

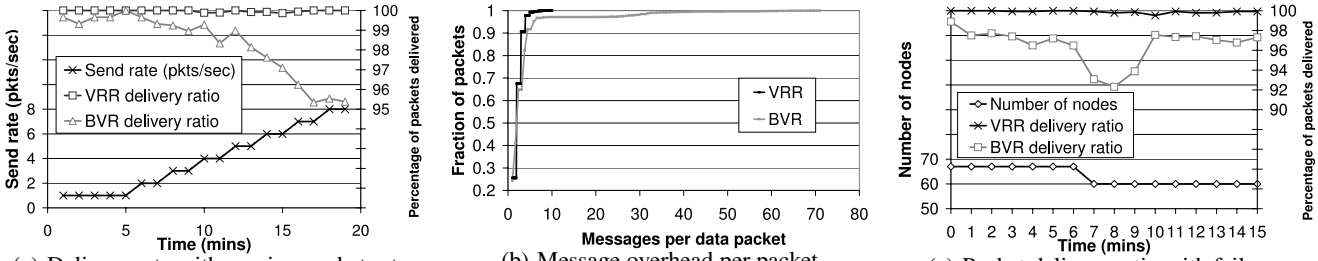
BVR's delivery ratio is lower than VRR's with a send rate of five packets per second (as shown in Figure 13(a)). When the motes fail, BVR's delivery ratio drops but later recovers. The ratio drops because node coordinates change. BVR guarantees delivery when coordinates are stable [15] but may fail to deliver packets when they change. Additionally, coordinate changes can increase routing overhead because of more greedy forwarding failures.

VRR's delivery ratio is high and it is mostly unaffected by the mote failures because it can route around them. VRR exploits path



(a) Average stretch (b) Average stretch per minimum distance (c) Distribution of minimum distances

Figure 12: Stretch with short-lived CBR flows in the static scenario.



(a) Delivery rate with varying packet rate (b) Message overhead per packet (c) Packet delivery ratio with failures

Figure 13: Sensor network testbed results.

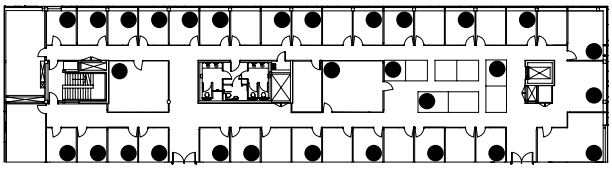


Figure 14: Floor plan of 802.11a PC testbed.

diversity and reroutes data packets dynamically when it encounters a failed mote.

The results show that both VRR and BVR perform well in sensor networks. BVR's performance degrades because of coordinate instability and overhead to recover from failures of greedy routing. VRR's performance appears to be more robust.

4.3 802.11a testbed

The final set of experiments compared the performance of VRR and MR-LQSR [11] on an 802.11a testbed. The testbed consists of 30 PCs running Windows XP that are distributed across a single floor in our office building. As shown in Figure 14, we placed most machines in offices and a small number in cubicles in open-plan areas. Each machine is equipped with a single NetGear WAG 311 wireless network card. The diameter of the network is 4.

4.3.1 Protocols

MR-LQSR [11] is the protocol distributed with the Mesh Connectivity Layer (MCL) toolkit from Microsoft Research [10]. MCL adds a new kernel module that appears as a virtual network adapter to the Windows TCP/IP stack, which allows the use of unmodified IP-based protocols and applications over the wireless mesh.

MR-LQSR represents the state of the art in wireless mesh routing. It modifies DSR to take into account link quality metrics when choosing routes. It uses a metric called Expected Transmission Time (ETT) that is computed using the Expected Transmission Count (ETX) [7] and an estimate of the link bandwidth from packet pair.

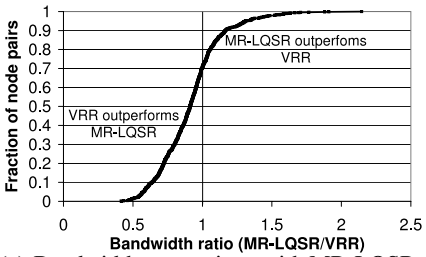
The Windows implementation of VRR replaces MR-LQSR as the routing protocol in the MCL framework. It exploits the ETX and bandwidth estimates computed by MCL to select the machines in physical neighbor sets; only links with ETX and bandwidth values above a threshold are selected. Additionally, VRR includes link quality metrics for each physical neighbor in hello messages. These are used to select between alternate two-hop routes to a node when forwarding a message: if there are multiple two-hop paths to the endpoint with identifier closest to the destination, VRR selects the path with lowest ETT. VRR was configured to use a vset size of four ($r = 4$), and a hello period of two seconds ($T_h = 2s$). Both MR-LQSR and VRR route using 48-bit virtual MAC addresses.

4.3.2 Experiments

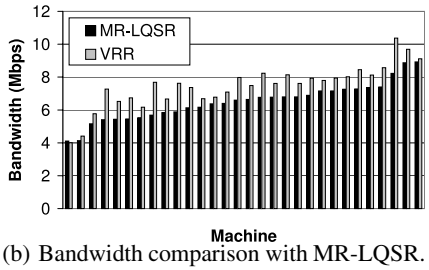
The first experiment compared TCP throughput. We used *ttcp* to transfer 8MB between all pairs of machines. We ran each transfer to completion before starting a new one. The experiment was run three times and VRR identifiers were selected randomly before each run. Figure 15(a) shows a CDF of the ratio between the throughputs of MR-LQSR and VRR for each pair of nodes for all three runs. VRR outperforms MR-LQSR when the ratio is less than one. Figure 15(b) shows the mean throughput between each machine and all other machines averaged over the three runs.

For 70% of the pairs in Figure 15(a), VRR has higher throughput than MR-LQSR. The results in Figure 15(b) also show better throughput for VRR: the average across all machines is 7.5 Mbps for VRR and 6.5 Mbps for MR-LQSR. Interestingly, these throughputs are higher than those provided by 802.11b wireless infrastructures that are still in widespread use.

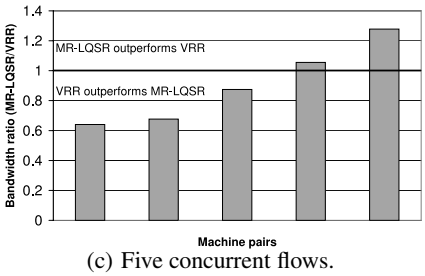
VRR can achieve better throughputs that MR-LQSR because it has lower per-packet overhead. MR-LQSR uses an MTU of only 1,280 bytes to reserve space in the packet for its headers, which include not only the source route but also per-link quality metrics. In contrast, VRR only needs the destination identifier in the packet and, therefore, it can use an MTU of 1,436 bytes.



(a) Bandwidth comparison with MR-LQSR.



(b) Bandwidth comparison with MR-LQSR.



(c) Five concurrent flows.

Figure 15: 802.11a testbed results.

We also measured throughput in a heavily loaded network with five concurrent *ttcp* transfers. We transferred 48MB of data between five random pairs of machines and started all transfers at the same time. We ran the experiment three times between the same pairs of machines. Figure 15(c) shows the ratio between the average throughputs of MR-LQSR and VRR for each pair of nodes. The results show that VRR's throughput is 18% better on average.

The final experiment compared ICMP ping delays. We measured 10 round trip delays between all pairs of machines. Both systems achieved very similar delays. The averages were 3.2ms for MR-LQSR and 3.4ms for VRR. We observed a loss rate below 0.01% for both systems.

4.4 Discussion

Our experimental results show that VRR performs well over a wide range of wireless environments and workloads.

The simulation results show that VRR achieves low delays and good delivery ratios in all experiments. The other protocols perform well in some experiments but poorly in others. It is particularly interesting that VRR can achieve lower delays because it inflates the length of routing paths relative to the shortest paths discovered by the other protocols. It can achieve this because it can route around failures without waiting for routes to be repaired, and because it can repair vset-paths efficiently. The sensor network experiments also show that VRR's performance is more robust than BVR's. Finally, the results from the 802.11a testbed show that VRR performs as well as MR-LQSR, even though the simulation results indicate that this is not the most favorable scenario for VRR.

5. RELATED WORK

There has been a large amount of work on wireless routing protocols. These protocols can be classified into five major types: reactive, proactive, hybrid, hierarchical, and coordinate-based.

Reactive protocols perform route discovery on-demand by flooding the network and they delay packets until the routes are set up. For example, AODV [35], DSR [22] and TORA [32] are reactive protocols. Proactive protocols maintain routes between all pairs of nodes. They flood information across the network whenever the topology changes, but they do not incur delay or overhead to discover routes on demand. DSDV [34], OLSR [6], and WRP [31] are examples of proactive protocols.

In general, proactive protocols work well in static scenarios while reactive protocols work best in mobile scenarios. Hybrid protocols such as ZRP [18] and SHARP [37] achieve good performance across a wider range of scenarios by combining both reactive and proactive components. They divide the network into zones. Nodes maintain routes proactively within their zone by flooding topology changes within the zone. Routes between zones are discovered on demand with an optimized flooding mechanism.

Hierarchical and coordinate-based protocols do not flood the network. For example, LANMAR [33] and L+ [5] are hierarchical protocols, and GPSR [24] and BVR [15] are coordinate-based protocols. They use location-dependent addresses to route. These identifiers can change with mobility and, in some protocols, with congestion and failures [5, 33, 15]. Therefore, these protocols use both a fixed identifier and a location-dependent address for each node and they usually require mechanisms to lookup the location of a node given its fixed identifier [28]. These mechanisms reduce resilience to failures, introduce overhead, and increase complexity.

VRR represents a unique point in the design space. Each VRR node maintains a small number of paths to its vset members proactively. These vset-paths are built and maintained without flooding. VRR is able to forward packets between any pair of nodes using these vset-paths without any route discovery overhead or delay. VRR avoids the problems with changes in location-dependent address because it only uses fixed identifiers.

The design of VRR is inspired by structured overlay routing protocols used in DHTs, for example, [38, 40, 44, 39]. Chord [40] and Pastry [39] both organize nodes in a virtual ring and maintain sets with the closest virtual neighbors of each node. The big difference is that DHTs assume an underlying network routing protocol that provides connectivity between all pairs of nodes. VRR *is* a network routing protocol; it is implemented directly on top of the link layer. Another difference is that VRR does not maintain a finger table like Chord or Pastry; VRR nodes only maintain paths to their virtual neighbors. The fingers are replaced by information about vset-paths that do not end at the node but are routed through it. VRR provides both point-to-point routing and DHT functionality. Many of the current applications built on top of DHTs could be efficiently supported by VRR.

There has been some recent work on providing DHT routing without perfect connectivity between all pairs of overlay nodes. The Unmanaged Internet Protocol (UIP) [16] introduces a routing layer above IP that can route around discontinuities and failures in the Internet. The design of UIP is derived from the Kademlia DHT [29] and is focused on NAT and firewall traversal. FreePastry [19] uses a limited form of source routing to ensure that a node can communicate with its virtual neighbors.

There have been several proposals for combining DHTs with wireless network routing, for example, PeerNet [13], DPSR [21, 36], MADPastry [43] and CrossROAD [8]. PeerNet [13] and MADPastry [43] route using location-dependent addresses, which has the disadvantages we mentioned before. In DPSR [21, 36], each node maintains a finger table similar to Pastry's, but it stores source routes to the nodes pointed to by each finger. DPSR uses flooding to discover the source routes. CrossROAD [8] implements a DHT on top of OLSR, which is a proactive link-state protocol that floods topology changes to all nodes. Since each node knows the identity

of all the other nodes in the network, it can determine locally the node whose identifier is closest to a hash table key and route a message to that node using OLSR. Unlike these systems, VRR does not use flooding or location-dependent addresses.

6. CONCLUSIONS

Virtual Ring Routing is a novel network routing protocol that provides both point-to-point routing and DHT functionality. VRR routes using only fixed location independent identifiers that determine the positions of nodes in a virtual ring. Each node maintains a small number of paths proactively to its neighbors in the virtual ring. These paths can be used to forward messages between any pair of nodes and they can be set up and maintained without flooding.

In this paper, we evaluated VRR in the context of ad hoc wireless networks. We have presented simulation results and results from two implementations running on wireless testbeds. The results demonstrate that VRR provides robust performance across a range of different environments and workloads. We believe that VRR could be used to route in other types of networks, for example, in enterprise networks or even in the Internet.

Acknowledgements

We would like to thank Christian Huitema and Gabriel Montenegro for many useful discussions on VRR.

7. REFERENCES

[1] ns-2 network simulator. `http://www.isi.edu/nsnam/ns/`.

[2] J.-Y. Le Boudec and M. Vojnovic. Perfect simulation and stationarity of a class of mobility models. In *Infocom*, 2005.

[3] J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Mobicom*, October 1998.

[4] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI*, December 2002.

[5] B. Chen and R. Morris. L+: scalable landmark routing and address lookup for multi-hop wireless networks. In *Technical Report 837, MIT LCS*, March 2002.

[6] T. Clausen and P. Jacquet. OLSR RFC3626, October 2003. `http://ietf.org/rfc/rfc3626.txt`.

[7] D. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Mobicom*, 2003.

[8] F. Delmastro. From Pastry to CrossROAD: Cross-layer ring overlay for ad hoc networks. In *PerCom Workshops*, 2005.

[9] J. Douceur. The sybil attack. In *IPTPS*, March 2002.

[10] R. Draves, J. Padhye, and B. Zill. Comparison of routing metrics for static multi-hop wireless networks. In *SIGCOMM*, August 2004.

[11] R. Draves, J. Padhye, and B. Zill. Routing in multi-radio, multi-hop wireless mesh networks. In *Mobicom*, September 2004.

[12] J. Dunagan, N. Harvey, M. Jones, D. Kostic, M. Theimer, and A. Wolman. Fuse: Lightweight guaranteed distributed failure notification. In *OSDI*, December 2004.

[13] J. Eriksson, M. Faloutsos, and S. Krishnamurthy. Peernet: Pushing peer-to-peer down the stack. In *IPTPS*, February 2003.

[14] R. Fonseca. Personal communication.

[15] R. Fonseca, S. Ratnasamy, J. Zhao, C. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable point-to-point in wireless sensornets. In *NSDI*, May 2005.

[16] B. Ford. Unmanaged Internet Protocol: Taming the edge network management crisis. In *HotNets II*, November 2003.

[17] D. Gay, P. Levis, R. vonBehren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI*, June 2003.

[18] Z. J. Haas and M. R. Pearlman. The zone routing protocol (ZRP) for ad hoc networks. July 2002. Internet-draft, draft-ietf-manet-zone-zrp-04.txt.

[19] A. Haeberlen, J. Hoye, A. Mislove, and P. Druschel. Consistent Key Mapping in Structured Overlays. In *Technical Report TR05-456, Rice CS Department*, August 2005.

[20] J. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 2002.

[21] Y. Hu, H. Pucha, and S. Das. Exploiting the synergy between peer-to-peer and mobile ad-hoc networks. In *Hot-OS IX*, May 2003.

[22] D. Johnson and D. Maltz. Dynamic source routing in ad hoc wireless networks. In *Ad Hoc Networking*, 2001.

[23] D.B. Johnson and D.A. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, 353, 1996.

[24] B. Karp and H. Kung. Greedy perimeter stateless routing for wireless networks. In *Mobicom*, August 2000.

[25] P. Key and G. Cope. Distributed Dynamic Routing Schemes. *IEEE Communications Magazine*, October 1990.

[26] Y-J Kim, R. Govindan, B. Karp, and S. Shenker. Geographic routing made practical. In *NSDI*, May 2005.

[27] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *NSDI*, March 2004.

[28] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. A scalable location service for geographic ad-hoc routing. In *Mobicom*, August 2000.

[29] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-peer Information System. In *IPTPS*, 2002.

[30] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host identity protocol (HIP), 2004. draft-moskowitz-hip-08.txt.

[31] S. Murthy and J.J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. In *Mobile Networks and Applications*, 1996.

[32] V. Park and M. Corson. Temporally-ordered routing algorithm (TORA) version 1: Functional specification. July 2001. Internet-draft, draft-ietf-manet-tora-spec-04.txt.

[33] G. Pei, M. Gerla, and X. Hong. LANMAR: Landmark routing for large scale wireless ad hoc networks with group mobility. In *MobiHoc*, 2000.

[34] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Sigcomm*, August 1994.

[35] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications*, February 1999.

[36] H. Pucha, S. M. Das, and Y. C. Hu. Imposed route reuse in ad hoc network routing protocols using structured peer-to-peer overlay routing. *IEEE Transactions on Parallel and Distributed Systems (to appear)*, 2006.

[37] V. Ramasubramanian, Z. Haas, and E. Sirer. SHARP: A hybrid adaptive routing protocol for mobile ad hoc networks. In *Mobihoc*, June 2003.

[38] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Sigcomm*, August 2001.

[39] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, November 2001.

[40] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Sigcomm*, August 2001.

[41] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys*, November 2003.

[42] J. Yoon, M. Liu, and B. Noble. Random waypoint considered harmful. In *Infocom*, 2003.

[43] T. Zahn and J. Schiller. MADPastry: A DHT substrate for practicably sized MANETs. In *ASWN*, June 2005.

[44] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: an infrastructure for fault-resilient wide-area location and routing. In *Technical report UCB//CSD-01-1141, U.C. Berkeley*, April 2001.