Beacon Vector Routing: Scalable Point-to-Point in Wireless Sensornets

Rodrigo Fonseca, Sylvia Ratnasamy, David Culler, Scott Shenker, and Ion Stoica

IRB-TR-04-012

May, 2004

Intel Research Berkeley

Beacon Vector Routing: Scalable Point-to-Point Routing in Wireless Sensornets

Rodrigo Fonseca *Sylvia Ratnasamy* *David Culler* *Scott Shenker* *Ion Stoica*

Abstract

This paper proposes a practical and scalable technique for point-to-point routing in wireless sensornets. This method, called *Beacon Vector Routing* (BVR), assigns coordinates to nodes based on the vector of distances (hop count) to a small set of beacons, and then defines a distance metric on these coordinates. Packets are routed greedily, being forwarded to the next hop that is the closest (according to this beacon vector distance metric) to the destination. This approach is evaluated through both simulation and a prototype implementation on motes.

1 Introduction

Because *ad hoc* networks cannot use careful address allocation to create significant route aggregation, as is used in Internet routing, it has proven difficult to design scalable multihop point-to-point routing for wireless sensornets. There is a vast literature on such algorithms in the MANET space [16], with most proposals using some form of (perhaps partial) flooding to discover individual routes on demand. While these designs perform admirably in many settings, they are not scalable when the traffic pattern involves many source-destination pairs.

One approach that is potentially very attractive is geographic routing [9, 1, 10]. Here, nodes know, and are identified by, their geographic coordinates. Routing is done greedily; at each step in the routing process, nodes pick as next-hop the neighbor that is closest to the destination. When a node has no neighbor that is closer to the destination, GPRS geographic routing enters perimeter mode, where routing is restricted to a planarized subset of the graph and the packet is sent along a "right-hand walk" until it either (1) reaches a node that has a neighbor closer to the destination than the starting point of perimeter mode (in which case it resumes its greedy forwarding), or (2) returns to the node at which it entered perimeter mode (in which case there is no node closer to the destination than that node, and the algorithm stops). While geographic routing is eminently scalable with small routing state and localized discovery, it has two problems. First, the most common planarization algorithms do not work on general graphs.[1] Second, and more seriously, such routing requires that each node must know its geographic coordinates. While there are some sensornet nodes that are equipped with GPS, the most widely used node — the Berkeley mote [5] — is not. Moreover, even when available GPS does not work in certain physical environments and localization algorithms are not sufficiently precise (at least not in all settings), to be used for geographic routing. Finally, even ignoring all the above, greedy geographic may be substantially suboptimal because it does not use real connectivity information and geography is not always in congruence with true network connectivity (*e.g.*, in the face of obstacles or consistent interference) Consequently, there are a number of sensornet deployments for which geographic routing is not a viable option.

There have been two recent attempts to use geographic routing ideas for sensornets without geographic information. The NoGeo scheme [14] creates synthetic coordinates in an iterative fashion, with each node adjusting to the coordinates of their neighbors until an equilibrium is reached. The initialization technique for this scheme requires roughly $O(\sqrt{n})$ nodes to flood the sensornet, and for each of these flooding nodes to store the entire $O(\sqrt{n})$ X $O(\sqrt{n})$ matrix of distances (in hops). This is keeping $O(n)$ state at roughly $O(\sqrt{n})$ nodes, clearly an impractical burden in large sensornets. The GEM scheme [13] is more scalable in steady state, but employs a very intricate recovery process in which, when nodes fail or radio links degrade, a potentially large number of nodes in the system must recompute routing labels so as to maintain GEM's regular topological structure. Thus, this algorithm may not be very robust in practice.

Note that neither of these schemes has been implemented on motes. While there are cases where implementation yields little additional insight, the severe resource restrictions of

[1]This problem may have been solved by some recent unpublished work, but because it is under anonymous submission we can't cite it here.

the lowest, most numerous tier of sensor networks impose daunting complexity restrictions that can only be appreciated through an actual implementation. Moreover, coping with the vicissitudes of real radios, especially in non-laboratory physical spaces, requires a fair degree of robustness. We discuss some of these issues below, but suffice it to say that we expect these issues to render both GEM and NoGeo quite difficult to implement and operate on resource constrained sensor network nodes.

Thus, our goal is to find a scalable point-to-point sensornet routing algorithm that does not rely on geographic coordinate and is both simple and robust. We propose such a scheme, called *Beacon Vector Routing* (BVR). The BVR approach combines the greedy routing notion from geographic routing with the use of landmark beacons from Landmark Routing [18].[2]

BVR uses a small set of beacons (typically 1-2%) that (conceptually) flood the sensornet so that each node learns its distance in network hops from the beacons. A node's vector of distances from the set of beacons serves as its *coordinates*. A packet's destination is described in terms of a distance vector and each sensornet node forwards a packet to its neighbor whose own distance vector is "closest" (according to some metric, which we define later) to the destination distance vector. When this greedy routing can make no more progress, the packet is forwarded towards the beacon closest to the destination. If while heading towards the beacon the packet can resume greedy forwarding it does so; if not, the packet is flooded with a limited scope once it reaches the said beacon, with the necessary scope determined by the distance vector.

The algorithm requires very little in the way of state, overhead, or preconfigured information (such as geographic location of nodes). Routes are based on connectivity and have low stretch over minimum paths. We evaluate this algorithm via simulations and prototype deployments on two mote testbeds. While our results from the testbeds are preliminary, the mote implementation itself provides evidence of the simplicity of the algorithm.

Before developing BVR in earnest, we note that the need for point-to-point routing on sensornets is not as obvious as in traditional computer networks. The first wave of sensornet applications used many-to-one or one-to-many routing (as in directed diffusion [7] and TinyDB [12]). This partly reflects application requirements but is also partly an indication of what routing problems have scalable, practical solutions. Several recent proposals for sensornet algorithms and applications do require point-to-point routing. These include tracking [**?**] and various forms of in-network storage and processing [11, 3, 4, 2, 17, 15]. Without having a scalable point-to-point routing algorithm, we will not even be able to test these ideas. Thus, we see our proposal as an enabler for the next round of exploration of sensornet applications.

[2]While Landmark Routing is superficially similar, in that nodes are located in terms of their distances to beacons, we can't directly apply that approach here because Landmark Routing depends on carefully placed beacons that self-organize themselves into a hierarchical structure. Our beacons are randomly chosen, and we don't require them to establish any structure.

The remainder of this document is organized as follows. Section 2 describes our routing algorithm. We also describe possible implementations of a location service to be used in conjunction with such routing, which is necessary for some applications. In Section 3, we show through high level simulations that the algorithm finds efficient routes, can scale to large networks, and works in the presence of obstacles and in low density networks. Section 5 present results of a protoype of BVR implemented in TinyOS [6] and deployed in two real testbeds of `mica2dot` motes. These deployments are a very effective way of testing the algorithm because it exposes us to the idiosyncrasies of the real radio environment, and the additional challenges the resource constrained platform imposes. Finally, future directions and our conclusions are presented in section 6.

2 The BVR Algorithm

As mentioned earlier, BVR defines a set of coordinates and a distance function to enable scalable greedy forwarding. These coordinates are defined in reference to a set of "beacons" which are a small set of randomly chosen nodes that periodically flood messages to the rest of the network using a fairly standard reverse path tree construction algorithm (we use one similar to that described in [19], but those used by TinyDB[12] or Directed Diffusion[7] would have also sufficed). These periodic messages allow every node to learn its distance, in hops, to each of the beacons.[3]

Let $B_i(q)$ denote the distance in hops from node q to beacon i. Let r denote the total number of beacon nodes. Then, we define a node q's position $\mathcal{P}(q)$ as being the vector of these beacon distances: $\mathcal{P}(q) = \langle B_1(q), B_2(q), \cdots, B_r(q)\rangle$. Note that two different nodes can have the same coordinates, so we will always retain a node identifier to disambiguate nodes in such cases. Nodes must know the positions of their neighbors (in order to make routing decisions), so nodes periodically (locally) broadcast messages announcing their coordinates.

To route, we need a distance function $\delta(p, q)$ on these vectors that measures how good p would be as a next hop to reach a destination q. We choose a distance function (or metric) that (1) emphasizes the components where q is closer to the beacon than p and (2) only considers the k closest beacons to q. The latter is to reduce the number of elements $B_i(q)$ that must be carried in the packet (see below), the former is because moving away from a beacon (to match the destination's coordinate) might be going in the wrong direction, while moving

[3]As described in Section 4, our implementation measures and takes into account the *bidirectional quality* of each link when constructing these trees, so in reality our network distance is the number of hops along a path of acceptable quality links.

Packet fields	Description
$pkt.dst$	the destination's unique identifier
$pkt.\mathcal{P}(dst)$	destination's BVR position
$pkt.\overline{\delta}_{min}$	$\delta_{min}(i)$ seen, $i \in C(k, dst)$

Table 1: BVR packet header fields

Para.	Description
r	total number of beacons
k	$k \subseteq r$; used to define a destination's position
w_i	weights used in calculation of $\delta()$

Table 2: BVR algorithm parameters

towards a beacon (to match the destination's coordinate) is always moving in the right direction. In other words, a beacon that "pulls" routes inwards towards itself (and hence the destination) is a better routing guide than one that "pushes" routes away from itself.

The distance function we use is:

$$\delta(k, p, q) = \sum_{i \in C(k,q)} \omega_i(p, q)|B_i(p) - B_i(q)|, \qquad (1)$$

where $C(k, q)$ is the set of the k beacons closest to q and $\omega_i(p, q) = 10$ if $B_i(p) > B_i(q)$ else $\omega_i(p, q) = 1$. While this functions performs well in our simulations and implementation, there may well be other distance functions that perform equally well.

To route to a destination node d, a packet holds three fields in the header: (1) the destination node's unique identifier, (2) the destination's position $\mathcal{P}(d)$, and (3) $\overline{\delta}_{min}$, a k-position vector where $\delta_{min}(i)$ is the minimum distance that the packet has seen so far using $C(i, d)$, the i closest beacons to d. Table 1 describes these fields. The reason we store the destination's unique identifier in addition to its position is to disambiguate in cases where neighboring nodes have identical positions.

Table 5 summarizes the parameters in our BVR algorithm and Algorithm 1 lists the pseudocode for the forwarding algorithm. Forwarding a message starts with a greedy search for a neighbor that improves the minimum distance we have seen so far. When forwarding the message, the current node (denoted $curr$) chooses among its neighbors the node $next$ that minimizes the distance to the destination. We start using the k closest beacons to the destination, and if there is no improvement, we successively drop beacons from the calculation.

If we do not find a neighbor that improves on $\delta_{min}(i)$ for any i, we have to resort to fallback mode. In fallback mode, a node forwards the packet towards the beacon closest to the destination; i.e., it sends the packet to its parent in the corresponding beacon tree. The parent will forward as usual – first attempting to forward the message greedily and, failing to do so, using fallback mode.[4] Hence the intuition behind fallback mode is that if a node cannot make progress towards the destination itself, it can instead forward towards a node that it knows is close to the destination and towards which it does know how to make progress.

Algorithm 1 BVR forwarding algorithm

```
BVR_FORWARD(node $curr$, packet $P$)
  // first update packet header
  for ($i$ = 1 to $k$) do
    $P.\overline{\delta}_{min}(i) = \min(P.\overline{\delta}_{min}(i), \delta(i, curr, P.dst))$

  // try greedy forwarding first
  for ($i$ = $k$ to 1) do
    $next \leftarrow \text{argmin}_{x \in NBR(curr)}\{\delta(i, x, P.dst)\}$
    if ($\delta(i, next, P.dst) < P.\delta_{min}(i)$ ) then
      unicast $P$ to $next$

  //greedy failed, use fallback mode
  $fallback\_bcn \leftarrow$ closest beacon to $P.dst$
  if ($fallback\_bcn$ != $curr$) then
    unicast $P$ to PARENT($fallback\_bcn$)

  //fallback failed, do scoped flood
  broadcast $P$ with scope $P.\mathcal{P}(dst)(fallback\_bcn)$
```

It is possible however that a packet ultimately reaches the beacon closest to the destination and still cannot make greedy progress. At this point, the root beacon initiates a scoped flood to find the destination. Notice that the required scope of the flood can be precisely determined – the distance in hops from the flooding beacon to the destination is determined from the destination's position in the packet header. While this ensures that packets can always reach their destination, flooding is an inherently expensive operation and hence we want to minimize the frequency with which this step is performed, and also the scope of the flood. Our results show both these numbers to be low.

The algorithm, as described so far, assumes that the originating node knows the coordinates of the intended destination. Depending on the application, it may be necessary for the origination node to first look up the coordinates by name, so we must provide a directory mapping node identities to coordinates. We describe a simple mechanism to achieve this functionality, but we have not focused on this mechanism in this paper. This mechanism uses the beacons as a set of storage nodes. Consistent hashing [8] provides a mapping $\mathcal{H} : nodeid \mapsto beaconid$, from node ids to the set of beacons. As all nodes know all beacons, any node can independently (and consistently) compute this mapping. The location service then consists of two steps: each node k periodically publishes its coordinates to its corresponding beacon

[4]Note that the use of $\overline{\delta}_{min}$ ensures that routing will never loop.

$b_k = \mathcal{H}(k)$. When a node i first wants to route to k, it sends a coordinate request to the beacon b_k. Upon receiving a reply, it then routes to the received coordinates. Further communication between the nodes may skip the lookup phase, if the nodes piggyback their own location information on the packets they send.

3 Simulation Results

To evaluate the BVR algorithm, we use both extensive simulations and experiments on testbeds of real sensor motes. To aid the development of BVR and to better understand its behavior and design tradeoffs we start by evaluating BVR using a high-level simulator that abstracts away many of the vagaries of the underlying wireless medium. While clearly not representative of real radios, these simplifications allow us to explore questions of algorithm behavior over a wide range of network sizes, densities, and obstacles that would not be possible using a real testbed.

In practice however, the characteristics of wireless sensor networks impose a number of challenges on actual system development. For example, the `mica2dot` motes have severe resource constraints – just 4KB of RAM, typical packet payloads of 29 bytes *etc.* – and the wireless medium exhibits changing and imperfect connectivity. Hence, our next round of evaluation is at the actual implementation level. We present the implementation and experimental evaluation of our BVR prototype in Sections 4 and 5 respectively and our simuala-tion results in this section. We start with a description of our high-level simulator:

Simulator We use a packet-level simulator implemented in `C++`. Our simulator makes several simplifying assumptions. First, it models nodes as having a fixed circular radio range; a node can communicate with all and only those nodes that fall within its range. Second, the simulator ignores the capacity of, and congestion in, the network. Finally, the simulator ignores packet losses. While these assumptions are clearly unrealistic, they allow the simulator to scale to tens of thousands of nodes.

In our simulations, we place nodes uniformly at random in a square planar region, and we vary the total number of beacons r, and the number of routing beacons, k. In all our tests, we compare the results of routing over BVR coordinates to greedy geographic routing over the true positions of the nodes.

Our default simulation scenario uses a 3200 node network with nodes uniformly distributed in an area of 200×200 square units. The radio range is 8 units, and on average nodes have approximately 16 neighbors. Unless otherwise stated, a node's neighbors are those nodes that fall within its one hop radius.

3.1 Metrics

In our evaluation, we consider the following performance metrics:

- **(Greedy) success rate:** The fraction of packets that are delivered to the destination without requiring flooding. We stress that the final scoped flooding phase ensures that all packets eventually reach their destination. **This metric merely measures how often the scoped flooding is not required.**
- **Flood scope:** The number of hops it takes to reach the destination in those cases when flooding is invoked.
- **Path stretch:** The ratio of the path length of BVR to the path length of greedy routing using true positions.
- **Node load:** The number of packets forwarded per node.

In each test, we're interested in understanding the overhead required to achieve good performance as measured by the above metrics. There are three main forms of overhead in BVR:

- **Control overhead:** This is the total number of flooding messages generated to compute and maintain node coordinates and is directly dependent on r, the total number of beacons in the system. We measure control overhead in terms of the total number of beacons that flood the network. Ideally, we want to achieve high performance with reasonably low r.
- **Per-packet header overhead:** A destination is defined in terms of its $k(\leq r)$ routing beacons. Because the destination position is carried in the header of every packet for routing purposes, k should be reasonably low.
- **Routing state:** The number of neighbors a node maintains in its routing table.

3.2 Routing Performance vs. Overhead

In this section, we consider the tradeoff between the routing success rate and the flood scope on one hand, and the overhead due to control traffic (r) and per-packet state (k) on the other hand.

We use our default simulation scenario and for each of ten repeated experiments, we randomly choose r beacons from the total set of nodes. We vary r from 10 to 80 each time generating $32,000$ routes between randomly selected pairs of nodes.

Figure 1 plots the routing success rate for an increasing total number of beacons (r) at three different values of k, the number of routing beacons ($k = 5$, 10, and 20) As expected, the success rate increases with both the number of total beacons and the number of routing beacons.

We draw a number of conclusions from these results:

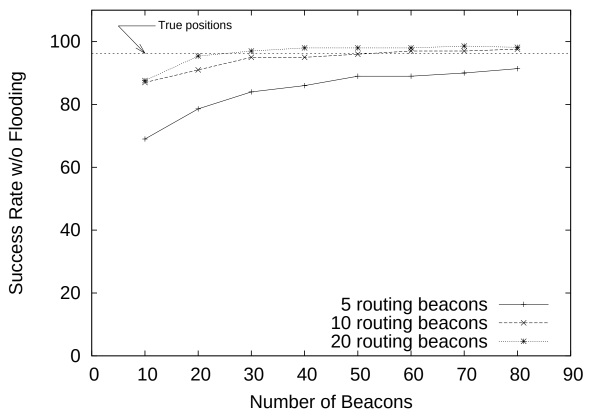

Figure 1: Success rate of routes without flooding in a 3200 node network, for different numbers of total beacons, r, and routing beacons, k.

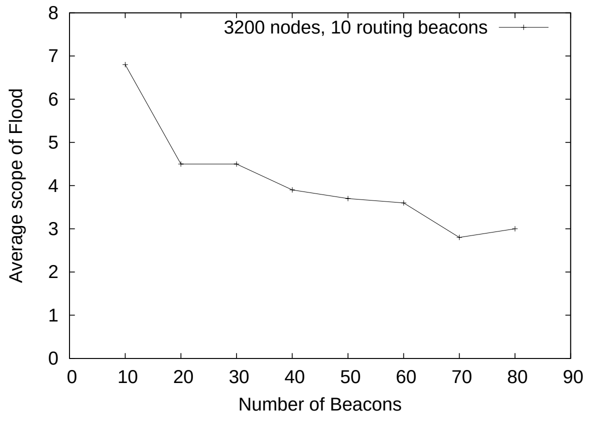

Figure 2: Scope of flood for the network corresponding to that in Figure 1, using k =10 routing beacons.

- We see that with just $k = 10$ routing beacons we can achieve routing performance comparable to that using true positions. The performance improvement in increasing k to 20 is marginal. Hence, from here on, we limit our tests to using $k = 10$ routing beacons as a good compromise between per-packet overhead and performance.

- Using $k = 10$, we see that only between 20 to 30 total beacons (r) is sufficient to match the performance of true positions. At less than 1% of the total number of nodes, this is very reasonable flooding overhead.

- The average path length in these tests was 17.5 hops and the path stretch, i.e., the length of the BVR path over the path length using greedy geographic routing over true positions, is 1.05. In all our tests, we found that the path stretch was always less than 1.1 and hence we don't present path stretch results from here on.

- We also compared the distribution of the routing load over nodes using BVR versus greedy geographic routing over true positions and found that for most nodes, the load is virtually identical though BVR does impose slightly higher load on the nodes in the immediate vicinity of beacons. For example, for the above test using $r = 40$ and $k = 10$, the 90%ile load per node was 48 messages using BVR compared to 37 messages using true positions.

In summary, we see that BVR can roughly match the performance of greedy geographic routing over true positions with a small number of beacons using only its one-hop neighbors.

Figure 2 illustrates the trade-off between the control overhead and the overhead due to scoped flooding of routes that fail, by plotting the scope of flood as a function of the total number of beacons. The scope of flood decreases from 7 hops to 3 hops as the number of beacons increases from 10 to 80. However, note that the improvement is more pronounced when the number of beacons is low. For 30 beacons the flood scope is already less than 4.5, and for 40 beacons the flood scope is 4.

3.3 The Impact of Node Density

In this section, we consider the impact of the node density on the routing success rate. Figure 3 plots the success rate for the original density of 16 nodes per communication range, and for a lower density of 9.8 nodes per communication range. While at high density the performance of both approaches is comparable, we see that at low densities BVR performs much better than greedy geographic routing with true positions. In particular, while the success rate of the greedy routing is about 61%, the success rate of BVR reaches 80% with

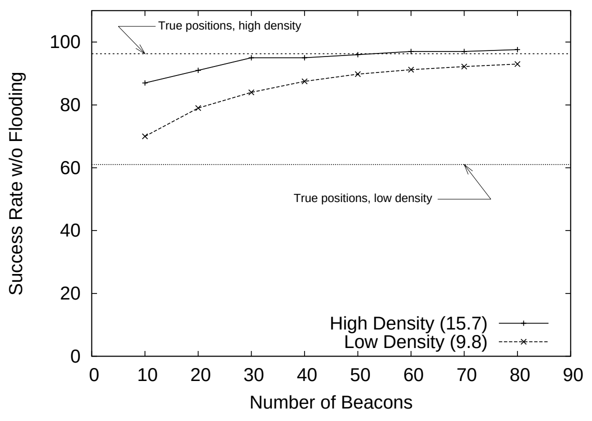

Figure 3: Success rate of routes without flooding, for 3200 node topologies with different densities, for $k = 10$ routing beacons.

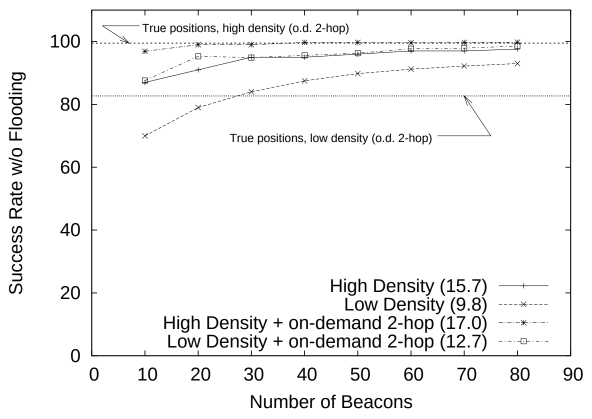

Figure 4: Success rate of routes without flooding, for the same topologies as in figure 3, comparing the use of on demand acquisition of the 2-hop neighborhood information.

30 beacons, and 90% with 40 beacons. Thus, BVR achieves an almost 30% improvement in the success rate compared to greedy routing with true positions. This is because the node coordinates in BVR are derived from the connectivity information, and not from their geographic positions which may be misleading in the presence of the voids that occur at low densities.

These results reflect the inherent tradeoff between the amount of routing state per node and the success rate of greedy routing. At lower densities, each node has fewer immediate neighbors and hence the performance of greedy routing drops. One possibility to improve the performance of our greedy routing is to have nodes maintain state for nodes beyond their one-hop neighborhood. This however increases the overhead and complexity of maintaining routing state. To retain high success rates without greatly (or needlessly) increasing the routing state per node, we propose the use of *on-*

Algorithm	avg ngbrs	max ngbrs	% nodes w/ 2hop	avg success
BVR (hi-dens)	15.7	30.7	0	96.1
BVR+2hop (hi-dens)	17.0	67.5	5	99.7
true postns (hi-dens)	15.7	31.7	0	96.3
true postns+2hop (hi)	15.8	48.0	0.7	99.5
BVR (lo-dens)	9.8	22.1	0	89.2
BVR+2hop (lo-dens)	12.7	50.0	15	97.0
true postns (lo-dens)	9.8	22.8	0	61.0
true postns+2hop (lo)	10.7	36.3	6	82.7

Table 3: State requirements using on-demand two hop neighbor acquisition for BVR and true positions at two different network densities. These state requirements are averaged over 10 runs with $k = 10$ and $r = 50$.

demand two-hop neighbor acquisition. Under this approach, a node starts out using only its immediate (one-hop) neighbors. If it cannot forward a message greedily, it fetches its immediate neighbors' neighbors and adds this two-hop neighbors to its routing table.[5] The intuition behind this approach is that the number of local minima in a graph is far smaller than the total number of nodes. Thus, the on-demand approach to augmenting neighbor state allows only those nodes that require the additional state to incur the overhead of maintaining this state.

To evaluate the effectiveness of using on-demand two-hop neighbor acquisition, we repeat the experiments in Figure 3 using this approach. The results are plotted in Figure 4. Not surprisingly, this approach greatly improves the routing success rate. With only 20 beacons, the success rate of BVR exceeds 99% for the high density network, and 96% for the low density network. Table 3 shows the average and worst case increase in the per-node routing state for both BVR and true positions. Using BVR, at high density, only 5% of nodes fetch their two-hop neighbors while 15% of nodes do so at the lower densities. This confirms our intuition that acquiring two-hop neighbors on demand represents a big win at a fairly low cost.

3.4 Scaling the Network Size

In this section, we ask the following question: how many beacons are needed to achieve a target success rate as the network size increases? To answer this question, we set the target of the routing success rate at 95%. Figure 5 plots the number of beacons required to achieve this target for both BVR using a one-hop neighborhood, and BVR using on-demand two-hop neighbor acquisition. In both cases the number of routing beacons is 10.

There are two points worth noting. First, the number of beacons for the on-demand two-hop neighborhood remains

[5]Note that greedy routing can fail even over this two-hop neighborhood in which case the forwarding node will resort to forwarding using fallback mode as before.

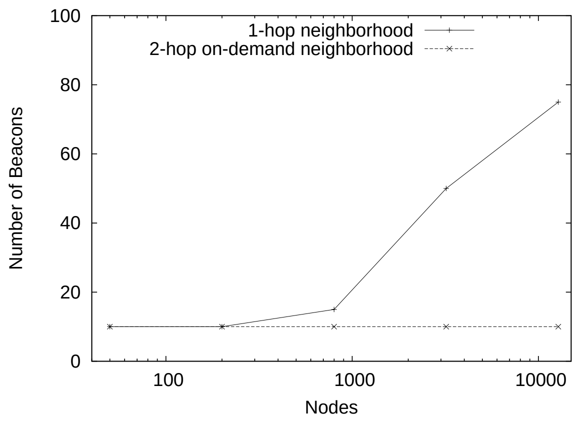

Figure 5: Number of beacons required to achieve less than 5% of scoped floods, with $k = 10$ routing beacons.

constant at 10 as the network size increases from 50 to $12,800$ nodes. Second, while the number of beacons in the case of BVR with one-hop neighborhood increases as the network size increases, this number is still very small. When the network is greater than 800 nodes, the number of beacons for the one-hop neighborhood never exceeds 2%.

These results show that routing rarely needs to resort to flooding, and when it does the scope of the flood is small. Moreover, the number of beacons required to achieve low flooding rates grows slowly (or not at all!) with the size of the system.

3.5 Performance under obstacles

In this section, we study the BVR performance in the presence of obstacles. We model obstacles as horizontal or vertical "walls" with lengths of 10 or 20 units. For comparison, recall that the radio range of a node is 8 units.

Table 4 shows the success rates of BVR routing over a one-hop neighborhood for different numbers of obstacles. For each entry, we also show, in parentheses, the success rate of greedy routing using true positions. Surprisingly, as the number of obstacles and/or their length increases, the decrease in success rate using BVR is *not* significant. In the worst case the success rate drops only from 96% to 91%. For comparison, the sucess rate of greedy routing with true positions drops from 98% to 43%!

As in the case of the lower density network scenario, the main reason BVR achieves much better results is because the node coordinates in BVR reflect their connectivity instead of their true positions. Two nodes on opposite sides of a wall may be very close in true geographic space even though they cannot communicate. In contrast, the same nodes might be quite far apart in the virtual coordinate space constructed by BVR.

Length of Osbstacles	Number of Obstacles			
	0	10	20	50
10	0.96 (0.98)	0.96 (0.91)	0.95 (0.87)	0.95 (0.79)
20	0.96 (0.98)	0.95 (0.84)	0.94 (0.70)	0.91 (0.43)

Table 4: Comparing BVR with greedy forwarding over true positions in the presence of obstacles

4 BVR Implementation

This section describes our prototype implementation of BVR in TinyOS [6] for the `mica2dot` motes. The resource constraints of the mote hardware and the vagaries of the wireless medium lead to a number of practical difficulties not addressed in our discussion so far. In particular, the following are four key issues that must be addressed in a real implementation:

- **Link estimation**: In a wireless medium, the notion of an individual link is itself ill-defined as the quality of communication varies dramatically across nodes, distance and time. Link estimation is used to characterize a link as the probability of successful communication rather than a simple binary on/off relation.
- **Link/neighbor selection**: The limited memory in the mote hardware prevents a node from holding state for all its links. Link selection determines the set of neighbors in a node's routing table.
- **Distance estimation**: Recall that our BVR algorithm defines a node's coordinates as its distance in hops to a set of beacons. We describe how we define the hop distance from a node to a beacon when individual links are themselves defined in terms of a quality estimate.
- **Route selection**: This addresses how a node forwards packets in the face of lossy links.

Each of the above is a research problem in itself (see [19, 20] for a detailed exploration of some of these issues); while our implementation makes what we believe are sound choices for each, a comprehensive exploration of the design space for each individual component is beyond the scope of this paper. In what follows, we describe our solutions to each of the above problems and present the results of our system evaluation in the following section.

For simplicity, our prototype sets the number of routing beacons equal to the total number of beacons ($k = r$) and does not implement the successive dropping of beacons in computing distances for greedy forwarding (*i.e.*, a node that cannot make greedy progress using all available beacons switches directly to fallback mode). Finally, the selection of beacons is preconfigured and we make no attempt to replace failed beacons. We also do not implement the on-demand neighbor acquisition described in the previous section. If anything, these simplifications can only degrade performance relative to our earlier simulation results.

4.1 Link Estimation

Estimating the qualities of the links to and from a node is critical to the implementation of BVR as this affects the estimated distance from beacons as well as routing decisions. For example, consider a node that on occasion hears a message directly from a beacon over a low quality link. If, based on these sporadic receptions, the node were to set its distance from the beacon to be one hop then that would have the undesired effect of drawing in traffic over the low quality link.

Our link estimator is based on that described by Woo *et al* [19]. Nodes perform passive estimation of link qualities by snooping packets on the wireless medium. This is possible due to the broadcast nature of the radio. All packets are assigned sequence numbers from which a node can determine the number of packets transmitted by and received from a given neighbor node. A node estimates the quality of a given link as an exponentially weighted moving average of that link's quality over successive time windows. The link's quality for a given time window is the fraction of transmitted packets received from the corresponding source in that window. A link over which no packets were received in a given time window is assigned a link quality of zero for that window and a link that has a quality of zero for 5 successive time windows is removed from the link table. Our implementation uses a time window of 120 seconds and an exponential average with a smoothing constant of 40%.

The above estimates the quality of an *incoming* link at a node. In addition, every node periodically transmits its current list of incoming links and their respective qualities. This allows nodes to learn the quality of links *from* them which is important for selecting good links along which to route messages. This is particularly important as previous work has shown that mote radio links can be highly asymmetric [19]).

The assumption underlying the above link estimation is that nodes transmit at a certain minimum rate. In BVR, nodes periodically broadcast "hello" messages announcing their coordinates to their immediate neighbors. This is the minimum control traffic required for routing and also serves the purpose of ensuring sufficient traffic for link estimation. To avoid synchronization effects, the interval between successive position broadcasts is jittered around an expected interval value. Our implementation uses an expected interval time of 40 seconds with a jitter of $\pm$ 20 seconds.

4.2 Neighbor Selection

As described above, a node maintains state for each link whose quality it must estimate. Motes however have very limited memory and hence a node may not be able to (or may not want to devote the memory needed to) hold state for all the nodes it might hear from. Hence on the one hand we want a node to hold state for its highest quality links but on the other hand the node does not have the memory resources necessary to estimate the quality of all its links.

To solve this problem we use a scheme that guarantees that a node's link table will eventually converge to storing the highest quality links. Let n denote the total number of slots available for link information. Of these, we let s slots be *testing* slots used to hold new neighbors. When a new neighbor is inserted into a test slot, it cannot be replaced until a *probation* period has passed. We set the probation period to be such that the link estimator would have converged to within 10% of the stable quality of the link. When a node hears from a new neighbor, it searches its testing slots for the lowest quality neighbor that is past probation. If such a neighbor exists, it is evicted from the link table and replaced by the new neighbor. When a neighbor in a testing slot has completed its probation period, we check to see if its link quality is better than the lowest quality node in the $n - s$ non-testing slots and if so, we swap the two. This replacement scheme guarantees that a node eventually holds the $n - s$ highest quality neighbors. In our prototype, we use $n = 12$ and $s = 1$.

4.3 Distance Estimation

Every node in BVR maintains two key pieces of information: (1) its distance in hops to the root beacons and (2) the positions of the node's immediate neighbors

Our prototype implements these using two distinct flows of control packets.[6] To set up the distances to the root beacons, each root periodically floods a message to the entire network. To avoid synchronization, the interval between two floods initiated by a root beacon is jittered. Our flooding protocol exhibits two important properties: first, we guarantee that every node will send each message once, and second, the protocol constructs trees that optimize the quality of the reverse paths to the root.

A beacon flood is identified by the beacon's identifier and a sequence number. For each beacon, a node maintains the sequence number of the last flood from that beacon and its parent and hopcount to that beacon. Upon receiving a beacon flood message, the node determines if it came from that beacon's current parent, or from a 'better' parent. In the latter case, the node updates its parent, and in either case, forwards the message on if it has a new sequence number. To avoid collisions, each node waits for a random interval before forwarding a message.

[6]An alternate option would have been to use the periodic local neighbor exchanges to also infer distances to beacons in the style of distance-vector routing algorithms. While this would use less control traffic, it requires implementing more complicated loop avoidance techniques such as split horizon, hold down timers, counting-to-infinity and so on. We opted for the simpler approach in our prototype but this is certainly a tradeoff that merits further exploration.

We define a 'better' parent as one that has a *lower expected path loss rate to the root*. The path loss rate is obtained by multiplying the reverse path probability from the parent up to the root with the node's estimate of the reverse link quality from itself to the parent. The goal of such estimates is to avoid selecting long, low quality links that result in low hopcounts to the root, albeit of very low quality. Thus, a node's path to the root is that with the lowest expected loss rate and we set the node's distance to the root to be the number of hops along that path (computed as usual by adding one to the parent's hopcount).

4.4 Route Selection

Our BVR prototype uses two optimizations to improve routing reliability in the face of lossy links. First, it uses link level acknowledgements of all unicast packet transmissions and retries every transmission up to five times. Second, if a transmission fails (despite the multiple retries), the node will successively try to forward a packet to *any* neighbor that makes progress to the packet destination (where progress is in terms of the distance metric defined in Section 2). The node will try each of these next hop in the decreasing order of their outgoing link quality. Only when it has exhausted all possible next hop options will the forwarding node revert to fallback mode.

5 Prototype Evaluation

This section presents the results of our experiments with the BVR prototype deployed over two testbeds. The first consists of 23 `mica2dot` motes [**?**] in an indoor office environment (the Intel Berkeley Lab.) of approximately 20x50m while the second is a testbed of about 42 `mica2dot` motes deployed across multiple student offices on a single floor of the computer science building at UCB. In both testbeds, motes are connected to an Ethernet backchannel that we use for logging and driving the experiments. These testbeds are of moderate scale – the Intel-Lab testbed has a network diameter of four hops and average routing pathlength of 2.0 hops,[7] while the UCB testbed has a diameter of approximately 8 hops and average pathlength of 3.22 hops – and hence do not truly stress BVR's scalability. Nonetheless, these deployments are an invaluable (if not the only!) means by which to test our algorithms using real radios with the non-uniform and dynamic link characteristics that cannot be easily captured in simulation.

On both testbeds, we preconfigure four corner nodes to act as beacons and set the parameters of the algorithm according to the values in Table 5. We selected these parameters to achieve what seemed like a good tradeoff between maintaining freshness of the routing state, and the amount of control traffic generated. For example, using the above parameters on the Intel-Lab testbed each mote generates approximately 0.1 messages per second.[8] For a channel capacity of approximately 30 messages per second (observed), and an average of 7 neighbors, this control traffic occupies only about 2% of the channel.

[7]The network diameter is computed as the 99th percentile of measured path lengths.

Link Estimator	
Size of Table	12
Expiration	5 succ. windows
Replacement	1 *testing* slot
Reverse Link Info T	70s (jittered)
Update Link T	120s (fixed)
Exponential Average	smoothing constant 40%
BVState	
Root Beacon T	60s $\pm$ 30s (uniform)
Position Broadcast T	40s $\pm$ 20s (uniform)

Table 5: Parameters used in the experiments on both the Intel-Lab and UCB testbeds

Our experiments consist of a setup phase of several minutes to allow the link estimations to converge, beacon trees to be constructed and nodes to discover their neighbors' positions. After this setup phase, we issue route commands from a PC to individual motes over the ethernet backchannel.

In the reminder of this section we evaluate three main aspects of the BVR design. First, we validate that a node selects indeed high quality neighbors. This is important because the performance of BVR depends heavily on the link quality. Second, we evaluate the BVR routing with respect to two metrics: success rate, and per-node load distribution. We find that the routing success rate is close to 100% when the network load is low, and it degrades gracefully as the load increases. Finally, we evaluate the stability of node coordinates, and we find that these coordinates are surprisingly stable. This result open the possibility of implementing higher level abstractions, such as data-centric storage, on top of BVR.

5.1 Link Estimation

The link estimation results presented here were obtained from an almost two hour long run of BVR on the Intel-Lab testbed. For this run, we did not impose any actual routing load on the network and hence the only traffic in the network is the regular control traffic (*i.e.*, beacon floods, neighbor exchanges) generated by BVR itself.

Based on the packets logged at each mote, we record the quality of every link over which even a single packet was received. Figure 7 compares these measured link qualities to those of the subset of links selected by motes in their routing

[8]There are 4 root beacons which means a node will forward on average 1 root message every $(60/4)$ seconds. In addition, each node will (on average) transmit 1 position message every 40s, and 1 reverse link update every 240s. Hence the expected combined rate is about $\frac{23}{240}$ messages per second.

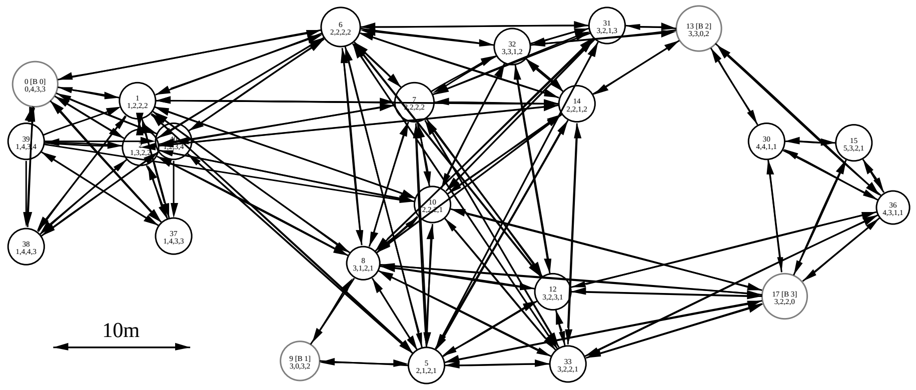
Figure 6: Neighborhood graph as determined by the neighbor tables of motes logged after 30 minutes of run time with the beacons and coordinate broadcasts. Each node is shown with its nodeID and its coordinates with respect to the four beacons used. The beacons are shown in gray. The positions of the nodes are to scale.

tables. We see that the fraction of neighbor links selected in each range of quality increases with the quality and hence the results attest that nodes choose links with comparatively good qualities to be part of their coordinate tables. We notice however, that some high quality links were not selected by any motes even though lower quality ones were. Closer examination revealed two primary causes for this. The first is due to the limit of 12 neighbors per mote because of which well connected motes might ignore even high quality links. The second is due to link asymmetry: if a link's quality is good in one direction but very poor in the other then a node might not receive the reverse link quality information (used to select neighbors) often enough.

Figure 6 shows a snapshot of the network connectivity as determined by the neighbor tables at each mote about 30 minutes into the test. Note that mote positions in the figure represent their actual layout on the testbed and the figure is drawn to scale. We see that network connectivity is frequently not congruent with physical distance. For example, mote 31 and 14 are not neighbors but 31 and 5 are. Likewise motes 10 and 12 are not connected though 10 and 17 are. We also note the existence of short but asymmetric links as between 39 and 1.

Finally, we found that the link qualities estimated by the motes, at least in this environment, did not vary as widely as anticipated; Figure 8 shows the link qualities from a (fairly representative) sample mote as estimated by its neighbors. We note that the link estimator is more stable when links have either very good or very poor quality, and this is the main distinction we need to make.

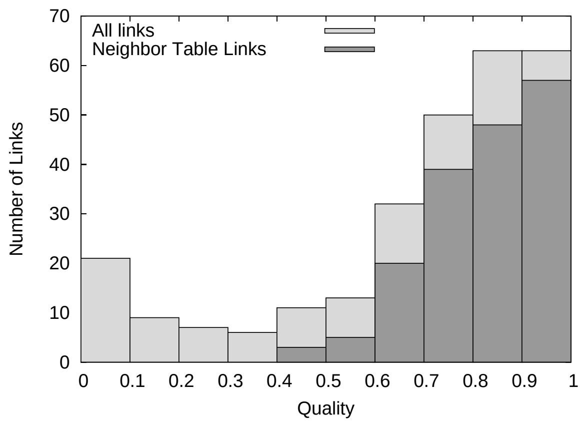

Figure 7: Histograms comparing the measured link qualities of all links with at least one message transmission to the link qualities of the subset of these links chosen by the motes to be part of their routing tables. Notice how the latter are proportionately better quality.

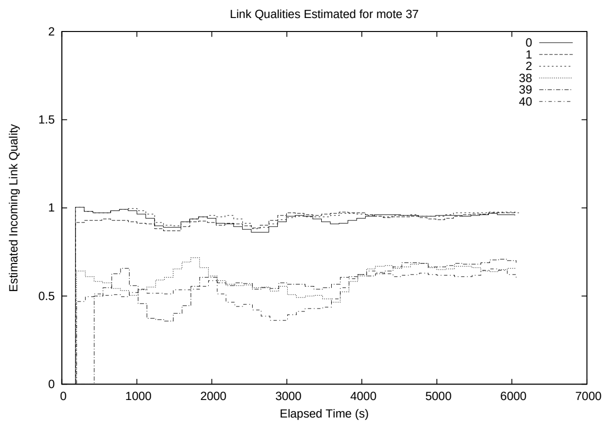

Figure 8: Qualities of the outgoing links from mote 37, as reported by its neighbors.

5.2 BVR

The experimental setup for the results in this section is as described above with the only addition that, after the setup phase, each mote will periodically attempt to route to a randomly selected destination mote. The rate at which each mote

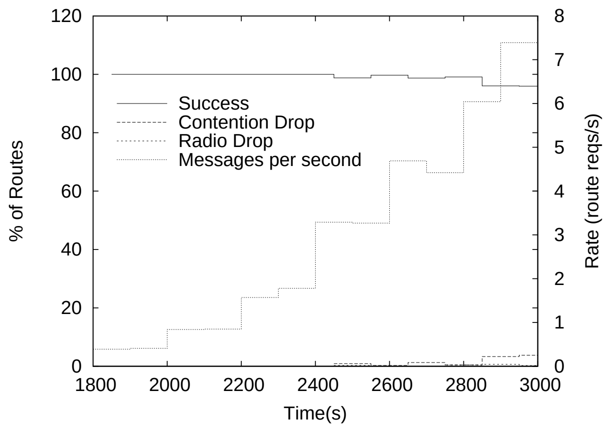

Figure 9: *Intel-Lab* Results of routing tests as we increase the routing load. Rate on the right hand Y axis is the number of route requests issued per node per second.

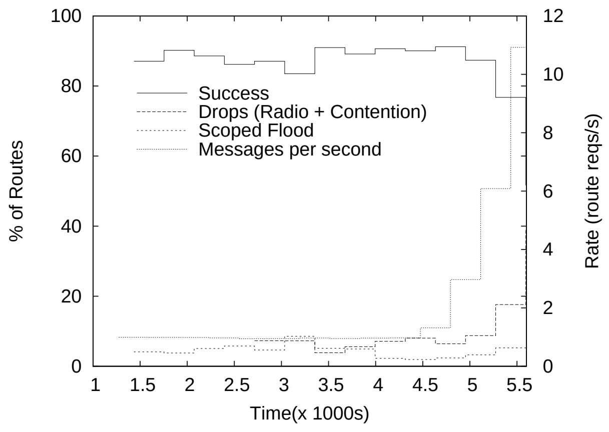

Figure 10: *UCB* Results of routing tests as we increase the routing load. Rate on the right hand Y axis is the number of route requests issued per node per second.

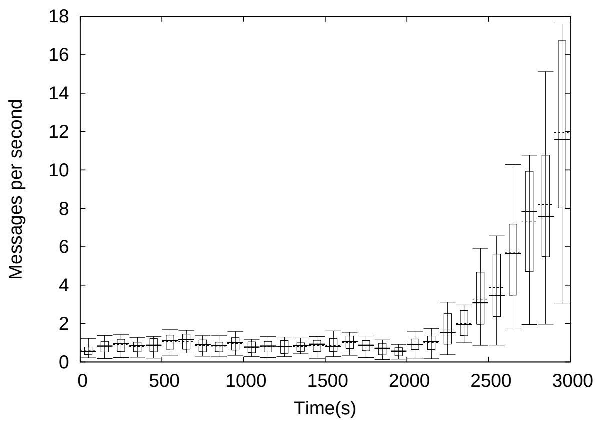

Figure 11: *Intel-Lab* Load distribution in number of total messages received per node per second, for the routing test on the Intel-Lab testbed.

initiates these routes is a test parameter and we present results from runs on both testbeds.

Our first set of results explore BVR routing performance in terms of successful greedy packet delivery and how this scales with increasing routing load. While our simulation results indicated that BVR rarely requires flooding to discover routes, those tests did not model realistic radios nor capture the effect of a finite (and small) channel capacity. As in the simulations, we measure success rate as the fraction of routes that arrived at the target destination *without* requiring scoped flooding. Figure 9 shows the results for the Intel-Lab testbed. Our test starts with each mote issuing 1 route request every 2 seconds (0.5 routes/second) and increases this rate every 200 seconds until we reach a rate of approximately 7 routes/second per mote. Figure 9 plots the success rate over time along with the percentage of routes that failed due to radio or contention drops (where contention drops are packets that were dropped due to a lack of sending or receiving buffers along the internal send/receive path in the stack and radio drops are transmissions that failed despite the multiple retries; *i.e.*, the message was repeatedly sent out over the channel but no acknowledgements were received). The graph also plots the per-mote route request rate over time with the scale on the right hand Y axis. We see that BVR achieves a success rate of 100% until a load of about 3 requests/second at which point we start seeing a very small number of contention drops. Only when the rate gets to approximately 6 requests/second do we observe radio drops. In this testbed, no route attempts required flooding, they either arrived at the destination or suffered drops.

Figure 10 shows the results from similar tests on the UCB testbed. Here the routing load is held low at 1 request/second until about 4,500 seconds into the experiment and then repeatedly increased up to a maximum of 10 requests/second per mote. Over the 3124 routes initiated during the period upto 4500 seconds, an average of 88.4% of routes reached their target destination without requiring any flooding and 6.5% were dropped (about 6.46% of these were radio drops, the remainder due to buffer contention). 4.57% of routes terminated at beacons thus requiring scoped floods; the average scope of the required flood was 2.6. Finally, we found that a very small percentage (approx. 0.5%) of routes were “stuck” at nodes along the way. Examination of the logs revealed that this was typically because all the potential next hop links at the forwarding node were of very poor quality. In fact, examination of the link estimation logs indicate that the average link quality on the UCB testbed is significantly lower than the Intel-Lab testbed. This is also indicated by the higher number of radio drops. The UCB testbed is a relatively new deployment and further exploration is required to determine the extent to which careful tuning of our link estimation algorithms might further improve performance.[9]

Finally, Figure 11 examines the load distribution load in the network, in terms of the number of messages received per second for the Intel-Lab testbed. For each 100 second interval, the boxplot shows the mean (dashed line), and the 5th, 25th, 50th, 75th, and 95th percentiles of the load distribution across all nodes. Our first observation is that when BVR control traffic is the only traffic in the network (during the setup phase, upto again 1800 seconds), the load is small and the distribution not too skewed. After this point the load grows with the increase in routing traffic as does the skew in the load. Notice (from Figure 9) that BVR sustained a routing success rate of over 95% even as 25% of the nodes were receiving as many as 17 messages per second.

In summary, the above experiments indicate that our BVR implementation works correctly in a real deployment, and, can sustain a significant workload of routing messages.

5.3 Coordinate Stability

Our results so far have shown that BVR generates good coordinates in that they correctly guide routes towards a target destination.

Because some applications require the location service (as described in Section 2) to route to node identifiers, it is important that the coordinates not change too frequently or by too much. Otherwise the traffic generated to maintain the location service reasonably up-to-date would be high and all communication would have to incur the overhead of first looking up the destination's current coordinates (because caching the results of an earlier lookup would not be effective).[10]

Hence, we explore two questions: (1) how frequently do a node's coordinates change? and (2) when change occurs, what is the magnitude of change? Accordingly, Figures 12 and 13 plot the distribution and histogram of the number of individual coordinate changes[11] experienced per mote over the entire duration of our routing tests for the Intel-Lab and UCB testbeds respectively. We see that changes are very infrequent. For example, on the UCB testbed, 50% of the motes saw no change and no individual node experienced more than 3 changes over a test period of over an hour.

Corresponding to the above, Figures 14 and 15 plot the distribution and histogram of the magnitude of individual coordinate changes over all coordinate changes. We see that change, when it occurs, is small.

[9]Both these testbeds are currently being expanded; a final version of this paper will report on results from testbeds with over 100 motes.

[10]Note that a node whose coordinates change frequently, but not by much, would be less problematic as the stale coordinates will still follow the generally right direction and might well arrive at the right location.

[11]If a single node experiences a change in its distance to (say) 3 beacons, we count that as three distinct changes.

6 Conclusions and Future Work

Beacon Vector Routing is a new approach to achieving scalable point-to-point routing in wireless sensornets. Its main advantages are its simplicity, making it easy to implement on resource constrained nodes like motes, and resilience, in that we build no large-scale structures. In fact, the periodic flooding from the beacons means that no matter what failures have occurred, the entire state can be rebuilt after one refresh interval. Our simulation results show that BVR achieves good performance in a wide range of settings, at times significantly exceeding that of geographic routing. Our implementation results suggest that BVR can withstand a testbed environment and thus might be suitable used for real deployments.

However, we are at the very early stages of our investigation. We need to understand much more about how BVR's performance is linked to radio stability, and how BVR responds to large-scale failure or very ephemeral nodes (particularly ephemeral beacons). Most importantly, we have not yet implemented any applications on top of BVR, so we don't yet know if it provides a suitably stable substrate on which to build. All of these items represent future work, to which we now turn.

References

[1] Prosenjit Bose, Pat Morin, Ivan Stojmenovic, and Jorge Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7(6):609–616, 2001.

[2] Deepak Ganesan, Deborah Estrin, and John Heidemann. DIMENSIONS: Why do we need a new data handling architecture for sensor networks? In *Proceedings of the ACM Workshop on Hot Topics in Networks*, pages 143–148, Princeton, NJ, USA, October 2002. ACM.

[3] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heiemann. An evaluation of multi-resolution storage for sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 63–75. ACM Press, 2003.

[4] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. Difs: A distributed index for features in sensor networks. In *Proceedings of 1st IEEE International Workshop on Sensor Network Protocols and Applications*, May 2003.

[5] Jason Hill and David Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture

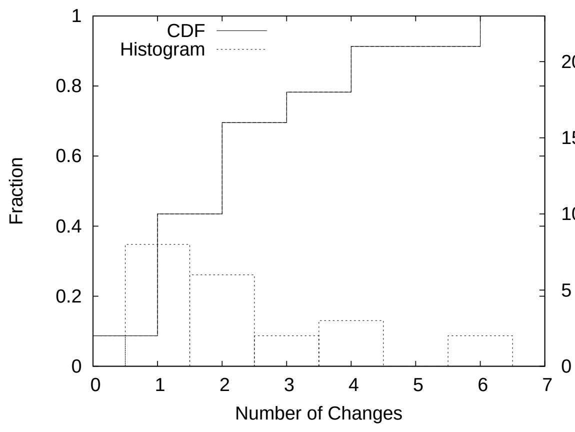

Figure 12: *Intel-Lab* Distribution and histogram of the number of individual coordinate changes experienced per mote over the entire duration of the test. The scale for the histogram is shown on the right hand Y axis.

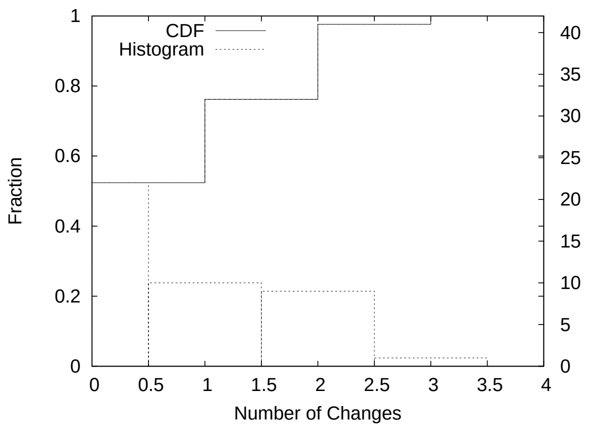

Figure 13: *UCB* Distribution and histogram of the number of individual coordinate changes experienced per mote over the entire duration of the test. The scale for the histogram is shown on the right hand Y axis.

directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, ASPLOS'00*, pages 93–104. ACM Press, 2000.

[7] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67. ACM Press, 2000.

[8] D. R. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. 29th ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[9] Brad 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*, pages 243–254. ACM Press, 2000.

[10] F. Kuhn, R. Wattenhofer, Y. Zhang, , and A. Zollinger. Geometric ad-hoc routing: Of theory and practice. In *22nd ACM Int. Symposium on the Principles of Distributed Computing (PODC)*, 2003.

[11] Xin Li, Young Jin Kim, Ramesh Govindan, and Wei Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 63–75. ACM Press, 2003.

[12] Samuel Madden. *The Design and Evaluation of a Query Processing Architecture for Sensor Networks*. PhD thesis, UC Berkeley, 2003.

[13] James Newsome and Dawn Song. Gem: Graph embedding for routing and data-centric storage in sensor networks without geographic information. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 76–88. ACM Press, 2003.

[14] Ananth Rao, Christos Papadimitriou, Scott Shenker, and Ion Stoica. Geographic routing without location information. In *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking*, pages 96–108. ACM Press, 2003.

[15] Sylvia Ratnasamy, Brad Karp, Scott Shenker, Deborah Estrin, Ramesh Govindan, Li Yin, and Fang Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mob. Netw. Appl.*, 8(4):427–442, 2003.

[16] E. M. Royer and C-K. Toh. A review of current routing protocols for ad-hoc mobile wireless networks. *IEEE Personal Communications*, 6:46–55, April 1999.

[17] Scott Shenker, Sylvia Ratnasamy, Brad Karp, Ramesh Govindan, and Deborah Estrin. Data-centric storage in sensornets. *SIGCOMM Comput. Commun. Rev.*, 33(1):137–142, 2003.

[18] P. F. Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. In *Symposium proceedings on Communications architectures and protocols*, pages 35–42. ACM Press, 1988.

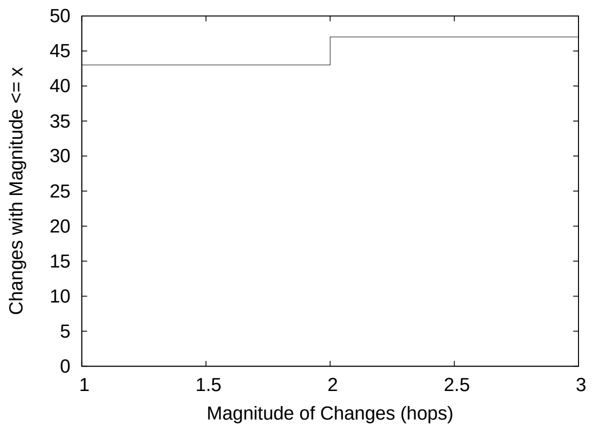

Figure 14: *Intel-Lab* Distribution and histogram of the magnitude of individual coordinate changes over all coordinate changes seen across all nodes over the entire duration of the test (ignores first change due to initialization).

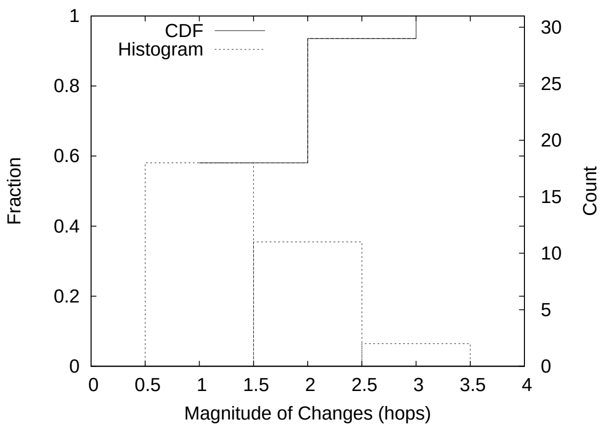

Figure 15: *UCB* Distribution (left Y axis) and histogram (right Y axis) of the magnitude of individual coordinate changes over all coordinate changes seen across all nodes over the entire duration of the test (ignores first change due to initialization).

[19] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.

[20] Jerry Zhao and Ramesh Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 1–13. ACM Press, 2003.