Octopus: A Fault-Tolerant and Efficient Ad-hoc Routing Protocol*

Roie Melamed
CS Department, Technion

Idit Keidar
EE Department, Technion

Yoav Barel
EE Department, Technion

Abstract

Mobile ad-hoc networks (MANETs) are failure-prone environments; it is common for mobile wireless nodes to intermittently disconnect from the network, e.g., due to signal blockage. This paper focuses on withstanding such failures in large MANETs: we present Octopus, a fault-tolerant and efficient position-based routing protocol. Fault-tolerance is achieved by employing redundancy, i.e., storing the location of each node at many other nodes, and by keeping frequently refreshed soft state. At the same time, Octopus achieves a low location update overhead by employing a novel aggregation technique, whereby a single packet updates the location of many nodes at many other nodes. Octopus is highly scalable: for a fixed node density, the number of location update packets sent does not grow with the network size. And when the density increases, the overhead drops. Thorough empirical evaluation using the ns2 simulator with up to 675 mobile nodes shows that Octopus achieves excellent fault-tolerance at a modest overhead: when all nodes intermittently disconnect and reconnect, Octopus achieves the same high reliability as when all nodes are constantly up.

Keywords: ad-hoc routing, position-based routing, fault-tolerance
Contact Author: Roie Melamed
Affiliation: The Technion Department of Computer Science.
Address: Technion - Israel Institute of Technology, Department of Computer Science, Technion City, Haifa, 32000 Israel.
E-mail: mroi@cs.technion.ac.il.

*A preliminary version of this paper appears in Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005) October 26-28, 2005, Orlando, Florida.

1 Introduction

Mobile ad-hoc networks (MANETs) consist of mobile wireless nodes that communicate with each other without relying on any infrastructure. Therefore, routing in MANETs is performed by the mobile nodes themselves. Such nodes often intermittently disconnect from the network due to signal blockage [6, 19]. Thus, an important challenge that ad-hoc routing protocols should address is coping with such failures (or disconnections) without incurring high overhead. Our goal is to provide *fault-tolerance*, i.e., high routing reliability when many nodes frequently disconnect and reconnect, without sacrificing efficiency in routing in large MANETs consisting of hundreds of mobile nodes.

We consider *position-based routing protocols*, in which each node can determine its physical location. Such protocols scale better than non-position-based ones [21]. Typically, the location of each node is stored at some other nodes, which act as *location servers* for that node [21, 12]. When a node wishes to send packets to another node, it first issues a *location query* in order to discover the target's location, and then *forwards* packets to this location. In position-based protocols, reliability is measured as the success rate of location queries [18].

Position-based protocols differ from each other mainly in how many location servers store each node's location [21]. E.g., in DREAM [5], each node acts as a location server for all nodes, and in LAR [16], each node is a location server for its one-hop neighbors only. It has been argued [18] that neither of these extreme approaches is appropriate for large networks, since they both use flooding to disseminate either position information (DREAM) or location queries (LAR). Li et al. [18] have proposed the Grid Location Service (GLS), which stores each node's location at small number of nodes. They have shown that this approach, called *all-for-some* [21], achieves good tradeoff between reliability and load: each node updates its location at small number of nodes without flooding the network, and location queries incur a reasonable overhead. Li et al. have further shown that in a small network, GLS tolerates intermittent node disconnections well [18]. However, as we show in Section 6.4, in large networks, GLS's fault-tolerance greatly degrades. For example, in a grid of $2.3km$ by $2.3km$, with an average of 400 nodes connected to the network at a given time, when half the nodes intermittently disconnect and reconnect, GLS's query success rate is only 65%; when all the nodes intermittently disconnect and reconnect, it drops to 53%.

There is an inherent tradeoff between fault-tolerance and load in all-for-some protocols, since fault-tolerance is achieved by constantly updating the location of each node at multiple location servers, which in typical all-for-some protocols [18, 14] are far from each other (in order to allow for quick location discovery). Thus, each node updates each of its location servers separately, causing the load to increase with the level of redundancy. Moreover, a location update packet is typically relayed several times before it reaches the appropriate location server, and the average number of relays increases with the network area. In order to reduce the location update overhead, in most all-for-some routing protocols, e.g., [18, 14], remote location servers are updated less frequently than close ones. In Section 6.4, we show that in large networks this approach greatly degrades the fault-tolerance as routing often uses stale information.

In order to achieve a better tradeoff between load and fault-tolerance, we introduce a new location update technique called *synchronized aggregation*. In this technique, each location update packet includes the locations of several nodes and updates many location servers. Moreover, updates are synchronized in the sense that only one node initiates the propagation of an aggregate update from a given region, and hence no duplicate updates are sent. It is worth noting that such a synchronized aggregation technique is not feasible in existing all-for-some protocols, e.g, [18, 14], in which the locations of nearby nodes are stored at non-adjacent location servers.

In Section 4, we present Octopus, a simple and efficient all-for-some routing protocol that employs synchronized aggregation in order to achieve high fault-tolerance without incurring a high load. Octopus divides the network area into horizontal and vertical strips, and stores the location of each node at all the nodes residing in its horizontal and vertical strips. This approach naturally supports synchronized aggregation: all the nodes in the same strip can learn each other's locations through the propagation of exactly two location update packets along the strip. Note that this location update technique does not require nodes to synchronize their clocks: by knowing its immediate neighbors' locations, a node can determine whether it needs to initiate a strip update. The propagation of a strip update packet does not require synchronization at all. Since synchronized aggregation dramatically reduces the location update over-

head, Octopus can update all the location servers at the same high frequency- at a low cost.

On the one hand, Octopus enforces higher redundancy and more freshness of location information than previously suggested all-for-some protocols [18, 14], and hence achieves much better fault-tolerance. On the other hand, by aggregating node locations and synchronizing their propagation, Octopus incurs lower overhead than these protocols in typical scenarios.

Octopus has a third important advantage over most previous all-for-some protocols, e.g., [18, 14]: In Octopus, the area in which nodes reside does not need to be pre-known or fixed; it can change at run time. This feature is crucial for rescue missions and battle field environments, in which the borders of the network are not known in advance and are constantly changing.

Finally, the redundancy of location information in Octopus has a fourth advantage: nodes use information they have about strip neighbors in order to improve the forwarding reliability. Hence, we eliminate the need to maintain designated information (for example, two-hop neighbor lists as in [18]) for improving the forwarding reliability.

In Section 5, we analyze Octopus's scalability: we prove that under a fixed node density, the number of location update packets per node per seconds is constant, and the byte complexity grows as $O(\sqrt{N})$ with the number of nodes N. We also analyze the probability for update and query propagation failures in Octopus's horizontal and vertical strips, and show that under reasonable density assumptions, the probability for holes is very small.

In Section 6, we evaluate Octopus's performance using extensive ns2 simulations with up to 675 mobile nodes. Our results show that Octopus achieves high routing reliability, low overhead, good scalability, and excellent fault-tolerance. For example, in a grid of $2.3km$ by $2.3km$ with nodes that *all* intermittently disconnect and reconnect, and an average of 400 connected nodes at a given time, Octopus achieves a query success rate of 95%, which is identical to the success rate when all nodes are constantly up. We also compare Octopus to GLS, the position-based protocol that achieved the best reliability-load tradeoff thus far. Our results indicate that in the absence of failures, Octopus achieves slightly better reliability than GLS, at lower overhead (both packets and bytes). In failure-prone settings, Octopus's reliability is greatly superior to that of GLS.

2 Related Work

Existing ad-hoc routing approaches can be roughly divided into two categories: *topology-based* and *position-based* [21]. Topology-based protocols do not assume that each node can determine its position. Such protocols usually employ global flooding to distribute either topology information (e.g., DSDV [24]) or queries (e.g., AODV [25], DSR [15], TORA [23], and ZRP [13]), and hence suffer from limited scalability [21, 18].

By assuming that each node can determine its location, position-based protocols achieve better efficiency and scalability than topology-based ones [21]. Position-based protocols can be classified according to how many nodes act as location servers and how many locations each of them holds [21]. In the *all-for-all* approach used by DREAM [5], every node acts as a location server for all nodes. This approach is fault-tolerant, and is practical in small networks. However, it has been argued that the overhead of this approach is prohibitive in large networks, since location updates are flooded [18, 11].

In the *some-for-some* [12] and *some-for-all* approaches [12, 30], some dedicated nodes act as location servers for some or all other nodes. These approaches are appropriate for failure-free networks, or for settings in which there are reliable servers. However, such approaches are problematic in failure-prone networks, since they are vulnerable to the movement or failure of a single dedicated location server (as explained in [18]).

Octopus employs the *all-for-some* approach, in which each node acts as a location server for some other nodes. Li et al. [18] have shown that this approach can achieve a good tradeoff between reliability and load, and can scale well up to at least 600 nodes. All-for-some-based protocols include GLS [18], GRSS [14], Homezone [10, 26], and [27]. Of these, GLS and GRSS are the only ones that were extensively evaluated in simulations with mobile nodes. Moreover, only GLS was evaluated in settings in which nodes intermittently disconnect from the network, and this study was only conducted in a small network.

Stojmenovic et al. [27] suggest a routing scheme in which each node periodically propagates its position in the north and south directions, and location queries are sent in the east and west directions. Similar approaches were also suggested for efficient content location [29], match-making in sensor networks [4], and as a general scheme for implementing

ad-hoc services [22]. However, unlike Octopus, none of these previous works aggregate updates, and they thus miss Octopus's important performance advantage; individually updating so many nodes is bound to induce a prohibitively high overhead [3, 20]. Moreover, of these works, only [29] was evaluated with mobile nodes, and none was evaluated in fault-prone settings. Another difference between Octopus and [27] is that Octopus employs more redundancy by storing node locations at both their horizontal and vertical strips. This additional redundancy yields a quadratic decrease in the probability for query failures. Finally, [27] does not make additional use of the stored location information in order to improve the reliability of forwarding. In fact, we are not aware of any previous ad-hoc routing protocol that exploits location information for more effective forwarding.

The most thoroughly studied position-based protocol thus far, GLS [18], partitions the world into a hierarchy of grids with squares of doubling edge sizes. In each level of the hierarchy, the location of each node is stored at three location servers, for a total of $O(\log N)$ location servers under uniformity and fixed density assumptions. Under the same assumptions, Octopus stores the location of each node at $O(\sqrt{N})$ location servers (see Section 5). In contrast to Octopus, in GLS remote location servers are updated less frequently than close ones. Thanks to the use of more location servers and fresher information, Octopus achieves much higher fault-tolerance than GLS. Thanks to aggregation, Octopus achieves this while incurring lower overhead. Moreover, Octopus is a simpler protocol than GLS.

Although Octopus requires more memory than GLS for storing location information, Octopus's memory requirements are quite reasonable: in our largest experiment, with 675 nodes, location information consumes less than 1KB of memory at each node. Note that in wireless networks, reducing the number of transmissions is most crucial, and 1KB of memory overhead is a small price to pay for the significant reduction in message overhead that Octopus achieves.

In almost all the previous location-based routing protocols, each location update packet includes the location of a single node and updates a single location server. The only exception we are familiar with is GRSS [14]. However, in contrast to Octopus, in GRSS location updates are not synchronized, i.e., several nodes in the same region can initiate a location update simultaneously, thus causing many duplicate packets to be sent. Consequently, as shown in [14], GRSS often fails to achieve lower overhead than GLS. Moreover, as opposed to Octopus, in which each location update packet contains identities of $O(\sqrt{N})$ nodes (assuming the system model described in Section 3), in GRSS, a location update packet can contain $O(N)$ node identities. In order to reduce the packet size, GRSS uses Bloom filters. However, this technique may lead to incorrect routing due to false positives [14].

In LAR [16], each node knows only the locations of its immediate neighbors. Stojmenovic et al. [28] improve LAR by using a selective flooding algorithm that eliminates loops, and hence reduces global flooding without impacting success rate. This approach is efficient when the number of location queries is low. However, when location queries are frequent, this approach is not practical, as location queries may be globally flooded [18].

Finally, some ad-hoc protocols, e.g., Span [7] and GAF [33], focus on reducing energy consumption by allowing nodes to sleep for extensive periods, leaving a minimal set of nodes awake to perform routing. Such an approach employs no redundancy, and hence is inherently not fault-tolerant. The fault-tolerance of this approach is improved by algorithms, e.g., Wu and Li [32] and Dai and Wu [8], that organize the nodes into a richly-connected approximate *dominating set*[1]. Such algorithms are efficient in a failure-free low mobility network. However, in a dynamic failure-prone network, these algorithms can incur high overhead, since they constantly need to update and recalculate the approximate dominating set when the underlying graph changes [32]. In addition, these algorithms perform badly in the worst case ($O(N)$-approximation) [9].

3 System Model

The network consists of a collection of mobile nodes moving in a rectangular space. The set of nodes can change over time as nodes connect and disconnect. The coordinates of the space can also change over time. Each node can determine its own position, e.g., using GPS. Each node can broadcast packets to all its neighbors within a certain radius r called the radio range. Packets can be lost due to MAC-level collisions or barriers.

[1] A set is dominating if all the nodes in the MANET are either in the set or neighbors of nodes in the set.

In our simulations, we use the MAC layer provided by the *ns2* simulator, which simulates packet loss in typical MANETs. As in other protocols [18, 14], a certain minimal node density throughout the grid is required in order to ensure reliability. Thus, we assume that the number of nodes grows proportionally with the area of the network. As in [18, 14], we assume that nodes are uniformly distributed in the space.

Octopus divides the space into horizontal and vertical strips. The strip width, w, is constant and known to all nodes. Knowing w, the zero longitude and latitude, and its current location, each node can determine which horizontal and vertical strips it resides in at a given time. For example, in Fig. 1, node S resides in the highlighted horizontal and vertical strips, and its radio range neighbors are circled. Each strip has a unique identifier (of type StripID), identifying its location relative to the global zero coordinates.

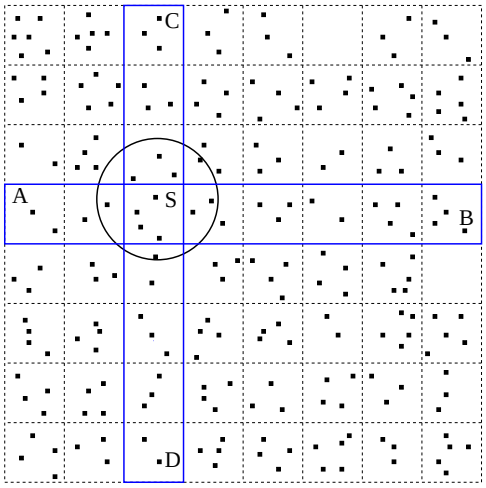


Figure 1: Node S's neighbors and strips. A, B, C, and D are end nodes in the highlighted strips.

4 Octopus

Octopus is composed of three sub-protocols: *location update*, *location discovery*, and *forwarding*. The location update protocol maintains each node's location at its designated location servers, as well as at its radio range neighbors. When a node wishes to send packets to another node, it first issues a *location query* to the location discovery protocol in order to discover the target's location, and then uses the forwarding protocol to forward packets to this location. Sections 4.1, 4.2, and 4.3 present Octopus's location update, location discovery, and forwarding sub-protocols, respectively. The types and data structures used in the three sub-protocols are presented in Fig. 2.

Types:
NodeID – a node identifier.
StripID – a strip identifier.
Direction – in {*north*= 0, *south*= 1, *west*= 2, *east*= 3}
Node – ⟨NodeID *id*, Real x, Real y, Time *time*, StripID *hid*, StripID *vid*, Real *p_x*, Real *p_y*, Time *p_time*⟩
Data structures
Node *this* – this node.
Set of Node *neighbors*, *strip*[4], *recent_locations*.

Figure 2: Octopus's types and data structures.

In all three sub-protocols, we use limited retransmissions in order to partially overcome packet loss: Whenever a node A sends a packet to a node B, and B is expected to send a packet in return (e.g., to propagate/forward the packet further or respond to a location query), node A waits to hear the appropriate packet from B. If A does not hear B's packet within a *retransmissions_timeout*, then A chooses another node C, distinct from B, and re-sends the packet to C. Up to two retransmission attempts are made per packet.

4.1 Location Update

Octopus synchronizes location updates by having them initiated only by each strip's *end nodes*. A north (south) end node is a node that has no neighbors in direction north (respectively, south) in its vertical strip, and a west (east) end node is a one that has no neighbors to the west (respectively, east) in its horizontal strip. For example, in Fig. 1, A, B, C, and D are end nodes in S's strips. Periodically, an end node initiates a strip update packet, which propagates along the strip towards the end node at the other side of the strip.

The location update protocol maintains two data structures at each node: *neighbors* – radio range neighbors, and *strip[i]* for $i \in$ {*north, south, west, east*} – nodes residing in direction i in the node's strip. Each element in these sets is of type Node. As shown in Fig. 2, this type is a tuple including the following fields: *id* – the node's identifier, x, y – the node's last reported coordinates, *time* – the time of the last received coordinates report, *hid*, *vid* – the node's horizontal and vertical StripIDs, *p_x*, *p_y* – the node's previous coordinates, and *p_time* – the time of the previous received coordinates report.

The *neighbors* set is updated upon receiving a short HELLO packet from another node. This packet is broadcast by every node every *hello_timeout* seconds, and it contains the broadcasting node's identity and physical coordinates. If a node does not hear from some neighbor n for $2hello\_timeout$ seconds, it removes n from *neighbors*.

The pseudo-code for maintaining *strip[*]* is presented in Fig. 3. The locations of all the nodes in a given strip are propagated through the strip via the periodic diffusion of STRIP_UPDATE packets initiated by the end nodes of the strip every *strip_update_timeout*. An end node broadcasting a STRIP_UPDATE packet to direction d includes in the packet all its *neighbors* that are in the same strip. A STRIP_UPDATE packet also includes the strip identifier, the packet direction, and a target node, which will forward this packet further. The target is chosen to be the farthest node in the propagation direction.

```
loop forever
   foreach Direction d do
      if (I have no neighbors in direction d) then
         strip[d] ← ∅
         StripID sid ← get_strip_id (d)
         propagate_packet(sid, opposite direction to d)
   sleep (strip_update_timeout)

Event handler:
upon receive ⟨STRIP_UPDATE, sid, d, set, next⟩ do
   if (sid = this.vid ∨ this.hid) then
      strip[opposite direction to d] ← set
   /* If I am the packet target */
   if (this = next) then
      propagate_packet (sid, d)

Procedures:
set of Node get_nodes_in_strip (sid)
   return {this} ∪ {n ∈ neighbors|n.hid = sid ∨ n.vid = sid}

StripID get_strip_id (d)
   if d ∈ {north, south} then
      return this.vid
   return this.hid

void propagate_packet (sid, d)
   set of Node set ← strip[opposite direction to d]
                     ∪ get_nodes_in_strip(sid)
   Node next ← farthest node in direction d in set
   /* If propagation is not complete */
   if (this ≠ next) then
      bcast ⟨STRIP_UPDATE, sid, d, set, next⟩
```

Figure 3: The strip update protocol.

Upon receiving a STRIP_UPDATE packet, a node updates the appropriate entry in *strip[*]*. If the node is designated as the packet target and is not the strip's end-node, then it appends to the packet all its *neighbors* that reside in the packet's strip, chooses a new target, and broadcasts the packet. The propagation of a STRIP_UPDATE packet completes when it reaches an end node, i.e., when the farthest node in direction d is the current node ($this = next$). For example, in Fig. 1, a STRIP_UPDATE packet with direction south begins at node C and propagates to the south-most node of the strip, D.

Forwarding holes

We define a *forwarding hole* to be a situation in which a node X cannot forward a STRIP_UPDATE packet to direction d in a strip s although there is another node in s that is in direction d of X. For example, in Fig. 1, there is a forwarding hole south of node B. In a typical scenario, the probability for a forwarding hole is small (less than 0.02, see Section 5.2). Moreover, as we describe in Section 4.2, storing each node's location at both the horizontal and vertical strips quadratically decreases the probability for query failures due to forwarding holes. Finally, we also implemented several mechanisms based on face routing [17] for bypassing a forwarding hole. However, these mechanisms achieve only a negligible increase in Octopus's reliability, since the probability for a forwarding hole is negligible as explained above. Hence, we present and evaluate Octopus without these mechanisms.

Although the probability for a routing failure due to forwarding holes is small, we have implemented a simple bypass mechanism in order to overcome such failures: in this mechanism, a node that cannot forward a STRIP_UPDATE packet to direction d in a strip s forwards the packet to a node that is in direction d of it and resides in an adjacent strip to s. Empirically, the additional reliability achieved by this bypass mechanism is negligible (less than 2%), since Octopus already achieves high reliability without it. Therefore, for simplicity reasons, we present and evaluate Octopus without the bypass mechanism.

Correctness

We now identify circumstances under which Octopus's location update protocol achieves 100% reliability, i.e., correctly stores node locations at all of their designated location servers. We note, however, that in the presence of failures, movements, packet loss, and uneven node distribution, these ideal circumstances are not always achieved. Nevertheless, in Section 6, we show that in typical scenarios with fre-

quent failures and movements, Octopus's reliability is close to 95%.

Lemma 1. *In a run in which there are no node movements or failures and no packet loss, if the strip width $w \leq \frac{\sqrt{3}}{2}r$ and the bound on packet delay is less than hello_timeout, then in every segment of a strip in which there are no forwarding holes, every node eventually knows the identities and locations of all the nodes that reside in this segment.*

Proof. We first note that all the nodes' neighbors' sets are accurate, i.e., include exactly all the nodes within their radio range, since there is no packet loss, the bound on packet delay is less than *hello_timeout*, and a node is removed from the current node's *neighbors* set only if the current node does not hear from this node for 2*hello_timeout* seconds. Therefore, after at most 2*hello_timeout* seconds, only an end node initiates a propagation of a STRIP_UPDATE packet. Note also that in a segment of a strip with no holes, a propagation of a STRIP_UPDATE packet from one end node is guaranteed to eventually reach the other end node of the segment, since there is no packet loss or failures.

Consider a segment of strip s with no holes. Assume that the segment's end node A sends a STRIP_UPDATE packet $m1$ to node B, and then B sends a STRIP_UPDATE packet $m2$ to node C. Without loss of generality, assume that s is a horizontal strip. Consider a node N in s whose x coordinate is between A's and B's, at distance Δx from A's x coordinate. If $\Delta x \leq \frac{r}{2}$, then N is in A's radio range, and hence it receives $m1$. Since $w \leq \frac{\sqrt{3}}{2}r$ and A's *neighbors* set is accurate, $m1$ contains all the nodes in s within $\frac{r}{2}$ meters of A in the direction of $m1$, as all these nodes are within A's radio range (see Fig. 4). Therefore, after receiving $m1$, N knows the identities and locations of all the nodes between it and A. If $\Delta x > \frac{r}{2}$, then N receives $m2$ as it is in B's radio range (see Fig. 4). According to the protocol, since A's and B's *neighbors* sets are accurate, $m2$ contains all the nodes in s that are within A's and B's radio ranges. Thus, in both cases, after the broadcast of $m2$, N knows the identities and locations of all the nodes in s whose x coordinates are between N's and A's. Note that, since there are no movements or failures, and since only end nodes initiate updates, parallel propagations of different STRIP_UPDATE packets do no violate the protocol's correctness, as such packets contain the same information.

By induction, we get that after propagating a STRIP_UPDATE packet from A to Z, the end node at the other end of the segment, each node knows the identities and locations of all the nodes in the segment between it and A. Likewise, after propagating a STRIP_UPDATE packet from Z to A, each node knows the identities and locations of all the nodes in s between it and Z. □

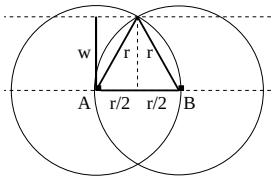


Figure 4: A strip of width $w = \frac{\sqrt{3}r}{2}$.

Although Lemma 1 requires $w \leq \frac{\sqrt{3}r}{2}$ to ensure that nodes are not missed by a STRIP_UPDATE propagation, the simulations in Section 6.1 show that increasing w from $\frac{\sqrt{3}r}{2}$ to r does not hurt the reliability, since increasing w also reduces the probability for forwarding holes (see Section 5.2), and hence may increase the reliability.

4.2 Location Discovery

The location discovery protocol uses the information stored in *strip[*]* and *neighbors*, as well as the set *recent_locations*, which is a cache of recently discovered target locations. The cache entries expire after *strip_update* seconds. The location discovery protocol is presented in Fig. 5.

The interface to the location discovery protocol consists of the function *locate*, which upon success results in addition of its target to recent_locations. It first searches the target in one of the locally maintained sets (*strip[*]*, *neighbors*, and *recent_locations*). If the target's location is not found in these sets, the protocol broadcasts two QUERY packets to the node's north-most and south-most neighbors in its square or in adjacent squares in its vertical strip. The recipient of a QUERY packet continues the search in the same manner, forwarding the packet in the same direction if needed. Once a QUERY packet reaches a node that knows the target, it broadcasts a REPLY packet with its information about the target towards the source. Every node that receives a REPLY packet adds the located target to its *recent_locations*. In rare cases in which no REPLY packet is received within *discovery_timeout* seconds, the search is repeated in the same manner in a west-east directions.

```
locate (Node_ID tid)
    Node target ← search_locally (tid)
    if (target = null) then
        search_location (this, tid, north)
        search_location (this, tid, south)
        sleep (discovery_timeout)
        if (target ∉ recent_locations) then
            search_location (this, tid, west)
            search_location (this, tid, east)

Event handlers:
upon receive ⟨QUERY, src, t_id, d, next⟩ do
    if (next = this) then
        Node target ← search_locally (t_id)
        if (target = null) then
            search_location (src, t_id, d)
        else /* target found - send reply */
            Direction d' ← opposite direction to d
            send_reply (src, target, d')

upon receive ⟨REPLY, src, target, d, next⟩ do
    recent_locations ← recent_locations ∪ {target}
    if (next = this) then
        send_reply (src, target, d)

Macro:
strip_neighbors[d] ≜ (neighbors ∩ strip[d]) ∪ {this}

Procedures:
Node search_locally (target_id)
    if (∃n s.t. n ∈ neighbors ∪ strip[∗] ∪ recent_locations
        ∧n.id = target_id) then
        return n
    return null

search_location (src, t_id, d)
    Node next ← farthest node in strip_neighbors[d] in the
                same square as this or in an adjacent square
    if (next ≠ this) then
        bcast ⟨QUERY, src, t_id, d, next⟩

send_reply (src, target, d)
    Node next ← closest node to src in strip_neighbors[d]
    if (next ≠ this) then
        bcast ⟨REPLY, src, target, d, next⟩
```

Figure 5: The location discovery protocol.

Fig. 6 depicts how node S discovers node T's location. S broadcasts QUERY packets to the north and south. The next hop of the north-going packet is I. I fails to discover T's location locally, and forwards the packet to its north-most neighbor J. T is in J's *strip[east]*. Thus, J broadcasts a REPLY packet containing T's location towards S. This packet reaches I, which in return broadcasts the packet to S.

Correctness

As in the previous section, we identify circumstances under which Octopus's location discovery service achieves 100% reliability.

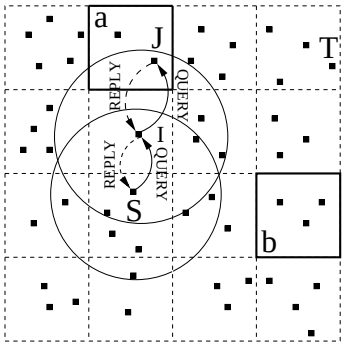


Figure 6: Successful query location.

Lemma 2. *Consider a run with no node movements, node disconnections, or packet loss, and assume that* $w \leq \frac{\sqrt{3}}{2}r$ *and the bound on packet delay is less than hello_timeout. Consider a location query with nodes* S *and* T *as the query's source and target, respectively. Let square a (b) be the intersection between* S*'s vertical (horizontal, respectively) strip and* T*'s horizontal (vertical, respectively) strip (see Fig. 6). If there are no forwarding holes between* S *and a and between* T *and a, or there are no holes between* S *and b and between* T *and b, then* S*'s* recent_locations *eventually includes* T*'s location.*

Proof. Without loss of generality, assume that there are no forwarding holes between S and a and between T and a. Since QUERY packets never skip over squares (see *search_location* in Fig. 5) and there is no packet loss, a QUERY packet propagating along the strip reaches to some node N that resides in a. By Lemma 1, N knows T's location. Since N does not move or fail, it initiates a REPLY packet. Since there are no holes or packet loss, this packet propagates back to S, and S includes T in its *recent_locations* set. □

4.3 Data Forwarding

Fig. 7 describes the process of sending a data packet m from the current node S to a target node T. First, S calls to the function *locate* (see Fig 5) in a separate thread. When S's *recent_locations* set contains T's location, S forwards the data packet to T using the interface $forward$ of the forwarding protocol.

Octopus employs geographic forwarding [21] in order to forward data packets to their destinations. The basic version of geographic forwarding works as follows: each node has knowledge of its one-hop neighbors and their locations. Each intermediate node forwards a data packet to its neighbor that is geograph-

```
send (m, T)
    spawn thread to run locate(T)
    wait until exists n ∈ recent_locations s.t. n.id = T
    forward (m, n)

forward (Packet p, Node target)
    update_coordinates (target)
    Node next ← closest node to target in neighbors ∪ {this}
    if (next = this) then
        target' ← closest node to target in strip[*]
        update_coordinates (target')
        next ← closest node to target' in neighbors
    bcast ⟨FORWARD, p, target, next⟩

Event handler:
upon receive ⟨FORWARD, p, target, next⟩ do
    if (target = this) then
        deliver p
    else if (next = this) then
            forward (p, target)

Procedure:
update_coordinates (t)
    Update t.x, t.y, t.time according to the current time and t's
    direction of movement obtained from t's last two reported
    coordinations. Store old values in t.p_x, t.p_y, t.p_time.
```

Figure 7: The forwarding protocol.

ically closest to the packet's destination. This protocol is efficient, but it may fail if an intermediate node is a *local maximum*, i.e, it is closer to the destination than all of its neighbors.

In case of a forwarding failure, Octopus chooses an alternative target, $target'$, which is the closest node to the packet destination from the sets *strip[*]* and forwards the packet to its neighbor that is geographically closest to $target'$. We illustrate this recovery technique in Fig. 8, where node S needs to forward a data packet to node T. S is closer to T than all of its radio range neighbors. S chooses node E (the closest node to T from S's *strip[*]*) as an alternative target, and forwards the packet to A (S's closest neighbor to E). Note that the packet's ultimate destination remains unchanged, and subsequent forwarding steps follow the basic geographic forwarding if possible. In Section 6, we show that this recovery technique is very effective, achieving the same reliability as two-hop geographic forwarding as used in [18] .

Since nodes continue to move while packets are en route to them, it is important to constantly *re-estimate* the target's location. In each forwarding step, the forwarding node forwards the data packet to the target's estimated location. This location is calculated according to the target's last two reported coordinates, which are included in the Node data structure sent in REPLY and FORWARD packets.

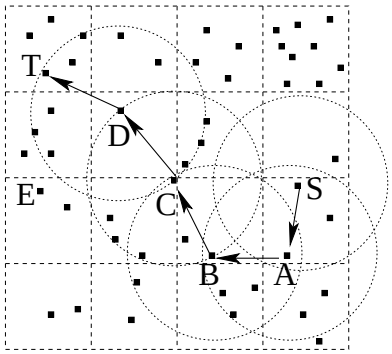


Figure 8: Octopus's forwarding protocol.

5 Analysis

In Section 5.1, we analyze Octopus's scalability, and in Section 5.2 we analyze the probability for forwarding holes.

5.1 Scalability

The following lemma shows that the message complexity of Octopus's location update protocol is constant with respect to the network size.

Lemma 3. *Assuming a fixed node density ρ, the per node per second packet complexity of the location update protocol does not grow with the network size.*

Proof. Consider a STRIP_UPDATE packet forwarded by a node n that is within a distance r or more of the end of the grid (that is, any internal node in the grid). Assuming a fixed node density ρ, then the expected distance between n and the packet's next destination depends only on ρ, and not on the network size. This is because n forwards the message to the farthest node in its neighborhood. Asymptotically, when the grid is large, most of the nodes are not close to the ends of the grid. Hence, we neglect the effect of the location of the forwarding node on the average propagation distance. Denote the average propagation distance by δ.

Second, we observe that the probability for a forwarding hole at any particular point in the strip is independent of the network size. Therefore, the average percentage of the strip in which there are no forwarding holes is constant with respect to the network size. Denote this portion by α.

In a single iteration of the strip update protocol, the propagation of STRIP_UPDATE packet(s) along a strip with an edge length of e requires an average of $\frac{\alpha e}{\delta}$ transmissions in each direction. Denote $\sigma = 1/strip\_update\_timeout$. Then on average, $\frac{2\alpha e\sigma}{\delta}$

STRIP_UPDATE packets per strip are sent in a second. In order to obtain the average per node message complexity, we divide this number by the expected number of nodes in a strip, which is ρew, and multiply it by 2 since STRIP_UPDATE packets are propagated in both horizontal and vertical strips. Therefore, on average, each node broadcasts $\frac{4\alpha e\sigma}{\delta \rho ew} = \frac{4\alpha\sigma}{\delta\rho w}$ STRIP_UPDATE packets per second, which is independent of the network size.

In addition to STRIP_UPDATE packets, the location update protocol also sends HELLO packets. Since each node broadcasts HELLO packets at a fixed frequency, the total per node per second message complexity incurred by the location update protocol is constant with respect to the network size. □

The next lemma shows that the byte complexity of Octopus's location update protocol with N nodes is $O(\sqrt{N})$.

Lemma 4. *Assuming a fixed node density, the per node per second byte complexity incurred by the location update protocol with N nodes is $O(\sqrt{N})$.*

Proof. Recall that in our model, we assume that N nodes are uniformly distributed in the network area. Therefore, assuming a fixed node density, when we increase N, the network edge size, e, increases by $O(\sqrt{N})$, and therefore, the number of nodes in each strip increases like $O(\sqrt{N})$. Thus, the number of bytes in STRIP_UPDATE packets increases like $O(\sqrt{N})$. The size of a HELLO packet is constant.

From Lemma 3, we get that the number of packets sent per node does not increase with N, and therefore the overall per node byte complexity of the location update protocol is $O(\sqrt{N})$. □

5.2 Update/Query Propagation Reliability

Forwarding holes in strips may hamper Octopus's reliability, as they may prevent location updates from propagating in the entire strip. We now analyze the probability for forwarding holes. We show that under reasonable density assumptions, this probability is very small, which explains why Octopus achieves excellent reliability in the simulations below.

A forwarding hole occurs when a node has no radio range neighbors in the strip in the direction the packet is going, i.e., when there are no nodes in the intersection between the forwarding node's radio range and the strip in the packet's direction. For example, in Fig. 9, a hole in N's east direction occurs if there are no nodes in the area denoted by A. The size of this area depends on w, r, and the node's location relative to the strip boundaries. Without loss of generality, let us examine a horizontal strip. Consider a node whose y coordinate is at distance d from the south boundary of the strip. Using the equation for the area of a circular segment [31], we compute A as follows:

$$A_s(d) = r^2 \cos^{-1}\left(\frac{d}{r}\right) - d\sqrt{r^2 - d^2}$$

$$A\ (d) = \frac{\Pi r^2 - (A_s(d) + A_s(w-d))}{2}$$

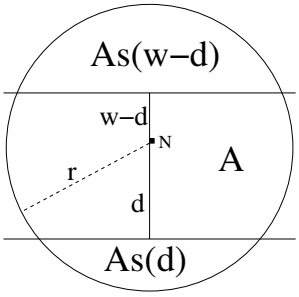


Figure 9: Node N has a forwarding hole in direction east if area A is uninhabited.

For an asymptotic analysis, we use a Poisson node distribution. Since the expected number of nodes in an area of size A is ρA, we get that the probability of no nodes residing in A is:

$$Pr_d = e^{-\rho A(d)}$$

Since this probability varies with d, in order to compute the average probability for a forwarding hole we need to average Pr_d for d's in $[0, w]$. We observe that Pr_d monotonically decreases when d grows from 0 to $w/2$ (as the area gets larger), and then symmetrically increases as d grows from $w/2$ to w. The highest probability occurs when $d = 0$ or $d = w$. We compute a coarse lower bound of the probability for holes by considering two cases: first, when d is between $w/4$ and $3w/4$, and second when d is not in the middle half of the strip. We bound the probability for the first case by looking at its minimum point, where $d = w/4$, and we bound the second case by looking at its minimum point, where $d = 0$. We get the following:

$$Pr[\text{hole}] < \frac{1}{2}Pr_{w/4} + \frac{1}{2}Pr_0$$

When we instantiate the formula above with $\rho = 75$, $w = r = 0.25$ (used in most of our simulations), we get that $Pr[\text{hole}] < 0.02$. This explains why Octopus achieves high query success rate in typical scenarios. With a strip width of $0.2 = \frac{4r}{5} < \frac{\sqrt{3}r}{2}$, which ensures that location updates and queries are received at all the nodes residing in segments of the strip they propagate through, we get that $Pr[\text{hole}] \approx 0.0327$. Hence, we see that two opposing tendencies affect the protocol's reliability: increasing w beyond $\frac{\sqrt{3}r}{2}$ reduces the probability for a forwarding hole, and hence increases the reliability, but it also increases the probability that a location update or query will not be received by all the nodes residing in segments of the strip it propagates through. Our simulations in Section 6.1 show that these two strip widths achieve virtually the same reliability.

6 Evaluation

We now evaluate Octopus using simulations. Octopus is implemented in *ns2* [2] with CMU's wireless extensions. Each node uses the IEEE 802.11 radio and MAC model provided by the CMU extensions, with a radio range r of 250 meters and a throughput of $1\frac{Mb}{sec}$. The nodes are initially placed uniformly at random in a square universe. In most of our simulations, there are 75 nodes per square kilometer. (Li et al. [18] have experimentally shown that such a node density is required in order to achieve high forwarding reliability.) Each node moves using the random waypoint model used in [18]: it chooses a random destination and moves toward it with a constant speed chosen uniformly between zero and $10\frac{m}{sec}$. When a node reaches its destination, it chooses a new destination and immediately begins moving toward it at the same speed. For each set of parameters, we run five 300 seconds long simulations, and in each simulation, each node initiates an average of one location query a minute to random destinations, starting 30 seconds into the simulation, and ending at 270 seconds. In all of our experiments, the results of all the five simulations were very close to each other. This consistency is due to the large number of events in each simulation.

In Section 6.1, we discuss our choice of the protocol's parameters. In Section 6.2, we examine Octopus's scalability as the number of nodes and network area increase. In Section 6.3 we evaluate the reliability of Octopus's forwarding sub-protocol and compare it with two-hop geographic forwarding. In Section 6.4, we study Octopus's fault-tolerance. Finally, in Section 6.5, we compare Octopus's reliability, overhead, and fault-tolerance to those of GLS.

6.1 The Choice of Parameters

In the simulations reported below, each node broadcasts a HELLO packet every 2 seconds, as was done in GLS [18]. We chose this frequency in order to allow a fair comparison between the two protocols. Nevertheless, we also ran experiments with a *hello_timeout* of up to five seconds, and the results were virtually identical. This occurs due to the nature of movement in the random way point model, which allows a node to predict a neighbor's location in the near future from the neighbor's last two reported coordinations.

We set the *strip_update_timeout* to 10 seconds. Empirically, reducing this value, e.g., to 5 seconds, results in a negligible increase in the protocol's reliability. On the other hand, increasing this timeout to 20 seconds, decreases the reliability by 5%−10%.

The *retransmissions_timeout* and *discovery_timeout* were set to 2 seconds each, as in other protocols, e.g., LAR [16]. This timeout value was chosen since, in all our failure-free experiments, more than 95% of the successful queries are received at the source within two seconds from the time they are issued. We allow up to two retransmissions per packet. Empirically, we observed that increasing the number of retransmissions beyond two has a negligible effect of the protocol's reliability.

Finally, we examine the effect of the strip width on the protocol's reliability and overhead. In Section 4.1, we proved that when $w \leq \frac{\sqrt{3}}{2}r$, location updates are guaranteed to cover all the nodes residing in segments of the strip they propagate through. Increasing w beyond this threshold may cause some nodes to be missed by location updates passing next to them. Nevertheless, increasing w does not necessarily hamper Octopus's reliability. This is so because it reduces the probability for forwarding holes, as it increases the area of the intersection between nodes' radio ranges and their strips (see Section 5.2), and thus reduces the probability that no nodes reside in this area. When $r = 250m$, $\frac{\sqrt{3}}{2}r = 216m$. We experiment with strip widths of 200 and 250 meters. In order to ensure a fair comparison, we examine grid edge lengths that are divisible by both 250 and 200. Fig. 10 shows the query success rate as a function of the number of nodes and the grid's edge length for OCTOPUS-250 (where $w = 250$) and OCTOPUS-

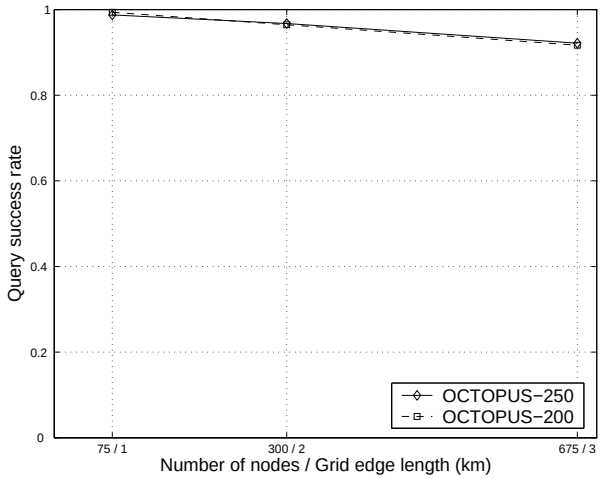
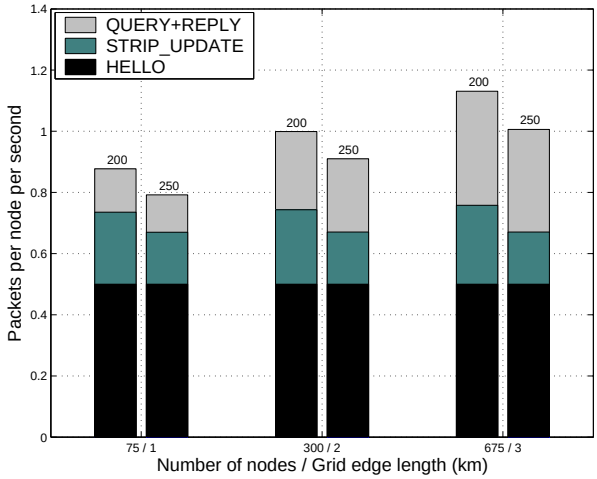


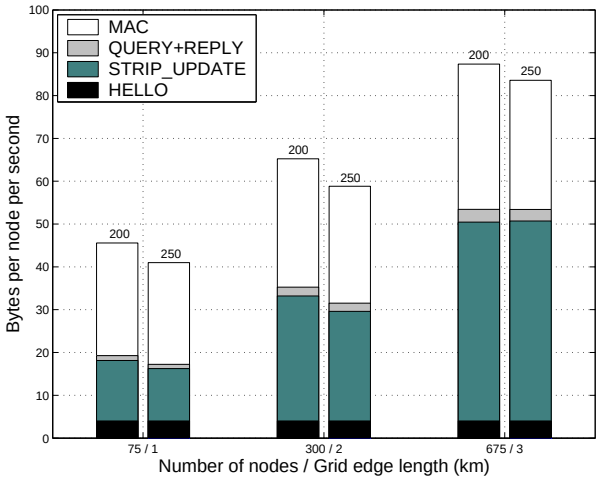
Figure 10: Octopus's query success rates for different strip widths.

200 (where $w = 200$). The 95% confidence intervals for the results presented in this figure are very tight: up to $\pm 0.8\%$ of the average value. We see that the query success rate is very similar for both strip widths. We conclude that under a density of 75 nodes per square kilometer, setting $w = r$ does not reduce the reliability compared to choosing $w \leq \frac{\sqrt{3}}{2}r$.

At the same time, increasing w reduces the number of STRIP_UPDATE packets sent, since there are fewer strips. Although the size of each STRIP_UPDATE packet increases as there are more nodes in each strip, the total number of node locations sent in all STRIP_UPDATE packets does not change. Since each transmitted packet also includes a MAC header, sending the same information in fewer packets reduces the total number of bytes sent by the protocol. Indeed, Fig. 11(a) and Fig. 11(b) show that increasing the strip width from $200m$ to $250m$ reduces the per node packet and byte complexities of Octopus. Fig. 11(a) shows for each setting the average number of packets of each type and Fig. 11(b) shows the average number of protocol bytes in each packet type as well as (in white) the average number of bytes in MAC headers. The 95% confidence intervals for the results presented in Fig. 11(a) and Fig. 11(b) are up to ± 0.01 packets and ± 0.1 bytes of the average value, respectively, indicating that the results are accurate. Henceforth, we fix the strip width at $250m$.



(a) packet overhead



(b) byte overhead

Figure 11: Octopus's overhead for different strip widths.

6.2 Scalability

We now examine Octopus's scalability. We first examine the impact of increasing the network size while maintaining a fixed node density, and then focus on the effect of increasing the node density.

6.2.1 Increasing the network size

As the network area increases, the probability for forwarding holes in the update/query path increases, and therefore, the reliability inevitably degrades. We observe that regardless of strip width or density, this degradation is very gradual (see Fig. 10).

Figure 11 examines the increase of Octopus's overhead as the network size and the number of nodes

grow. Fig. 11(a) shows that the number of location update packets sent by Octopus is constant, matching the analysis in Section 5.1. The overall packet overhead gradually increases with the network size and the number of nodes. The moderate increase in the per query overhead stems from the increased failure probability of the first discovery attempt (in the north-south directions), which leads to more cases in which locations are also searched in the east-west directions. Nevertheless, this increase is gradual, because the failure probability is low even in large grids. We note that similar phenomena occur in other all-for-some protocols [18, 14, 10, 26], where the probability for query failures also increases with the network area. This, in turn, increases the overhead due to query retries or trying alternative location servers.

Fig. 11(b) examines the increase in Octopus's byte overhead as the network size and the number of nodes grow. We note that the byte (and packet) overhead incurred by broadcasting HELLO packets is constant with respect to the networks size. Although most of the broadcasted packets are of type HELLO, their byte overhead is small, since these packets are very small. As expected, the number of bytes in STRIP_UPDATE packets increases with the network size (see Section 5.1). As explained above, the number of QUERY and REPLY packets also increases with the network size (see Fig.11(a)), and hence the number of bytes in these two types of packets also increases with the network size. However, this increase is negligible, as these packets are very small.

6.2.2 The effect of node density

We now examine what happens when the node density increases from 75 to 100 nodes per square kilometer. Fig. 12 shows that the query success rate remains similar. This occurs because of two opposing tendencies: On the one hand, increasing the density reduces the probability for forwarding holes, and thus improves reliability. On the other hand, as the node density increases, the probability for MAC-level collisions increases, and therefore, more packets are lost, which reduces the reliability. The 95% confidence intervals for the results presented in Fig. 12 are up to $\pm 1\%$ of the average value.

In Fig. 13(a) and Fig. 13(b), we see that increasing the density reduces Octopus's per node message and byte complexity. The message complexity is reduced since the number of STRIP_UPDATE packets sent in each strip does not grow, while these packets are divided among more nodes. Although the number of node locations sent in each STRIP_UPDATE increases, sending fewer packets per node reduces the MAC overhead, and the overall per node byte complexity is therefore also reduced. The 95% confidence intervals for the results presented in Fig. 13(a) and Fig. 13(b) are up to ± 0.01 packets and ± 0.1 bytes of the average value, respectively.

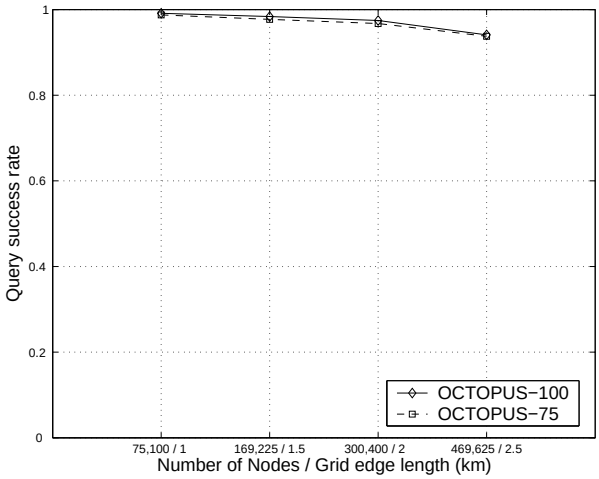
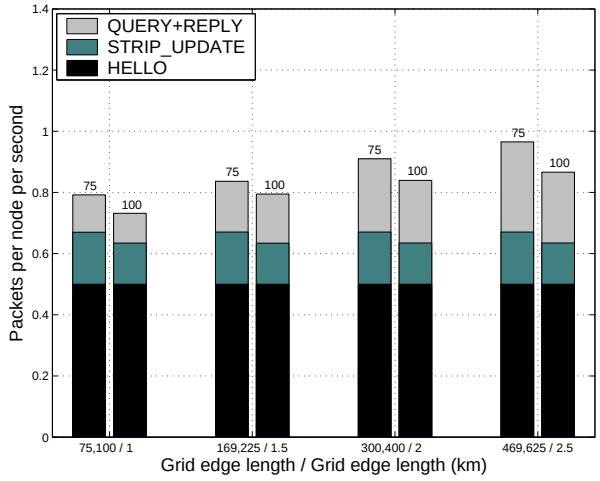


Figure 12: Octopus's query success rates for different node densities.

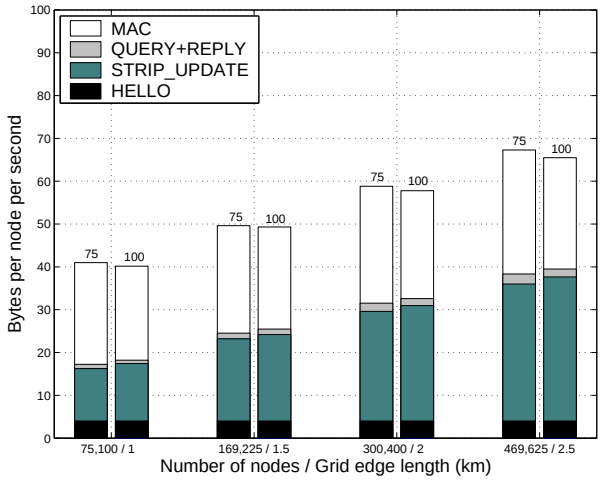
6.3 Data Forwarding

In order to evaluate the reliability of Octopus's forwarding sub-protocol, we run simulations in which data traffic is sent. Our simulation scenario follows the one in [18]. Each node's radio bandwidth is $2\frac{Mb}{sec}$. In each simulation, data traffic is generated by a number of constant bit rate connections equal to half the number of nodes; no node is a source in more than one connection; no node is a destination in more than three connections. Each source sends four 128-byte data packets each second for 20 seconds. Each simulation lasts for 300 seconds, and data packets are sent at random times between 30 and 270 seconds into the simulation. All other parameters are as in the simulations described above. We vary the number of nodes and the grid's edge length, while maintaining a node density of roughly 75 nodes per square kilometer.

We compare the reliability of Octopus's forwarding sub-protocol with that of two-hop geographic forwarding, which is employed, e.g., by GLS. For both protocols, target locations are discovered using Octopus's location discovery sub-protocol. Fig. 14 shows that the forwarding reliability of the two protocols is



(a) packet overhead



(b) byte overhead

Figure 13: Octopus's overhead for different node densities.

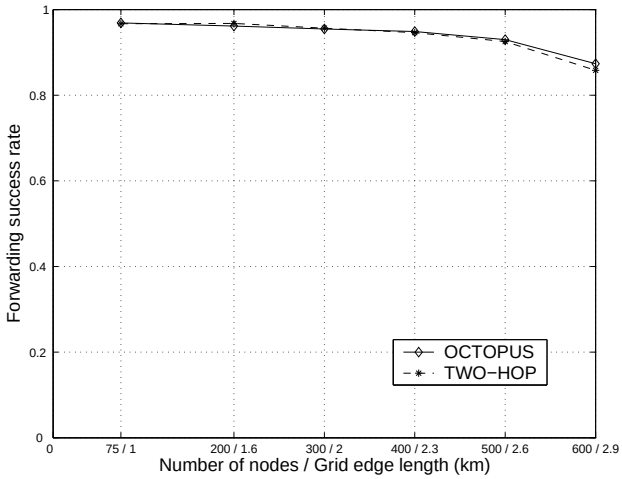


Figure 14: Octopus's data forwarding reliability.

virtually identical. The 95% confidence intervals for the results presented in this figure are up to ±1%. We conclude that the high redundancy of Octopus's location information is an adequate substitute for storing dedicated information for increasing forwarding reliability. Note that the additional overhead for maintaining the two-hop neighbor lists needed for two-hop forwarding is substantial, and it grows with the node density.

6.4 Fault-Tolerance

Octopus's main design goal was to provide high fault-tolerance in the presence of intermittently disconnecting nodes. We now examine whether this design goal is met. To this end, we introduce *unstable* nodes, which alternate between being connected and disconnected [18]. Each time an unstable node awakens, it remains connected for a time interval chosen uniformly at random in the range $[0, 120]$ seconds. And when it disconnects, it remains disconnected for a time interval chosen uniformly at random in the range $[0, 60]$ seconds. Thus, at any given time, an average of $\frac{2}{3}$ of the unstable nodes are connected. We experiment with a varying percentage p of *unstable* nodes. The remaining nodes are connected throughout the simulation. We experiment in a fairly large grid of $2.3km$ by $2.3km$. In order to isolate the effect of node disconnections without impacting the density, we fix the average number of connected nodes at a given time at 400. That is, we run $\frac{400}{1-p+\frac{2}{3}p}$ nodes (e.g., 480 nodes when $p = 0.5$). Note that although the average density of live nodes at any given time is not reduced, it is still challenging to achieve high reliability, since part of the global state is lost with each node disconnect, whereas new nodes connect without any location information. Therefore, protocols that employ low redundancy, e.g., GLS, fail to achieve high routing reliability in the face of disconnects (see Fig 19).

Clearly, location queries for nodes that are disconnected during the location query or shortly beforehand or afterwards are bound to fail. Likewise, nodes that disconnect shortly after issuing a location query will inevitably not receive the query response. We therefore only take into account queries whose target is connected during the interval $[t - 10, t + 10]$ seconds, where t is the query issue time, and whose query source is connected during the interval $[t, t+10]$ (the same approach was taken in [18]). Note that we only require the source and query target to remain

connected– all other nodes, including the target's location servers and the nodes along the search path, can disconnect at any time. A successful query location is followed by the transmission of one 128-byte data packet from the source to the target.

Fig. 15 shows the query success rate and the overall data forwarding reliability as a function of the percentage of unstable nodes. The 95% confidence intervals for the results presented in this figure are up to ±1.4%. We see that Octopus achieves perfect fault-tolerance: its query and forwarding success rates do not degrade at all as we increase the percentage of unstable nodes. This impressive fault-tolerance is achieved thanks to the high level of redundancy in Octopus, and the freshness of the redundant information: Consider a source S issuing a query for a target T. The query succeeds when it reaches a location server in the intersection of S and T's strips. There are at least two such squares (one in S's horizontal strip, and one in its vertical strip). Every 10 seconds, T's location is stored at all the nodes residing in these two squares (since *strip_update_timeout* is 10 seconds). Assuming there are no forwarding holes, as long as one of the nodes in these squares remains connected during the 10 seconds interval, the query should be successful. When the node density is 75, the average population of these two squares is 9.375 nodes. Even when all the nodes in the network are unstable, the probability of all these nodes failing within 10 seconds is negligible. Note also that the probability for holes does not increase when nodes are unstable, since the average node density is fixed. Therefore, Octopus's forwarding reliability does not degrade as we increase the percentage of unstable nodes. This is due to the fact that forwarding failures mainly occur due to holes. In addition, forwarding failures due to node disconnections are usually overcome using retransmissions to alternative nodes.

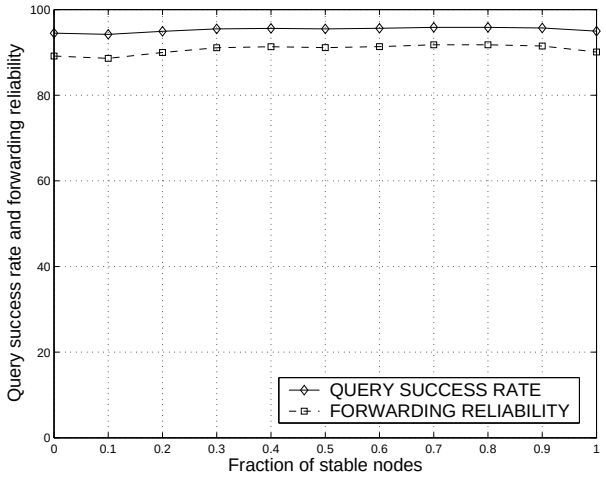


Figure 15: Octopus's fault-tolerance: query success rate and data forwarding reliability are virtually unaffected by the percentage of the unstable nodes.

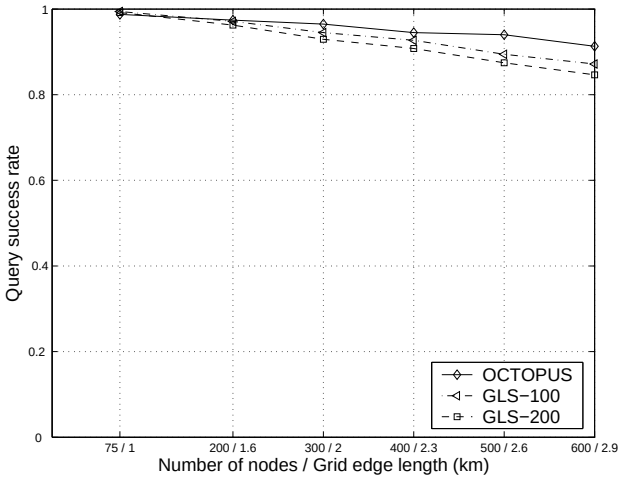
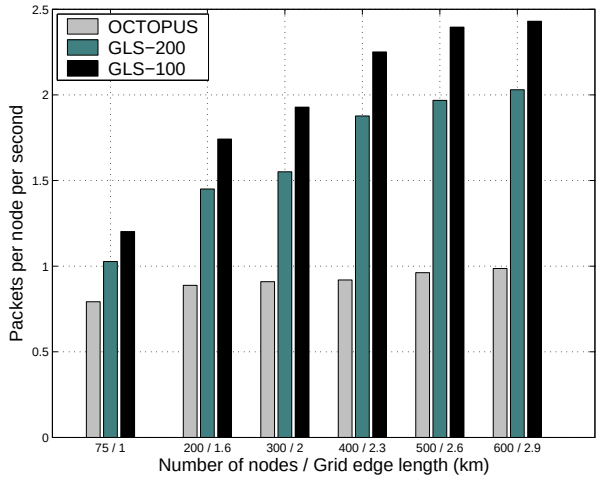


Figure 16: Octopus versus GLS: query success rates.

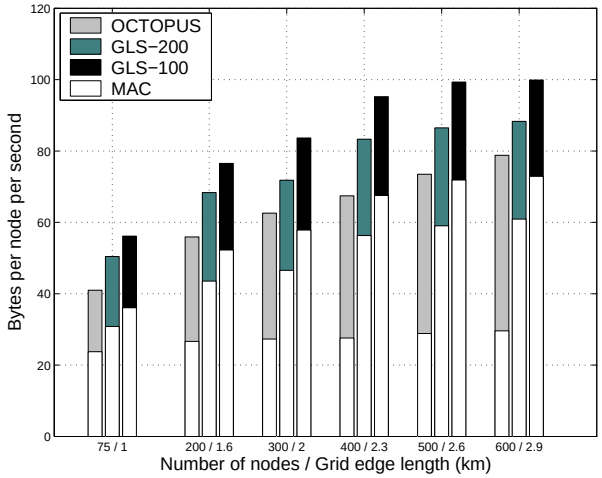
6.5 Comparison with GLS

We now compare the reliability, overhead, and fault-tolerance of Octopus to those of GLS. We use the ns2 implementation of GLS from MIT [1]. In these experiments, we use the grid sizes and densities from GLS's original evaluation [18], with one exception: in the smallest grid ($1km$ by $1km$) we place 75 nodes instead of 100 in order to maintain a similar node density of roughly 75 nodes per square kilometer in all grid sizes. Note that these scenarios are not optimized for Octopus, since most of the grid edge sizes are not multiples of Octopus's strip width ($250m$).

Fig. 16 shows the query success rates of Octopus and GLS. The 95% confidence intervals for the results presented in this figure are up to ±0.8%. GLS-100 and GLS-200 are GLS simulations with a location update threshold of $100m$ and $200m$, respectively. In GLS-d, a node updates its order-i location servers after each movement of $2^{i-2}d$ meters. We see that with either threshold, Octopus achieves similar reliability to GLS in a small network, and better reliability than GLS in medium and large networks. Octopus's advantage is most notable in the largest grid, where Octopus's reliability is roughly 4% and 7% higher than GLS-100's and GLS-200's, respectively. The reliability gap between Octopus and GLS increases with the grid size because of the lower freshness of location



(a) packet overhead



(b) byte overhead

Figure 17: Octopus versus GLS: overhead.

information stored at GLS's remote location servers. Whereas in Octopus, a node updates all its location servers at the same high frequency (every 10 seconds), in GLS, the average frequency at which a node updates its location servers grows with the grid size. For example, in the $2.9km$ by $2.9km$ grid, a GLS-100 node updates its order-4 location servers only after moving 400 meters, and its order-5 location servers after a movement of 800 meters. Thus, a node moving at the average speed ($5\frac{m}{sec}$) updates its order-4 (order-5) location servers only every 80 (respectively, 160) seconds.

Fig. 17 compares Octopus's overhead to that of GLS. The 95% confidence intervals for the results presented in Fig. 17(a) and Fig. 17(b) are up to ± 0.01 packets and 0.1 bytes, respectively. We observe that thanks to aggregation, Octopus sends a smaller number of packets than GLS. Moreover, as the network size grows, GLS's packet overhead increases drastically, while Octopus's packet overhead increases very moderately. This occurs since, as opposed to Octopus, GLS does not employ aggregation, and hence the number of location servers each node needs to update grows with the network size. In addition, the average distance between a node and its location servers also grows with the network size. Although Octopus's location update packets are larger than GLS's, by sending fewer packets, Octopus reduces the number of bytes sent in MAC-level headers. Therefore, overall, Octopus's byte complexity is smaller than GLS's (see Fig. 17(b)). Although GLS's overhead appears to grow more moderately in large networks, this is simply because its reliability drops more sharply in such settings: e.g., in a $2.9km$ by $2.9km$ grid, GLS's reliability drops to only 85%, and therefore many location update and query packets do not reach their destinations, and are hence relayed less times than needed.

Next, we consider simulations with data traffic. In Section 6.3, we showed that the reliability of Octopus's forwarding sub-protocol is similar to the reliability achieved by the two-hop geographic forwarding protocol employed by GLS. We now compare their overhead. We measure the total (data and protocol) packet overhead incurred by both protocols in the simulation scenario of Section 6.3. Fig. 18 shows the average per node per second number of packets sent by Octopus and the more efficient version of GLS, GLS-200. The 95% confidence intervals for the results presented this figure are up to ± 0.01 packets. We do not measure the byte overhead, because it is dominated by the data traffic. As the figure shows, Octopus sends fewer packets than GLS. In addition, Octopus's overhead grows more moderately with the network size than GLS's overhead.

Finally, Octopus's greatest advantage over GLS is its fault-tolerance. In Fig. 19, we contrast Octopus's fault-tolerance against that of the more reliable version of GLS, GLS-100. The 95% confidence intervals for the results presented in both of these figures are up to $\pm 1.4\%$. As explained in Section 6.4, we experiment with an average of 400 connected nodes at a time, on a $2.3km$ by $2.3km$ grid. Whereas Octopus's reliability does not degrade when the percentage of unstable nodes increases, GLS's reliability greatly degrades with the number of unstable nodes: when 50% of the nodes are unstable, GLS's query success rate goes down to less than 65%, and when all the nodes

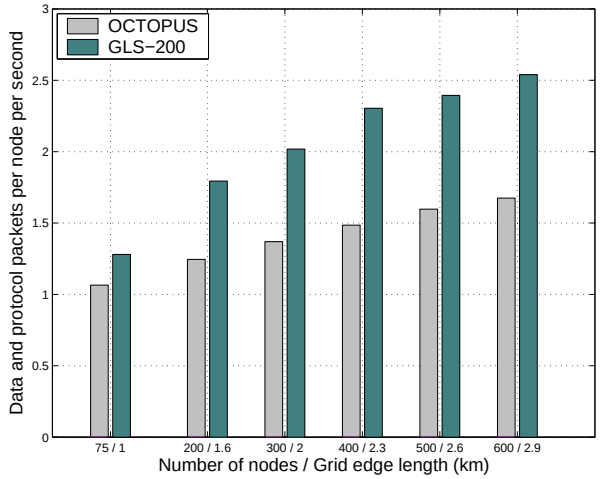


Figure 18: Octopus versus GLS: data and protocol packets sent.

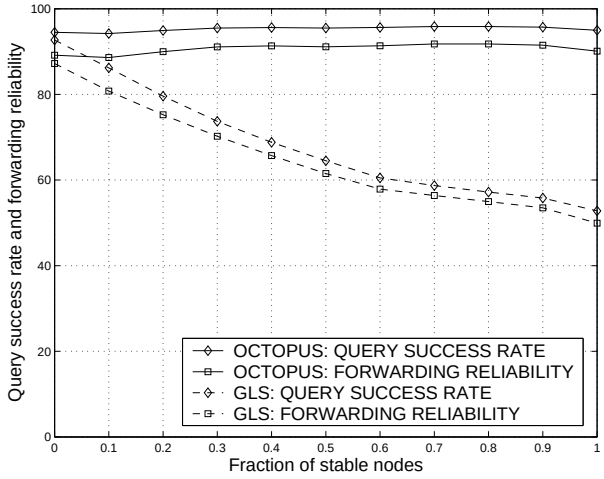


Figure 19: Octopus versus GLS: fault-tolerance.

are unstable, it drops to less than 53%. GLS is less fault-tolerant than Octopus for two reasons: first, GLS employs less redundancy, and second, in GLS, it takes reconnecting nodes a long time to update their remote location servers.

7 Discussion

In this paper, we have shown that, by employing synchronized aggregation, Octopus can enforce higher redundancy and more freshness of location information than previously suggested all-for-some protocols, and hence achieves much better fault-tolerance. In addition, in Section 6.5, we have shown that in a dynamic or failure-prone setting, Octopus incurs lower overhead than an all-for-some protocol that does not employ aggregation. However, Octopus can incur higher overhead than such a protocol in a setting in which the number of location queries, location updates, and failures is low. Therefore, an aggregation-based protocol such as Octopus is suitable for a MANET in which nodes either intermittently disconnect from the network or they constantly move, but it is not suitable for a failure-free static or semi-static network such as a sensor network.

We also note that, as opposed to all-for-some protocols that do not employ aggregation, e.g., GLS [18], Homezone [10, 26], and [27], in Octopus, the size of a location update packet is not fixed and grows logarithmically with the number of nodes in the system (assuming the system model described in Section 3). However, even in a large setting with 675 nodes, the size of a location update packet is no more than a few hundred bytes, which is substantially smaller than the IEEE 802.11's MTU value of 2304 bytes. As we have shown in Section 6.5, thanks to aggregating node locations, Octopus sends a substantially smaller number of packets than GLS, and hence Octopus reduces the number of bytes sent in MAC-level headers. Therefore, the overall number of bytes sent by Octopus is smaller than the overall number of bytes sent by GLS.

8 Conclusions

We have presented Octopus, a simple, fault-tolerant, and efficient routing protocol for large MANETs, which supports movement of the area in which nodes are located. We have proven Octopus's scalability: in Octopus, as opposed to other ad-hoc routing protocols, e.g., [18, 14], the number of location update packets does not increase with the network size. The number of bytes in such packets grows like $O(\sqrt{N})$ with the number of nodes N (and the network size). Empirically, this constitutes a smaller increase in the overhead than exhibited by previous protocols, e.g., [18, 14].

We have conducted thorough empirical evaluation of Octopus using the ns2 simulator with up to 675 mobile nodes. Our extensive simulations have shown Octopus to be scalable, efficient, and have illustrated Octopus's perfect fault-tolerance: in a large grid with hundreds of nodes that intermittently disconnect and reconnect, Octopus achieves the same high reliability as when all nodes are constantly up. At the same time, Octopus incurs less overhead than previous efficient position-based routing protocols. This

is achieved thanks to the use of synchronized aggregation. While we employed aggregation only in the context of location discovery, we believe that similar aggregation can be used to improve the fault-tolerance of various additional protocols and to reduce their overhead, e.g., by aggregating queries or information about various searchable resources in resource location services [3].

Finally, we have introduced a recovery technique that overcomes forwarding failures by using information stored at the location servers. We have shown that the basic geographic forwarding protocol combined with this recovery technique achieves similar reliability to two-hop geographic forwarding, while incurring substantially less overhead.

References

[1] Grid modules for ns2. `http://www.pdos.lcs.mit.edu/grid`

[2] The network simulator - ns-2. www.isi.edu/nsnam/ns/.

[3] I. Abraham, D. Dolev, and D. Malkhi. LLS : a Locality Aware Location Service. In *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing (DIALM-POMC 2004), Philadelphia, Pennsylvania, USA.*, October 2004.

[4] I. Aydin and C. C. Shen. Facilitating matchmaking service in ad hoc and sensor networks using pseudo quorum. In *Proceedings of 11th International Conference on Computer Communications and Networks (ICCCN 2002)*, 2002.

[5] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and B. A. Woodward. A distance routing effect algorithm for mobility (DREAM). In ACM/IEEE MobiCom, Dallas, Texas, 1998.

[6] C. Basile, M.-O. Killijian, and D. Powell. A survey of dependability issues in mobile wireless networks. Technical report, LAAS CNRS, France, February 2003.

[7] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks. In *7th ACM MOBICOM*, Rome, Italy, July 2001.

[8] F. Dai and J. Wu. An extended localized algorithm for connected dominating set formation in ad hoc wireless networks. *IEEE Transaction on Parallel and Distributed Systems*, 15(10):334–347, October 2004.

[9] J. Gao, L. J. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Discrete mobile centers. In *Symposium on Computational Geometry*, pages 188–196, 2001.

[10] S. Giordano and M. Hamdi. Mobility management: The virtual home region, Technical report, October 1999.

[11] Z. J. Haas, J. Y. Halpern, and L. Li. Gossip-based ad hoc routing. In *IEEE INFOCOM 2002*, 2002.

[12] Z. J. Haas and B. Liang. Ad hoc mobility management with uniform quorum systems. IEEE/ACM Trans. on Networking, vol. 7, no. 2, pp. 228–240, Apr 1999.

[13] Z. J. Haas and M. R. Pearlman. The performance of query control schemes for the zone routing protocol. In *SIGCOMM*, pages 167–177, 1998.

[14] P. H. Hsiao. Geographical region summary service for geographical routing. Mobile Computing and Communications Review, vol. 5, no. 4, 2001.

[15] D. Johnson. Routing in ad hoc networks of mobile hosts. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, U.S., 1994.

[16] Y.-B. Ko and N. H. Vaidya. Location-aided routing (LAR) in mobile ad hoc networks. In *Mobile Computing and Networking*, pages 66–75, 1998.

[17] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proceedings of the 11th Canadian Conference on Computational Geometry*, pages 51–54, 1999.

[18] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. A scalable location service for geographic ad-hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 120–130, Aug. 2000.

[19] Q. Li and D. Rus. Communication in disconnected ad hoc networks using message relay. *Parallel Distrib. Comput.*, 63:75–86, 2003.

[20] D. Malkhi. Locality-aware network solutions (a survey). TR 2004-6, School of Computer Science and Engineering, The Hebrew University, 2004.

[21] M. Mauve, J. Widmer, and H. Hartenstein. A survey on position-based routing in mobile ad hoc networks, 2001.

[22] B. Nath and D. Niculescu. Routing on a curve. In *HotNets-I, Princeton, NJ*, 2002.

[23] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM (3)*, pages 1405–1413, 1997.

[24] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[25] C. Perkins, E. Royer, and S. R. Das. Ad hoc on demand distance vector (aodv) routing. internet draft (work in progress), internet engineering task force, october 1999. `www.ietf.org/internet-drafts/draft-ietf-manet-aodv-04.txt`.

[26] I. Stojmenovic. Home agent based location update and destination search schemes in ad hoc wireless networks, Technical report, September 1999.

[27] I. Stojmenovic and P. Pena. A scalable quorum based location update scheme for routing in ad hoc wireless networks. TR 99-09, SITE, University of Ottawa, 1999.

[28] I. Stojmenovic, A. P. Ruhil, and D. K. Lobiyal. Voronoi diagram and convex hull based geocasting and routing in wireless networks. *Wireless Communications and Mobile Computing, Special Issue on Ad Hoc Wireless Networks*, 6(2):247–258, March 2006.

[29] J. Tchakarov and N. Vaidya. Efficient Content Location in Wireless Ad Hoc Networks. In *Proceedings of IEEE International Conference on Mobile Data Management (MDM)*, January 2004.

[30] P. Tsuchiya. The landmark hierarchy : A new hierarchy for routing in very large networks. In ACM Sigcomm, 1988.

[31] E. W. Weisstein. *CRC Concise Encyclopedia of Mathematics, Second Edition.*

[32] J. Wu and H. Li. On calculating connected dominating set for efficient routing in ad hoc wireless networks. *Telecommunication Systems*, Special Issue on Mobile Computing and Wireless Networks (18 1/3):13–36, September 2001.

[33] Y. Xu, J. S. Heidemann, and D. Estrin. Geography-informed energy conservation for ad hoc routing. In *Mobile Computing and Networking*, pages 70–84, 2001.