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.

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 [4, 10]. Thus, an important challenge that ad-hoc routing protocols should address is coping with such failures (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 [11]. Typically, the location of each node is stored at some other nodes, which act as *location servers* for that node [9, 11]. 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 [9].

Position-based protocols differ from each other mainly in how many location servers store each node's location [11]. E.g., in DREAM [3], each node acts as a location server for all nodes, and in LAR [8], each node is a location server for its one-hop neighbors only. It has been argued 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) [9]. Li et al. [9] 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* [11], 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 [9]. However, as we show in Section 5.3, 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 inherent tradeoff between fault-tolerance and load in GLS and other all-for-some protocols, e.g., [7], since fault-tolerance is achieved by constantly updating the location of each node at multiple location servers, which are typically 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 number of relays increases with the network area [9]. In order to reduce the location update overhead, in most all-for-some routing protocols, e.g., [9, 7], remote location servers are updated less frequently than close ones. In Section 5.3, 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 location 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, [9, 7], 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. Moreover, since synchronized aggregation dramatically reduces the location update overhead, Octopus can update all the location servers at the same high frequency.

On the one hand, Octopus enforces higher redundancy and more freshness of location information than GLS, and hence achieves much better fault-tolerance. On the other hand, by aggregating node locations, Octopus incurs lower overhead than GLS in typical scenarios.

Octopus has a third important advantage over most previous all-for-some protocols, e.g., [9, 7]: 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 [9]) for improving the forwarding reliability.

In Section 5, 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.

In the full paper [12], we prove Octopus's correctness, and we also analyze Octopus's scalability: we prove that under a fixed node density, the number of location update packets per node per second is constant, and the byte complexity grows as $O(\sqrt{N})$ with the number of nodes N. We also analyze the probability for forwarding holes in Octopus's horizontal and vertical strips, and show that under reasonable density assumptions, the probability for holes is very small.

2 Related Work

We now compare Octopus to other position-based routing protocols. In DREAM [3], 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 [9]. In LAR [8], each node knows only the locations of its immediate neighbors. 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 are globally flooded [9]. In [6, 15], some dedicated nodes act as location servers for some or all other nodes. This approach is appropriate for failure-free networks, or for settings in which there are reliable servers. However, such an approach is problematic in failure-prone networks, since it is vulnerable to the movement or failure of a single dedicated location server (as explained in [9]).

Li et al. [9] have shown that by having each node act as a location server for some other nodes, one can achieve a good tradeoff between reliability and load, and good scalability up to at least 600 nodes. A similar approach is taken in GRSS [7], Homezone [5, 13], and [14]. 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. [14] 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. However, unlike Octopus, Stojmenovic et al. do not aggregate updates, and they thus miss Octopus's important performance advantage; individually updating so many nodes is bound to induce a prohibitively high overhead. Moreover, Stojmenovic et al. evaluated their protocol in static failure-free settings only. Another difference between Octopus and [14] 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, [14] 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 [9], 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 (see Section 3). Under the same assumptions, Octopus stores the location of each node at $O(\sqrt{N})$ location servers. 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 [7]. 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 causing to many duplicate packets to be sent. Consequently, as shown in [7], 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 [7].

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. We assume that nodes are uniformly distributed in the space. 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. In our simulations, we use the MAC layer provided by the *ns2* simulator, which simulates packet loss in typical MANETs. As in other protocols, 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.

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.

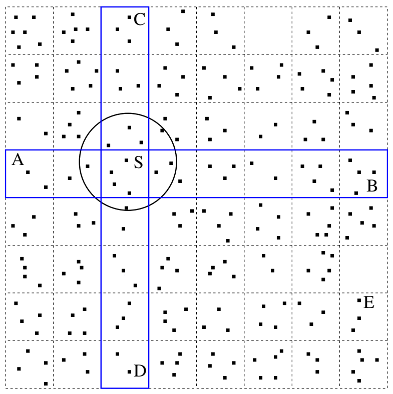


Figure 1. Node S's neighbors and strips. A, B, C, and D are end nodes in the highlighted strips.

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.

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, as if B does not exist, and re-sends the packet to C. Up to two retransmissions per packet are sent.

4.1 Location Update

The location update protocol is initiated 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.

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, and p_x, p_y – the node's previous coordinates.

```
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 $p\_y$⟩
Data structures
Node $this$ – this node.
Set of Node $neighbors$, $strip[4]$, $target\_locations$.
```

Figure 2. Types and data structures.

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.

```
loop forever
    foreach Direction $d$ do
        if (I have no neighbors in direction $d$) then
            StripID $sid$ ← get_strip_id ($d$)
            set of Node $set$ ← get_nodes_in_strip ($sid$)
            bcast ⟨STRIP_UPDATE, $sid$, opposite direction to $d$,
                $set$, farthest node in $set$⟩
    sleep (strip_update_timeout)

Event handler:
upon receive ⟨STRIP_UPDATE, $sid$, $d$, $set$, $t$⟩ do
    if ($sid = this.vid \vee this.hid$) then
        $strip$[opposite direction to $d$] ← $set$
    /* If I am the packet target */
    if ($this = t$) then
        set of Node $set'$ ← get_nodes_in_strip ($sid$)
        Node next ← farthest node
                    in direction $d$ in $set' \cup \{this\}$
        /* If I need to forward the packet */
        if ($this \neq next$) then
            bcast ⟨STRIP_UPDATE, $sid$, $d$, $set \cup set'$, $next$⟩

Procedures:
set of nodes get_nodes_in_strip ($sid$)
    return $\{n \in neighbors | n.hid = sid \vee n.vid = sid\}$

StripID get_strip_id ($d$)
    if $d \in \{north, south\}$ then
        return $this.vid$
    return $this.hid$
```

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 north of node E. In a typical scenario, the probability for a forwarding hole is small (less than 0.02, see the full paper [12]). 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.

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 that resides in an adjacent strip to s. Empirically, the additional reliability achieved by this mechanism is negligible (less than 2%), since the basic Octopus's implementation already achieves high reliability. Therefore, for simplicity reasons, we present and evaluate Octopus without the bypass mechanism.

In the full paper [12], we prove the following lemma: In a run in which there are no node movements or failures and no packet loss, if the strip width $w \leq \frac{\sqrt{3}}{2}r$, then in every segment of a strip in which there are no forwarding holes, every node knows the identities and locations of all the nodes that reside in this segment after the propagation of STRIP_UPDATE packets in this segment completes.

4.2 Location Discovery

The location discovery protocol uses the information stored in *strip[*]* and *neighbors*, as well as the set *target_locations*, which is a cache of recently discovered target locations. The cache entries expire after *strip_update* seconds. The code for this protocol is presented in Fig. 4.

The interface to the location discovery protocol consists of the function *locate*, which first searches the target in one of the locally maintained sets (*strip[*]*, *neighbors*, and *target_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 *target_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 $target \leftarrow$ search_locally ($tid$)
    **if** ($target = null$) **then**
        search_location ($this, tid, north$)
        search_location ($this, tid, south$)
        sleep (discovery_timeout)
        **if** ($target \notin target\_locations$) **then**
            search_location ($this, tid, west$)
            search_location ($this, tid, east$)

Event handlers:
**upon** receive $\langle$QUERY, $src, t\_id, d, next\rangle$ **do**
    **if** ($next = this$) **then**
        search_location ($src, t\_id, d$)

**upon** receive $\langle$REPLY, $src, target, d, next\rangle$ **do**
    $target\_locations \leftarrow target\_locations \cup \{target\}$
    **if** ($next = this$) **then**
        send_reply ($src, target, d$)

Macro:
strip_neighbors[$d$] $\triangleq$ ($neighbors \cap strip[d]$) $\cup \{this\}$

Procedures:
**Node** search_locally ($target\_id$)
    **if** ($\exists n\ s.t.\ n \in neighbors \cup strip[*] \cup target\_locations$
        $\wedge n.id = target\_id$) **then**
        **return** $n$
    **return** $null$

search_location ($src, t\_id, d$)
    Node $target \leftarrow$ search_locally ($t\_id$)
    **if** ($target = null$) **then**
        Node $next \leftarrow$ farthest node in strip_neighbors[$d$]
                        in the same square as $this$
                        or in an adjacent square
        **if** ($next \neq this$) **then**
            bcast $\langle$QUERY, $src, t\_id, d, next\rangle$
    **else if** ($src \neq this$) /* target found - send reply */
        Direction $d' \leftarrow$ opposite direction to $d$
        send_reply ($src, target, d'$)

send_reply ($src, target, d$)
    Node $next \leftarrow$ closest node to $src$ in strip_neighbors[$d$]
    **if** ($next \neq this$) **then**
        bcast $\langle$REPLY, $src, target, d, next\rangle$
```

Figure 4. The location discovery protocol.

In the full paper [12], we prove the following lemma: Assume that there are no node movements, node disconnections, or packet loss, and that $w \leq \frac{\sqrt{3}}{2}r$. 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. 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 *target_locations* eventually includes T's location.

4.3 Forwarding protocol

Upon a successful location discovery, the forwarding protocol forwards data packets 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 packets.

Octopus employs geographic forwarding [11] 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 geographically 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. 5, 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 5, we show that one-hop geographic forwarding using this recovery technique is very effective, achieving the same reliability as two-hop geographic forwarding. The pseudo-code of the forwarding protocol appears in Fig. 6.

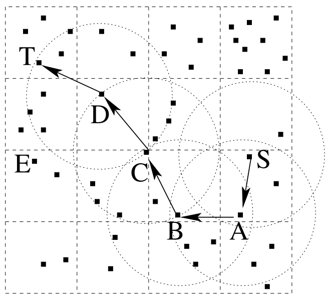


Figure 5. Octopus's forwarding protocol.

```
forward (Packet $p$, Node $target$)
    Node $next \leftarrow$ closest node to $target \in neighbors \cup \{this\}$
    if ($next = this$) then
        $target' \leftarrow$ closest node to $target$ from $strip[*]$
        $next \leftarrow$ closest node to $target'$ from $neighbors$
    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 ($p$, $target$)
```

Figure 6. The forwarding protocol.

5 Evaluation

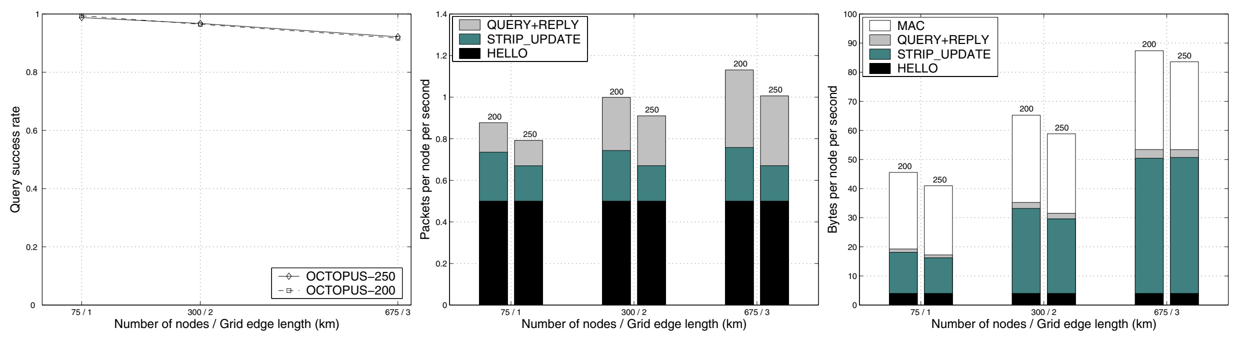
We now evaluate Octopus using simulations. Octopus is implemented in *ns2* [2] with CMU's wireless extensions. Each node uses the IEEE 802.11 radio and MAC model provided by the CMU extensions, with a radio range r of 250 meters and a throughput of $1\frac{Mb}{sec}$. The nodes are initially placed uniformly at random in a square universe. In most of our simulations, there are 75 nodes per square kilometer. (Li et al. [9] 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 [9]: 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 begins moving toward it immediately in 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 described below is due to the large number of events in each simulation.

In Section 5.1, we discuss our choice of the protocol's parameters. In Section 5.2, we examine Octopus's scalability as the number of nodes and network area increase. In Section 5.3, we study Octopus's fault-tolerance. In Section 5.4 we evaluate the reliability of Octopus's forwarding sub-protocol and compare it with two-hop geographic forwarding. Finally, in Section 5.5, we compare Octopus's reliability, overhead, and fault-tolerance to those of GLS.

5.1 The choice of parameters

Each node broadcasts a HELLO packet every 2 seconds, as in GLS [9]. We chose this frequency in order to allow a fair comparison between the two protocols. Nevertheless, in experiments with update frequency of up to one in five seconds, the results are virtually identical, 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, increasing this frequency, e.g., to one in 5 seconds, results in a negligible increase in the protocol's



(a) query success rate (b) packet overhead (c) byte overhead

Figure 7. Octopus's query success rates and overhead for different strip widths.

reliability. On the other hand, reducing this frequency to one in 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 [8]. 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 the full paper [12], we prove 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 the full paper [12]), 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. Fig 7(a) 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-200 (where $w = 200$). The 95% confidence intervals for the results presented in this figure are up to $\pm0.8\%$ of the average value. In order to ensure a fair comparison, we examine grid edge lengths that are divisible by both 250 and 200. 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. 7(b) and Fig. 7(c) show that increasing the strip width from $200m$ to $250m$ reduces the per node packet and byte complexities of Octopus. Henceforth, we fix the strip width at $250m$. The 95% confidence intervals for the results presented in Fig. 7(b) and Fig. 7(c) are up to ±0.01 packets and ±0.1 bytes of the average value, respectively.

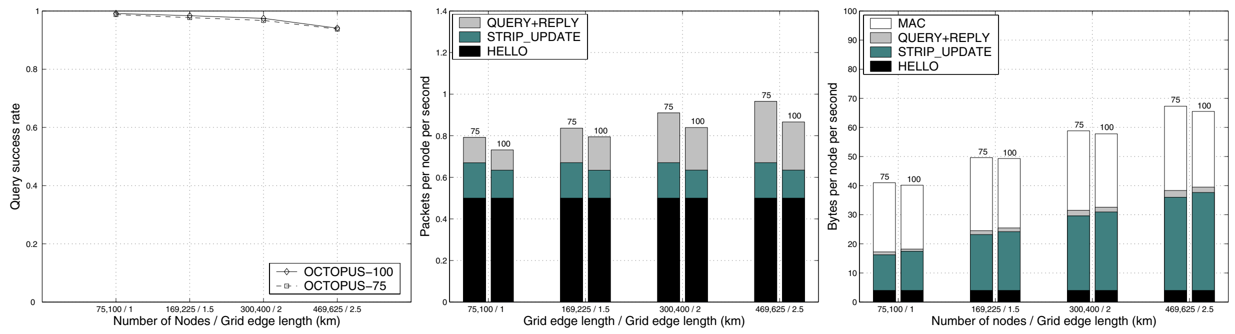
5.2 Scalability

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

5.2.1 The effect of node density

We now examine what happens when the node density increases from 75 to 100 nodes per square kilometer. Fig. 8(a) shows that the query success rate remains similar. This occurs because of two opposing tendencies: On 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. 8(a) are up to $\pm1\%$ of the average value.

In Fig. 8(b) and Fig. 8(c), 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



(a) query success rate (b) packet overhead (c) byte overhead

Figure 8. Octopus's query success rates and overhead for different node densities.

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. 8(b) and Fig. 8(c) are up to ± 0.01 packets and ± 0.1 bytes of the average value, respectively.

5.2.2 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.

We further observe that the number of location update packets sent by Octopus is constant, matching the analysis in the full paper [12]. The overall 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 [9, 7, 5, 13], 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.

5.3 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 [9]. 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 9).

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 [9]). 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.

Fig. 9 shows the query success rate as a function of the percentage of unstable nodes. The 95% confidence intervals for the results presented in this figure are up to $\pm 1.4\%$. We see that Octopus achieves perfect fault-tolerance: its reliability does 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.

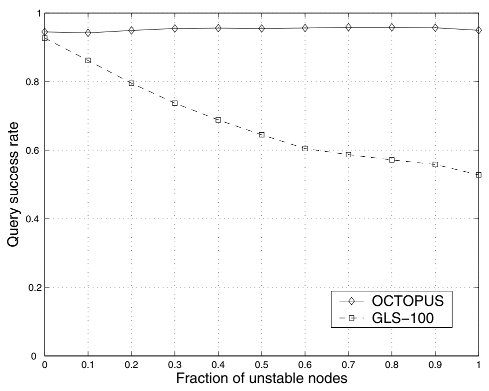


Figure 9. Query success rate as a function of the percentage of the unstable nodes.

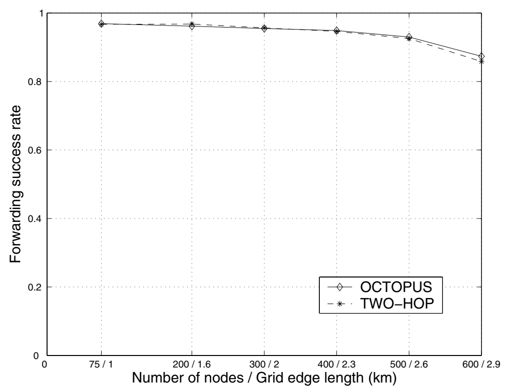


Figure 10. Data forwarding reliability.

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 density is fixed.

5.4 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 [9]. 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. 10 shows that the forwarding reliability of the two protocols is virtually identical. The 95% confidence intervals for the results presented in this figure are up to $\pm1\%$. 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.

5.4.1 Fault tolerance

In Section 5.3, we have evaluated Octopus's query success rate in a failure-prone setting, in which nodes intermittently disconnect and reconnect. We now evaluate Octopus's overall routing reliability in the same setting. We repeat the experiments of Section 5.3 with a single exception: a successful query location is followed by a sending of one 128-byte data packet from the source to the target. Octopus forwards data packets using its previously described forwarding protocol. Fig. 11 shows 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 $\pm1.4\%$. Note that the overall routing reliability achieved by Octopus is very close to its query success rate, since the forwarding reliability in this failure-prone setting is virtually identically to the forwarding reliability achieved in failure-free settings. This is due to the fact that forwarding reliability is mainly dominated by the probability for holes, which does not increase when nodes are unstable, since the average density is fixed. In addition, forwarding failures due to node disconnections are usually overcome using retransmissions to alternative nodes.

5.5 Comparison with GLS

We now compare the reliability, overhead, and fault-tolerance of Octopus to those of GLS. We use the ns2

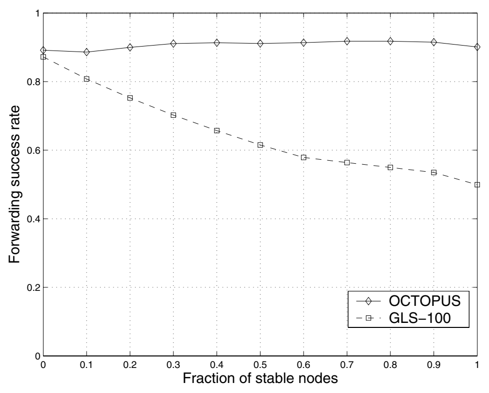


Figure 11. Data forwarding success rate as a function of the percentage of the unstable nodes.

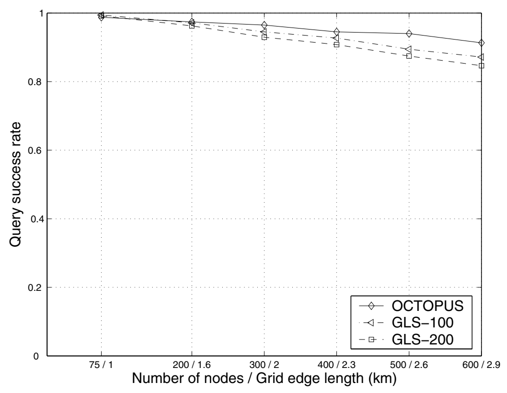
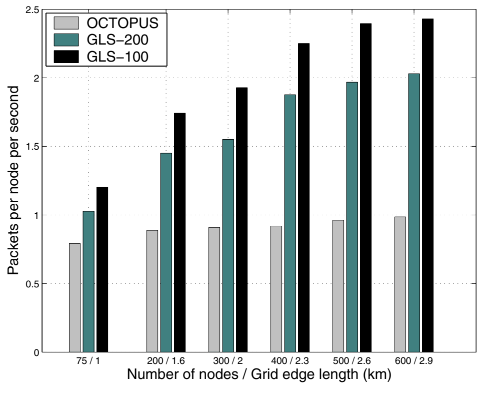


Figure 12. Octopus versus GLS: query success rates.

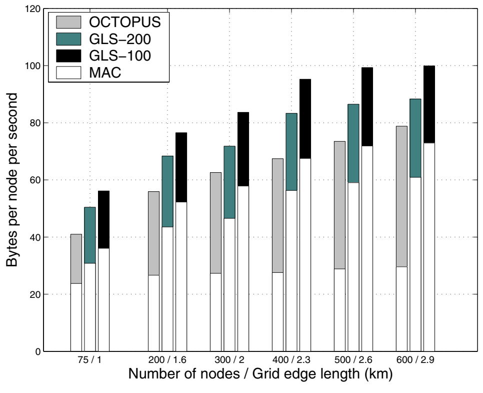
implementation of GLS from MIT [1]. We use the grid sizes and densities from [9], with one exception: in the smallest grid ($1km$ by $1km$) we place 75 nodes instead of 100 in order to maintain a similar node density of roughly 75 nodes per square kilometer in all grid sizes. Note that these scenarios are not optimized for Octopus, since most of the grid edge sizes are not multiples of Octopus's strip width ($250m$).

Fig. 12 shows the query success rate for Octopus and GLS simulations. The 95% confidence intervals for the results presented in this figure are up to $\pm 0.8\%$. 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 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. 13(a) and Fig. 13(b) compare Octopus's overhead to that of GLS. 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, respectively. Thanks to aggregation, Octopus sends a smaller number of packets than GLS. Furthermore, as opposed to Octopus, in GLS, the message complexity incurred by the location update protocol grows with the grid size, as on average each location update packet is relayed more times. 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. 13(b)).



a) packet overhead



b) byte overhead

Figure 13. Octopus versus GLS: packet and byte overhead.

Octopus's greatest advantage over GLS is its fault-tolerance. In Fig. 9 and Fig. 11, we contrast Octopus's fault-tolerance against that of the more reliable version of GLS, GLS-100. As explained in Section 5.3, 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. 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.

Finally, we consider simulations with data traffic. In Section 5.4, 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 measure the total (data and protocol) packet overhead incurred by both protocols in the simulation scenario of Section 5.4. Fig. 14 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.

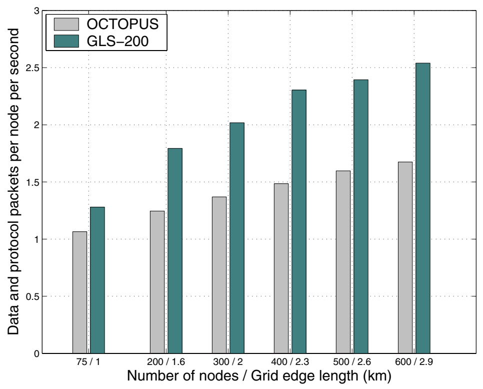


Figure 14. Octopus versus GLS: data and protocol packet overhead.

6 Conclusions

We have presented Octopus, a simple fault-tolerant and efficient routing protocol for large MANETs. We have proven Octopus's scalability: the number of location update packets does not increase with the network size, and the number of bytes in such packets grows like $O(\sqrt{N})$. Our extensive simulations 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. Nevertheless, Octopus incurs less overhead than previous efficient position-based routing protocols. This is achieved thanks to the use of aggregation.

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] 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.

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] R. Melamed, I. Keidar, and Y. Barel. Octopus: A fault-tolerant and efficient ad-hoc routing protocol. TR, department of Electrical Engineering, Technion, April 2005. http://www.ee.technion.ac.il/people/idish/ftp/octopus-tr.ps.

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

[14] 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.

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