REIHE INFORMATIK
7/2002

A Simulation Study of a Location Service for Position-Based Routing in Mobile Ad Hoc Networks

Hannes Hartenstein, Michael Käsemann
NEC Network Laboratories
NL-E Heidelberg
Adenauerplatz 6
D-69115 Heidelberg

Holger Füßler, Martin Mauve
Universität Mannheim
Praktische Informatik IV
L15, 16
D-68131 Mannheim

A Simulation Study of a Location Service for Position-Based Routing in Mobile Ad Hoc Networks[1]

Michael Käsemann⋆, Hannes Hartenstein⋆, Holger Füßler•, Martin Mauve•

⋆NEC Europe Ltd. Network Laboratories
Adenauerplatz 6
69115 Heidelberg, Germany
{Michael.Kaesemann|Hannes.Hartenstein}@ccrle.nec.de

•Praktische Informatik IV, Universität Mannheim
L 15,16
68161 Mannheim, Germany
{fuessler|mauve}@informatik.uni-mannheim.de

Abstract: Position-based routing in a mobile ad hoc network requires geographic addresses. Thus, a node that wants to send a packet to some target node has to know the target's (approximate) current position. In order to provide each node's position to the other network nodes, a distributed location service has to be used. J. Li et al. recently put forward a promising approach called the 'Grid Location Service' (GLS). In this paper we provide some analyses and evaluations of GLS by means of simulation with *ns-2* beyond the results of the original paper. We present quantitative results with respect to location query failure rate and bandwidth consumption. We analyze in detail why queries failed and how the query failure rate can be decreased for scenarios with a low density of nodes.

1 Introduction

Routing in a mobile ad hoc network is a demanding task since the network's topology is changing frequently. At the same time, the bandwidth consumption of routing-related messages should be low since the overall available bandwidth is rather limited.

'Traditional' IETF Manet [Man] protocols have either followed distance vector or link state ideas in case of so-called 'pro-active' approaches, or have employed on-demand protocols where a route is only established when there is an actual need for this specific route. On-demand approaches are generally considered to be more scalable than pro-active ones for mobile ad hoc networks when it can be assumed that not all nodes communicate with all other nodes all the time.

While known for a long time (see, e.g. [TK84]), position-based routing has recently received significant attention [BBC+01, BMSU99, Kar00] for ad hoc networking. One main reason for this is that today it is possible for nodes to know their geographic positions, e.g., by means of GPS. The motivation to use position-based routing is that it eliminates the need to maintain routes and therefore is very well suited for highly mobile networks. In order to forward a packet a node using position-based routing only needs information about the immediate neighborhood and the position of the destination. A survey on position-based routing for mobile ad hoc networks is given in [MWH01].

For position-based routing one assumes that nodes are addressed by a node identifier (ID) together with the node's current geographic position. A forwarding node makes use of the target's position given in the packet header as well as of some locally available information: the forwarding node's own position and knowledge of the positions of its 1-hop neighbors. With a greedy forwarding strategy, the forwarding node selects as next hop the 1-hop neighbor that lies in the direction of the target and results in the 'most forward progress'.[2]

Note the main difference between traditional topology-based routing and position-based routing: with position-based routing there is no route setup or maintenance, forwarding is done 'on-the-fly'.

[1]This report is a revised and extended version of the paper with the same title that we published in Proceedings of WMAN 2002, GI-Edition Lecture Notes in Informatics, March 2002.

[2]When there is no 1-hop neighbor in the direction of the target, a 'recovery strategy' has to be employed.

However, as an essential prerequisite for position-based routing, a *location service* is needed from which a node can learn the current position of its desired communication partner. A survey on various distributed location services is also presented in [MWH01]. Clearly, in order to learn the position of another node one could flood the whole network as with a route discovery of an on-demand approach. Alternatively one could let each node flood its current position in regular intervals as a 'pro-active' approach. Of course, the optimum is likely to lie somewhere in the middle of both approaches. One proposal where each node sends its current position only to a small subset of all nodes has been put forward by J. Li et al. under the name of Grid Location Service (GLS) [LJC$^+$00]. While in our opinion GLS appears to be a promising candidate for a scalable distributed location service, the approach shows some degree of complexity and requires a deeper analysis and evaluation than has been done before. In particular, we would like to obtain answers to the following questions:

- How accurately does GLS manage locations?
- When a location query fails, what is the reason?
- What is the exact amount of signaling load that is required to provide sufficiently up-to-date information on the position of nodes?
- What density of nodes is required for successful greedy forwarding? In other words, when would position-based routing as well as a location service benefit from a more advanced forwarding strategy?
- How is the performance of GLS affected when the degree of mobility of the nodes is increased?

In this paper we present results derived from simulations with *ns-2* [ns] that answer the above questions. In Section 2 we first give an introduction to GLS. In Section 3 we outline our simulation scenarios and present results for *a)* a detailed analysis of location query failure rate vs signaling load, *b)* the impact of the density of nodes on the performance of GLS and greedy forwarding in general and *c)* the impact of the mobility of nodes on the same. Section 4 concludes the paper, while Appendices A to F contain detailed information on the simulation environment.

2 Grid Location Service

GLS structures an ad hoc network's area by using a fixed grid whose grid lines are known to all nodes in the network. It is assumed that each node in a cell of the grid knows about all other nodes in the same cell. This is achieved by maintaining 1- and 2-hop neighbor lists at each node and by choosing an appropriate cell size with respect to the transmission range of the nodes. The neighbor lists are built from periodic 'hello' broadcasts that are sent by all nodes and indicate a node's location as well as other parameters like the node's 1-hop neighbors. Thus, in order to reach a node one only has to reach the corresponding cell or, in GLS terminology, the node's 'location'. Note that the location of a node, therefore, represents 'quantized' positional information in contrast to the actual geographical position.[3]

GLS represents a fully distributed location service, i.e., each node in the network maintains a part of the overall location database. In this service a node is called a location server of another node if it stores the mapping of this node's ID to its location. The main problem that has to be solved in such a system is how to find one or more location servers for a node with a certain ID. This mapping is needed in two situations: *i)* when a node changes its position it needs to store the new position in *all* of its location servers and *ii)* when a node needs the position of a communication partner it has to find at least *one* of that node's location servers. In the following we first present the basic GLS concepts inspired by 'consistent hashing' and afterwards give some more technical details.

GLS adheres to the following two design guidelines: *i)* no flooding should be used, and *ii)* the density of a node's location server should be decreasing with an increase in distance to the node. These constraints are imposed for scalability reasons. The design guidelines are fulfilled by superimposing a quadtree on top

[3]Greedy forwarding can now proceed as follows: first, 'candidate locations' in the direction of a target's location are selected; then, within these locations the node with the 'best' geographic position is chosen as next hop.

of the grid structure (Figures 1 and 2). The original grid cells are now called 1-order squares, the cells of the next upper level are called 2-order squares, and so on. For a node A, GLS selects one location server per n-order square that is in the same $n+1$-order square as node A if this n order square does not already contain a location server for A from a previous step of the recursion (see example below).

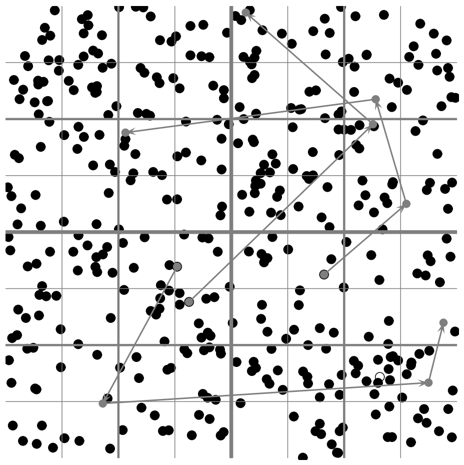


Figure 1: GLS grid structure as well as movement paths for 3 selected nodes.

The selection of location servers is based on node identifiers (IDs consisting of a natural number). A node A's location server in a specific square is determined by a hash function that takes as input the IDs of all nodes in the respective square as well as the ID of node A: the location server for node A in an n-order square is the one that has the 'nearest' ID to node A's ID in the respective n-order square. The nearest ID to node A's ID is the least ID larger than node A's ID.[4] The hash function is designed in a way that *each* node can make use of the hash function for a specific square and target ID in order to find the location servers of the target ID. The location server selection process is best explained using an example depicted in Figure 2 (taken from [MWH01]). For node 10 the selected location servers in the three surrounding first order squares are nodes 15, 18, 73. In the surrounding three second-order squares, again the nodes with the nearest ID are chosen to host the node's location; in the example these are nodes 14, 25, and 29. This process repeats until the area of the ad-hoc network is covered. The 'density' of location information about a given node thus decreases logarithmically with the distance to that node.

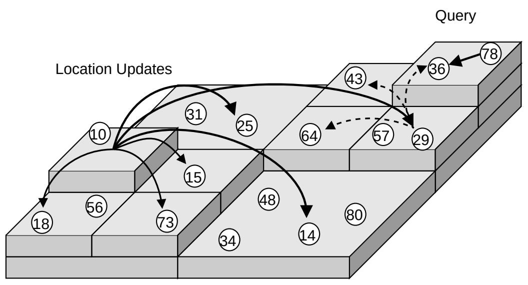


Figure 2: GLS example.

In order to explain how a node can find a location server for the mapping of an arbitrary node ID to that node's location, we use again the example given in Figure 2. Assume that node 78 wants to obtain the location of node 10. It therefore should locate a 'nearby' node that knows about the location of node 10. In the example this is node 29. While node 78 does not know that node 29 holds the required position, it is

[4]ID numbers wrap around after the highest possible ID.

able to discover this information. To see how this works, it is useful to have a look at the location servers for node 29. Its location is stored in the three surrounding first order squares in nodes 36, 43, and 64. Note that each of these nodes and node 29 are automatically also the ones in their respective first order square with the nearest ID to 10. Thus, there exists a ‘trail’ with descending node IDs to the correct location server from each of the squares of *all* orders. Location queries for a node can now be directed to the node with the nearest ID the querying node knows of. In our example this would be node 36. The node with the nearest ID does not necessarily know the sought-after node, but it will know a node with a nearer node ID (node 29, which already is the sought-after location server). The process continues until a node that has the location information available is found.

When a location server is found that has the requested information, the location query is forwarded to the queried target node. The target node itself sends back a query response to the query originator indicating the current location.

After having explained the basic concepts of GLS we like to point out some technical details. It is important to note that a node does not need to know the IDs of its location servers, which makes a bootstrapping mechanism that discovers a node’s location servers unnecessary. A node sends out location update messages towards the respective *n*-order squares for which a location server has to be selected. Within such a square, location update messages are forwarded to nodes with nearer IDs in a process closely resembling location queries that are explained above. The only difference is that information is written instead of read, ensuring that the location information reaches the correct node where it is then stored. Proof of correctness can be done in an inductive fashion (see the original paper on GLS [LJC$^+$00]). In general, forwarding decision are based either on locations or on node ID’s: for example, when sending a location update, this messages is first forwarded to the respective square it is intended for by means of position-based forwarding. Within the target square the update is then forwarded with respect to node IDs.

Clearly, the ‘consistency’ of the location server selection might suffer from mobility. When a node moves into a new grid cell, information in the node’s location servers is outdated. Furthermore, since the moving node also acts as location server for some other nodes, there might be no location server for these nodes in the square the moving node has just left. Thus, there is a clear trade-off between bandwidth consumption of GLS signaling messages (in particular, location updates) and the success rate of location queries. This trade-off is studied in detail in the following section.

A trigger for sending a location update is based on the ‘update distance’ parameter d that indicates the distance a node might travel until a location update has to be sent again. In particular, for $i > 1$ a node updates each ith order location server after a movement of $2^{i-2}d$ since sending the last update.

To alleviate the problem of looking for a node that has just moved to a neighboring cell, the concept of a ‘forwarding pointer’ has been introduced. The forwarding pointer consisting of a node ID and its new location is broadcasted by a node shortly after the node moved into the new cell. Nodes of the previous cell that receive the ‘forwarding pointer update’ append it to their hello message.

To improve the performance of location queries, each node maintains location information it has obtained from forwarded packets in a ‘location cache’.

For more details on GLS the reader is referred to [LJC$^+$00].

3 Simulation Results

We ported the original *ns-2* modules [gri] to the ns version *2.1b8a* and extended the modules for our measurements. In particular, we added:

- Support for detailed location query failure analysis. We can check the specific reason why a query was not successfully completed.
- Support for deviation analysis. When a location query ‘meets’ a location table/cache with an entry for the query’s target, we measure the distance between the target’s current position and the target’s position when the entry in the table/cache was created.
- Support for accurate bandwidth measurements. For the bandwidth measurements we set the header types and field sizes as given in Tables 1 and 2.

Table 1: Fields and their respective sizes in GLS packet headers used in our bandwidth calculations. The Table length parameters refers to the size of the Neighbor Table and Forwarding Pointer Table in a Hello packet.

Field type	**Size in bytes**
Identifier (ID)	3
Location (LOC)	2
Speed	1
Position (X,Y)	2
Timestamp	2
Timeout value	2
Packet type and flags	1
Table length	2

Table 2: The main header types of GLS and their respective fields as used in our experiments. The forwarding pointer update is the message sent by a moving node in order to inform its previous cell about its new location.

Header type	**Fields** All packet headers include the packet type and flags as well as a timestamp field (contributing 3 bytes). Additional fields:	**Size** (Bytes)
HELLO	Source LOC, Source ID, Source (X,Y), Table length, Neighbor table, Forwarding Pointer Table (entries in a table consist of ID and LOC)	>10
LOCATION UPDATE	Source LOC, Source ID, Destination LOC, Location Timeout	12
LOCATION QUERY	Source LOC, Source ID, Destination ID, Next LOC, Next ID, Target Data Timestamp	18
QUERY RESPONSE	Source LOC, Source ID, Destination LOC, Destination ID	13
LOCATION NOTIFICATION	Source LOC, Source ID, Destination LOC, Destination ID	13
FP UPDATE	Source LOC, Source ID, Destination LOC	10

- Support for forwarding analysis. If a packet does not make it to its destination, we can check whether *i)* there exists a theoretical path to the destination and *ii)* whether there exists a 'greedy' path to the destination.

- Enhanced tracing facilities.

In all simulations we make use of the random waypoint mobility model [BMJ+98] as movement pattern. In this model each node randomly selects a waypoint in the area that contains the network and moves from its current location to the waypoint with a random but constant speed. Once a node has arrived at the target waypoint it immediately selects a new waypoint and speed and the process continues. IEEE 802.11 with 1Mbits/s is the simulated protocol for the wireless communication. The transmission range is set to 250m. The size of the first order squares is set to 250m $\times$ 250m. The update distance d is set to 100m. All nodes move with a maximum speed of 10 m/s (average 5 m/s). The location queries are generated by randomly selecting querying nodes and target IDs with a uniform distribution.

3.1 Location query failure rate vs costs

In this section we study how often location queries succeed or fail, the reasons for query failures, and what amount of GLS signaling is needed to achieve the observed location query success rate. We first study a scenario of size 2km $\times$ 2km populated with 400 mobile nodes, thus, with a density of 100 nodes per square kilometer as recommended in the original GLS paper. In the subsequent section we will vary the density of nodes and study the corresponding effects. For all results described below an average of 5 runs was taken.

Table 3: Query analysis.

Event	Number of packets
Total number of queries	6000
Number of queries not resolved locally	4598
Number of queries received by appropriate location server	4311
Number of queries received by destination	4233
Number of successful queries	4170
Number of lost queries	428

Table 4: Shown are the sources of information for successful responses on location queries together with the amount of queries that made use of the respective source, the deviation of the target's current position compared to its position when the target's location information was inserted in the cache/table, as well as the age of the entry that was used for answering the query. Note that the deviation information is external to GLS but measured by the simulation environment.

Source of response	Proportion (%)	Deviation in m	Age in s
2-Hop Neighbor Table	20	7.58	0.44
Location cache (local)	2	122.76	15.66
Location table (local)	1	263.37	92.95
Location cache (foreign)	32	115.78	14.78
Location table (foreign)	45	237.79	83.84

From the results presented in Table 3 we see that about 1400 location queries out of a total of 6000 queries were answered by the querying node itself, i.e., via its neighbor table, location table, or location cache. Only the remaining 4598 queries were actually sent to the network. Losses of queries occur between requesting node and location server, between location server and target destination, and between destination and requesting node. About 7.1% of all queries, or 9.3% of all queries actually sent, were lost.

We classify the reasons for a loss of a location query into 4 classes.

- No route: GLS forwarding fails since at some forwarding node there is no option for a next-hop node.
- TTL expired: The packet either travels for too long or enters a routing loop.
- No location server found: no location server for the requested target node can be found since no node with a 'better' ID is known.
- Location error: a location for the target node is determined but the target node is no longer there and cannot be reached.

Our simulations results show that about 15% of the lost queries suffered from 'No route' and about 40% were classified as 'TTL expired'. About 40% of the lost queries could not be resolved since no appropriate location server could be found. Finally, about 5% of queries failed due to a location error.

An analysis of the successful queries as presented in Table 4 shows that most queries are resolved either using the local 2-hop neighbor table or a foreign location table or cache. The average deviation gives the deviation in meters between the current position of the target node and the target's position when the corresponding table/cache entry was created. When we assume that a node that has traveled no further than approximately 250m since its previous location update might still be reachable, we see that the location tables and caches are sufficiently 'up-to-date' on average. For a further study, we like to investigate to what degree the number of hello messages and location updates can be reduced without significantly affecting performance.

The signaling load in the current study is as follows. There were 38790 updates sent with a size of 12 bytes. Each update traveled 3.1 hops on average. Only 1.4% of the updates were lost. Thus, when we compute the average number of kilobits per second per node, we obtain an average load of about 0.1 kbps per node for the location updates. The hello packets are of size 165.22 bytes on average due to the neighbor and forwarding pointer tables. A total of 60184 hello messages where transmitted, thus, the hello messages contribute 0.66 kbps per node. Thus, in total the signaling overhead is about 0.8 kbps per node.

3.2 Impact of Node Density

We now study the impact of a varying node density on GLS performance, i.e., we are interested to assess how the various performance parameters scale with the number of nodes. Furthermore, since ad hoc networks are based on the principle of mutual assistance, it is interesting to know how many nodes have to participate in GLS in order to succeed. For this reason, we varied the number of nodes between 100 and 400 for the scenario of the previous section. Figure 3 shows the amount of queries sent to the network as well as the amount of queries that failed compared to all submitted queries. The portion of queries that were not resolved locally (compared to all location queries) keeps constant, while the amount of query failures compared to all queries increases significantly with a decreasing number of nodes.

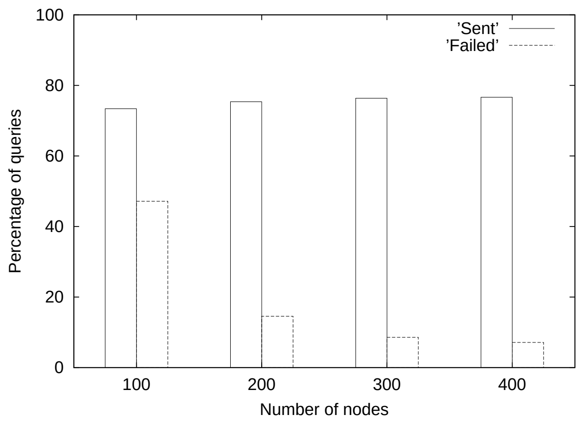


Figure 3: Shown is the amount of queries sent to the network as well as the amount of queries that failed, both in comparison to the total number of queries.

The reasons for the significant query failure rate within a low density scenario are explained in Figure 4. For the scenario with only 100 nodes the most common error (60% of all query failures) is due to 'no route'. We were now interested in the question of whether a better greedy or non-greedy forwarding strategy for queries and updates can alleviate this particular problem.

We examined whenever a 'no route' error occurred whether a greedy forwarding strategy that has full instantaneous knowledge on node positions could successfully deliver the query. In addition, we checked whether there is a path to the destination at all (using Dijkstra's algorithm). The results are presented in Figure 5. Obviously, the 'improved' greedy strategy will eliminate a significant part of the occurred errors while the introduction of a recovery strategy to the simple greedy forwarding can eliminate 70% to 90% of all 'no route' errors.

Another observation that can be made by viewing Figure 4 is that the proportion of 'TTL expired' errors is increasing for an increasing number of nodes. While a greedy forwarding strategy is inherently loop-free for a *static* network, node mobility can and actually does introduce some amount of 'looping' as explained in the following example.

Figure 6 (left) shows an example of a path of a location update that ended in a forwarding loop. The sending node 4 sends an update for the second order square located top left (a location update is not directly send to a specific destination since the destination, i.e., the corresponding location server, has to be determined 'on-the-fly'). The packet travels along the path via nodes 91, 25, 48, 65, to node 21. When it arrives there, node 21 has the 'misconception' that the update should be sent to node 8. However, node 21 is not aware of the fact that node 8 is not in the target second order square anymore. As we see from Figure

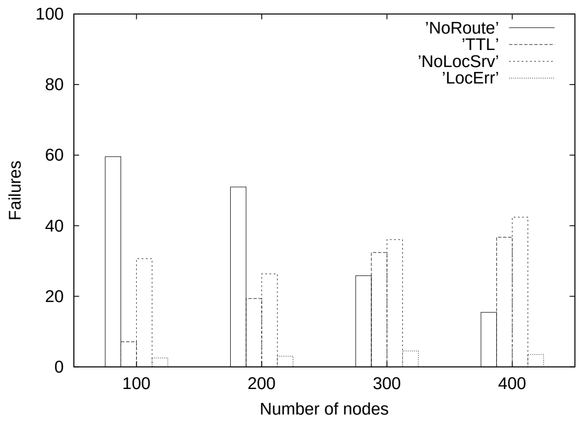


Figure 4: Shown are the distributions for the various error classes of query failures.

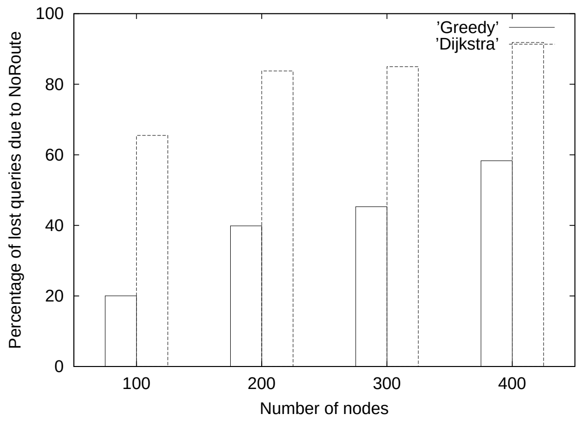


Figure 5: Compared to the number of queries that failed due to 'no route', this graph presents the proportion of queries that would have been successful with an 'ideal' greedy method as well as with using Dijkstra's shortest path algorithm.

6 (right) that shows a snapshot taken a few seconds before the one of 6 (left), node 8 traveled out of the target second order square. Now, when node 21 forwards the packet to node 44, this decision is also done based on somewhat 'outdated' position information. Finally, nodes 44 and 57 have an inconsistent view of their respective positions. Again, this can be easily explained from the 'travel history' as depicted in Figure 6 (right). Basically, node 44 thinks that node 57 might still reach node 8, while node 57 thinks that node 44 might actually still reach node 8. These kinds of inconsistencies can occur simply due to motion and 'under-sampling' of positional changes. However, since hello messages are unacknowledged broadcasts, it can also happen that a node misses the announcement of a positional change of a neighbor. Constellations of nodes with an inconsistent view of the positions might, therefore, act as 'black holes' for packets until the positional information is correctly updated again.

The analysis of the signaling load depending on node density as given in Figure 7 shows that the amount of updates per node is independent of the number of nodes. However, the amount of bytes for hello messages significantly increases with an increase in node density since the neighbor tables and forwarding pointer tables are increasing in size.

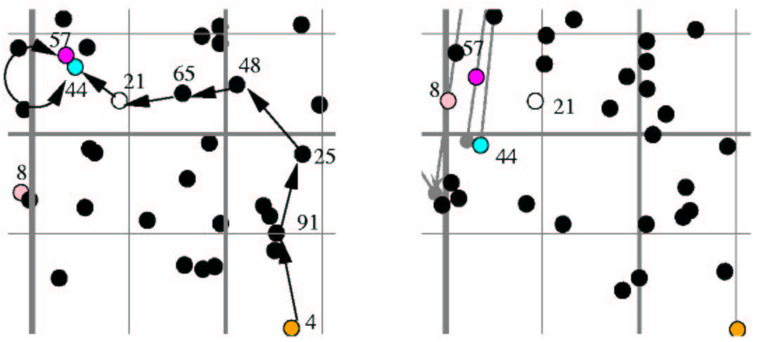


Figure 6: Two snapshots that illustrate why forwarding loops can occur. The figure on the right shows a snapshot with movement paths a few seconds before the snapshot given on the left side was taken.

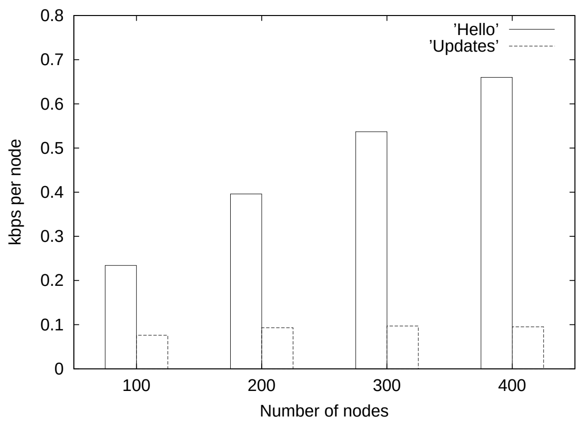


Figure 7: The per-node bandwidth consumption for hello messages and location updates.

3.3 Impact of Mobility

Following the study of varying node densities, we studied the effects that different speeds have on GLS performance. For this, we complemented the movement patterns we had used so far with two more sets of movement patterns. In these, nodes moved with a maximum speed of 30 m/s and 50 m/s respectively. We also became aware of a bug in the IEEE 802.11 MAC protocol implementation of the ns-2 simulator[5] and fixed it to get correct results from the simulation.

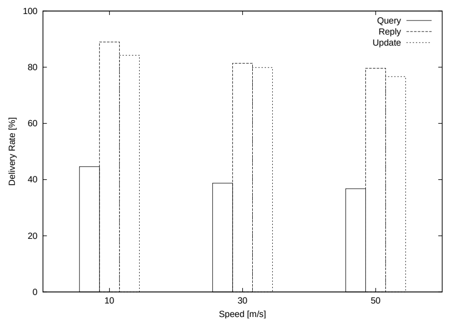
Figure 8 shows the delivery rates for the three types of packets that GLS' functionality is based on: Queries, Responses and Location Updates. It depicts them organized according to node density and speed per node density.

As was described in Section 3.2, scarcely poulated networks suffer from connectivity problems. Therefore the scenarios with 100 participating nodes showed low delivery ratios throughout the range of speeds that were simulated. We observed that speed had little influence on the delivery ratio in these scenarios, about 2-5% more loss, and concluded that the connectivity problems dominate the performance of GLS. Indeed, Figure 9(a) shows that 'no route' failures added up with 'TTL' failures are the most common causes of packet drops for this node density.

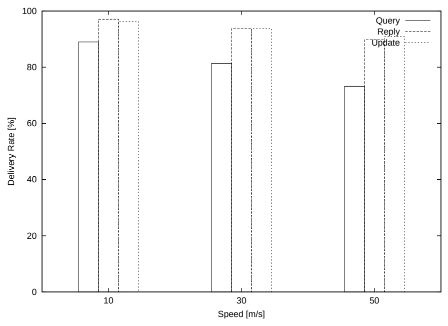
For increasing node densities, connectivity does not pose a problem and it was very interesting to see that speed still influenced the delivery rate only moderately with about 5-8% decrease in delivery, as long as speed and node density did not exceed a certain threshold[6]. Further study of this phenomenon revealed

[5]For details about the nature of this bug, please refer to Appendix F.

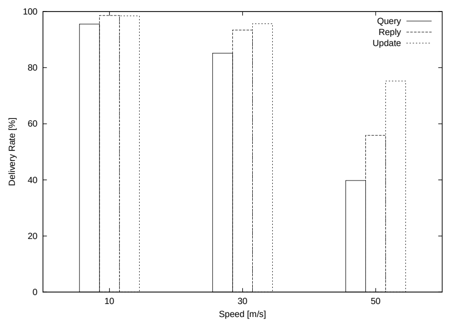
[6]For higher speeds and node densities other effects decreased the delivery rate, as will be explained later on in this section.



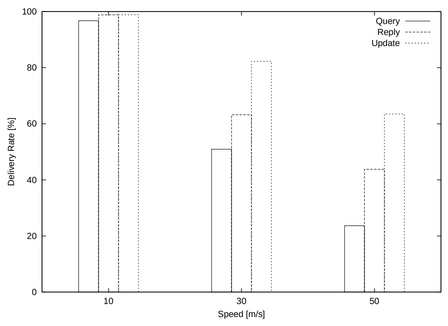
(a) 100 Nodes



(b) 200 Nodes



(c) 300 Nodes



(d) 400 Nodes

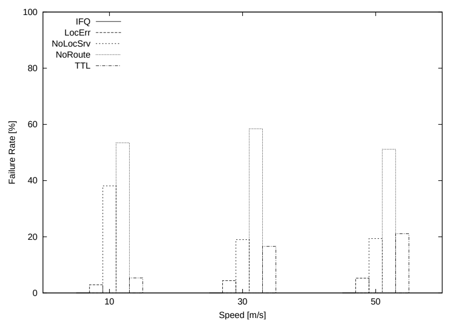
Figure 8: Percentages of query, response and update packets successfully delivered.

the resons behind it: In geographical routing no topological information is used to route from source to destination. Therefore, it does not matter *which* node between them forwards a packet, as long as *some* node within radio range *does* forward it. An increase in speed results in a higher probability of existance of such a node ’in between’, because mobility increases. While this may break existing routes it also creates possibilities for new ones that might not have existed before. A look at Figure 9 reflects this by showing a big decrease in ’no route’ errors for higher speeds.

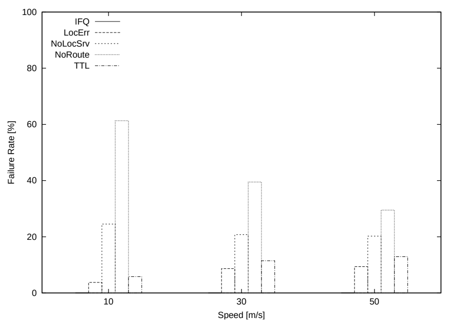
Unfortunately, the downside to speed is a decrease in positional accuracy. Greedy routing has a target position it routes a packet to – be it a data packet or a GLS maintenace packet with fixed destination. However, the faster nodes move about, the more unlikely it gets that a destination node is still at the position the packet was routed to, thus creating a location error. The ’quantizing’ effect[7] provided by the wireless link in form of the radio range mitigates this problem, but can not solve it altogether. Figure 9 shows the above mentioned increase in location error failures. We believe that the problem of positional inaccuracy can be alleviated further through several means of optimzation.

Another observation we made from Figures 8(c) and 8(d) was that for higher node densities and speeds GLS suddenly started to break down with respect to performance. Since this did not match our expectations for GLS, we analyzed these cases a little more. We discovered that the break down was due to a harmful interaction between the GLS and the IEEE 802.11 protocol. In Figures 9(c) and 9(d) we see a new failure type that did not occur before: IFQ drops. This failure type represents packets, that were dropped by the interface queue between the link layer and the MAC layer, because of an occuring overflow. In dense scenarios the MAC layer might often encounter a busy medium while trying to send a packet and has to

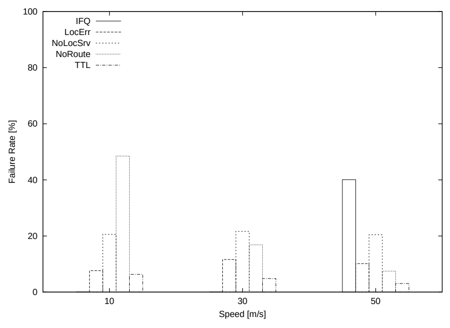
[7]This means that the exact position of a node does not matter as long as it is within radio range, thus providing a certain area in which all posiutions can be considered equal.



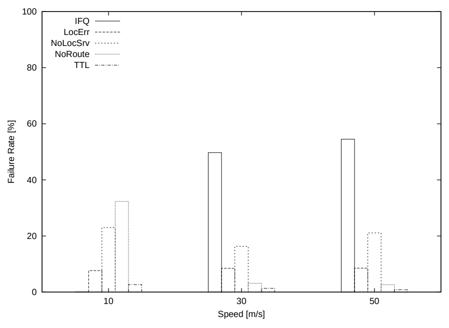
(a) 100 Nodes



(b) 200 Nodes



(c) 300 Nodes



(d) 400 Nodes

Figure 9: Percentages of query drops due to different failure reasons.

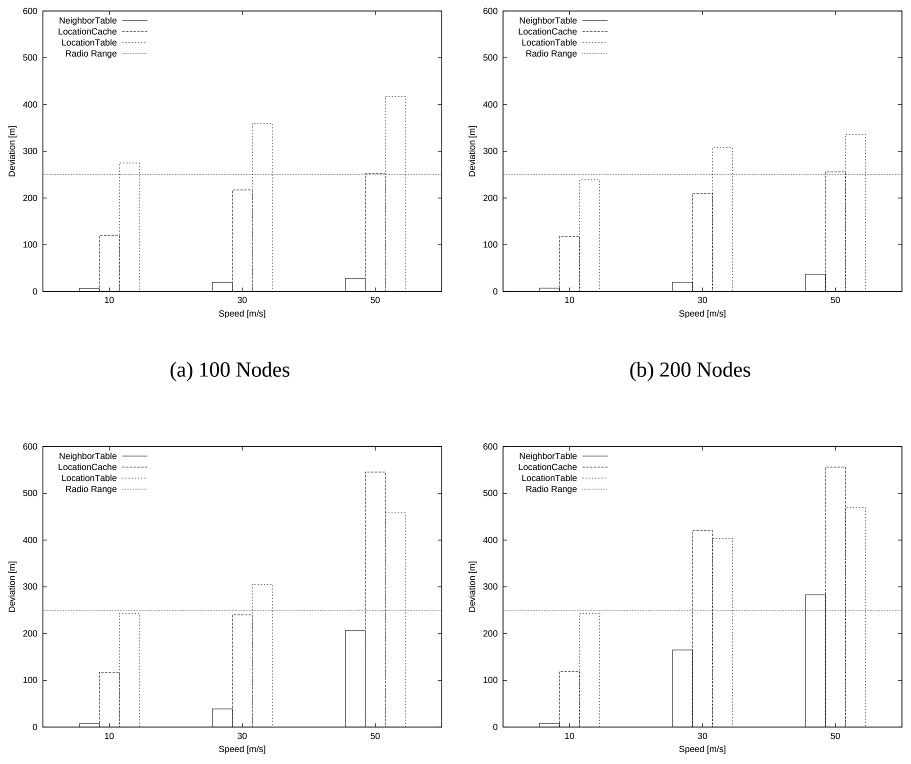
delay the send (i.e. retry). Node movement only adds to this delay problem, because even if the MAC aquires the right to send a packet, the intended recipient may not be within radio range anymore[8]. High traffic load may now lead to packets being held in the interface queue and thus inheriting and accumulating delays of packets queued and send before them. Delay in information distribution increases positional inacuracy which in turn increases the retries needed to find a valid next hop on the MAC layer. It is quite obvious that these effects will magnify each other, until GLS' functionality is crippled.

Figure 10 shows, that GLS' performance takes the most damage whenever the average positional deviation in the location caches rises above the radio range. This makes sense, because in those cases the position lookup mechanism starts to fail due to unreachable nodes. Again, some form of recovery mechanism could probably alleviate this problem.

4 Conclusions and Future Work

We presented a detailed simulative study on the Grid Location Service proposal. In our 2km$\times$2km scenario with 400 nodes, GLS consumes about 0.8kbps per node for signaling purposes and results in a location query failure rate of about 7%. The failure analysis and, in particular, the study on improved greedy or non-greedy forwarding indicate that improving the forwarding strategy by a reasonable recovery strategy will lead to a significant improvement of the location query success rate. This will also make lower density scenarios (with less than 100 nodes per square kilometer) more robust. We also showed that by increasing

[8]The Greedy Routing implementation of HGPS is especially vulnerable to this, because nodes chosen to be the best next hop tend to be in the outer reaches of the radio range and thus are most likely to soon leave the same.



(a) 100 Nodes (b) 200 Nodes

(c) 300 Nodes (d) 400 Nodes

Figure 10: Average deviations of positional information used to forward queries.

the node density, the bandwidth needed for hello messages dominates the costs for signaling. In addition we showed that node movement speeds of of 30 m/s and 50 m/s do not pose a problem for the GLS protocol, but may lead to harmful effects between network layers. We are currently investigating whether reducing the frequency for sending hello messages as well as increasing the update distance parameter can save bandwidth without affecting the query success rate. Furthermore, a study combining GLS and Greedy Perimeter Stateless Routing [Kar00] is underway.

A Original HGPS Code

This section gives an overview of the HGPS functions and what they do. It is sorted according to the file and class structure of the original code. For details, the reader is refered to the source code itself and the comments contained in it.

A.1 General / Non-Classed

A.1.1 bool inSameGrid(location src, location dst)

Found in `hgpsrouter.cc` and `locserver.cc`. Determines if two locations are in the same grid of a given order.

A.1.2 void loctodistance(int &x, int &y, location loc)

Found in `hgpsrouter.cc`. Converts a grid location into a geographical position. Used to calculate reference positions for grids.

A.1.3 bool BackOffTest(HGPSEntry *e, double time)

Found in `locserver.cc`. Links to the request table and decides if a new request may be send.

A.1.4 bool closer(nsaddr_t me, nsaddr_t him, nsaddr_t dst)

Found in `locserver.cc`. Checks if *me* is closer to *dst* than *him* in id-space.

A.1.5 double getOutofRangeTime(double min, double max, double speed, double now_pos)

Found in `locserver.cc`. Returns a speed-based prediction for the point in time that a given distance will be exceeded.

A.1.6 void getXY(double &min_x, double &max_x, double &min_y,double &max_y, location loc)

Found in `locserver.cc`. Returns the square limits that *loc* is in. Needed for predicting when a node will leave a square.

A.2 HGPSAgent

The agent code is contained in `hgpsagent{.h/.cc}` and is the base for all HGPS functionality. It is based on the DSR agent and uses a lot of its structure and function names.

A.2.1 void trace(char* fmt, ...)

Trace() is a function that can be called to print information to the trace file that is not supplied by the standard trace objects. Its usage is analogue to the C sprintf() function.

A.2.2 void startUp()
void sleep()

StartUp() and *sleep()* are called when a node is de-/activated (this includes an initial calling at the start of the simulation). They wake / put to sleep the agent as well as the router, the locserver, the link layer and the MAC layer.

A.2.3 int command(int argc, const char*const* argv)

Command() is the TCL link that *every* agent needs to have. It parses commands that were called in TCL scripts during run-time.

A.2.4 void recv(Packet *p, Handler *)

Gets all packets, that are handed up the stack by the MAC and the LL or handed down the stack by an application. If the packet has just been generated by the application attached to this node it is set up with a valid HGPS header. This includes initilization of basic header fields. Then it is handed over to *handlePktWithoutLoc()*. If the packet came up the stack transmission counters are incremented and the packet is handed over to *forwardHGPSPkt()*.

A.2.5 void forwardHGPSPkt(Packet *p)

Evaluates the packet type and hands the packet to a function in the agent, the router or the locserver. The packet type indicates which function is appropriate for its processing.

A.2.6 void handlePktWithoutLoc(Packet *&p,bool retry)

Checks the neighbor table in the router for the location of the destination. If successful the packet is send. Otherwise, the packet is handed to the locserver function *getLocation()* to inquire the location. Should this fail as well, the packet is inserted into the sendbuffer if it is not already queued in it.

A.2.7 void handlePktReceipt(Packet *p)

Packets that are destined for this node are evaluated here, according to their type:

Data Packets are evaluated for cachable positional information and passed to the application.

Error Packets are used to purge outdated location information from the cache.

Location Notification Packets are evaluated for the updated positional information the communication partner supplied.

Query Packets are handed over to the locserver.

Query Response Packets are handed over to the locserver.

A.2.8 void notifyLocation(nsaddr_t id,location loc)

Sends out all packets that are stored in the sendbuffer and are destined for *id*.

A.2.9 void stickPacketInSendBuffer(Packet *p) void dropSendBuff(Packet *&p) void sendBufferCheck()

Sendbuffer related functions to add packets to the sendbuffer, remove packets from the sendbuffer and check the sendbuffer for packets that have timed out or can be send. Periodically called by a timer.

A.3 HGPSRouter

The router sources can be found in `hgpsrouter{.h/.cc}`. The general idea of this class is to collect all functions that deal with the neighbor table and route selection (not location selection).

A.3.1 void setLocServer(LocServer *s) void start() void sleep()

Set up router with link to locserver and do de-/activation.

A.3.2 void broadcastOneHopRT()

Creates a beacon (hello) packet, adds information about one and two hop neighbors to it and calls the locserver function *piggybackFP()* to enable piggybacking of forwarding information by the locserver. The beacon is then broadcast. This function is called periodically by a timer.

A.3.3 void recvNeighborUpdate(Packet *pkt)

Evaluates beacon packets and adds their content to the neighbor table. The packet is also passed to the locserver function *recvFPUpdate()* to evaluate forwarding information that might be piggybacked.

A.3.4 void deleteEntry(nsaddr_t nextnode)
void deleteEntry(nsaddr_t fromnode, nsaddr_t tonode)

Removes entries from the neighbor table by either deleting *nextnode* or removing *fromnode* from the path to *tonode* that is recorded in the neighbor table. It also cleans the table of entries that have timed out.

A.3.5 void addUpdateEntry(location loc,nsaddr_t next_hop,double x, double y,nsaddr_t dst, int distance, double speed)

Adds or updates an entry in the neighbor table and the route table, according to the data that is given in the parameter list.

A.3.6 nsaddr_t getBestNextHopNode(hgps_route_entry &rnt)

Returns the best next hop node from the list of next hop nodes that is provided. Not used in the current implementation.

A.3.7 nsaddr_t getNextHopNode(hgps_route_entry &rnt)
nexthop_entry* getNextHopNode(nexthop_entry *&next_hop)

Returns the next node or the next valid nexthop_entry on the path to *next_hop*.

A.3.8 nsaddr_t getNextHop(location hisloc, nsaddr_t prev_hop)

Searches the one and two hop neighbor table for the best next hop (greedy routing) and returns it. If the best hop is a two hop neighbor, the one hop neighbor that leads to it is returned.

A.3.9 void sendOutPktWithLocation(Packet *&packet,double delay,bool checkNT, bool invoke_loc_server)

Preprocesses a packet that is about to be send. If the packet has the location information of an intermediate destination, it is either forwarded according to neighbor table information, dropped due to unavailability of the same or dropped if a route loop is detected. If next intermediate or final destination location information has to be found first, *getNextHop()* is called to find the best greedy neighbor. Should it not return one, differentiated error handling is performed according to the packet type, the packet state information and the reason no next hop could be found.

A.3.10 void HGPS_XmitFailureCallback(Packet *pkt, void *data)
void xmitFailed(Packet *pkt)

Called if the MAC is unable to deliver the packet. The packet is analysed for its type and handled appropriately, i.e. handed over to its handler funcion. Afterwards every packet in the interface queue that was primed for the same destination is extracted and also handled in a suitable way.

A.4 LOCServer

Sources for the location server are placed in `locserver{.h/.cc}`. This class is used to manage the location service aspect of HGPS. Therefore it represents GLS. It keeps the location caches and the location table and handles queries, responses, updates and forwarding information.

A.4.1 void LocServerTriggerHandler::handle(Event *e)

This function is a timer function, that is called periodically to test the need for sending of update packet. It checks all hierarchy levels for square crossings (i.e. has a node changed the square it is in), and initiates update sends if a crossing did occur. Then it calls the locserver function *checkPeriodicUpdateLoc()* to test other update conditions.

A.4.2 void setRouter(HGPSRouter *r)
void setMobileNode(MobileNode *m)
void start()
void sleep()

Set up the locserver with a pointer to the router and the mobilenode and do de-/activation. This includes calculation of the superimposed grid structure, that is used in GLS.

A.4.3 location getGrid()
double getSpeed()
location getGridPos(double &my_x, double &my_y)
void getPos(double &my_x, double &my_y)
location getGrid(double x, double y)

Functions to gather various informations:

- return the grid number corresponding to the node itself or a given position
- return the speed the node is moving at
- return the grid number along with the momentary position of the node
- return the grid a given position is contained in

A.4.4 void recvFPUpdate(Packet *packet)

Evaluates a beacon or forwarding packet for the forwarding information piggybacked to or contained in it. The information is added to a forwarding cache, that discards oldest information – if the table is full – to store it.

A.4.5 void piggybackFP(Packet *packet)

Adds the information contained in the forwarding cache to a beacon packet.

A.4.6 void dumpFPTable()

Invalidate forwarding cache.

A.4.7 void recvUpdate(Packet *packet)

If the node is not in the area that is targeted by the update packet, the packet is handed to the router function *sendOutPktWithLocation()* to simply forward it. The same happens if the node realizes that the packet has been in the to-be-updated area before but was forwarded here, because that means some other node had outdated information. Should the node actually be in the right grid, then it asseses whether it should keep the update packet, and add the location information to its location table, or forward it on to a known node that is closer to the source node in id space.

A.4.8 bool getLocation(nsaddr_t dst_, location &dst_loc, Packet *packet)

This function tries to find location information for *dst_*. First it queries its own location cache for this information. Should this be unsuccessful, the location table is queried. In case both are unable to produce results the location table is searched for a node closer in id-space to the destination, that might have the required information. If one is found, a query is set up and send to this closer node, as long as no other query is already on its way. In the unlikely case that the location table can not provide a closer next node the location cache and neighbor table are searched for one. Should none be successful, false is returned to indicate a location lookup failure.

A.4.9 void recvQuery(Packet *pkt, bool dst_loc_failure)

Evaluates queries and decides how to proceed in handling them. If the node should be able to deliver the query to it's target but is not, the forwarding cache is searched for a rescue route. Should no such route exist (i.e. no forwarding information can be found in the forwarding cahce), the packet is dropped. In case the node itself is the query target it sends a response and discards the query packet. As long as the query has not aquired the position of the destination node it is routed on to nodes closer in id-space that might possess the needed positional information. This is done the same way as in *getLocation()* (A.4.8). Another possibility is that the node, while not the destination node itself, is one of its location servers and can provide the positional information itself. In this case it adds the location to the query and forwards the packet towards the target node.

A.4.10 void recvQueryResponse(Packet *pkt)

Response packets are evaluated for positional information which is added to the location cache. Then the agent function *notifyLocation()* is called to check the sendbuffer for waiting packets, before the response packet is discarded.

A.4.11 void checkPeriodicUpdateLoc()

Periodically called to evaluate the need for updates. This function tests if the nodes travelled distance has exceeded a given value (the update distance) or if location server nodes might thinks it has (false prediction). If so, *updateLoc* is called to generate and send update packets to those grids that might need them. A division mechanism makes sure location servers further away do not get updated as often as closer ones do. Afterwards, nodes that are considered to be active communication partners are also informed of the nodes new position by sending them a location notification packet.

A.4.12 void updateFPGrid(location which_loc)

Called whenever a node changes its square in hierarchy level 0, this function generates and broadcasts a forwarding packet This packet type is received by every node within radio range and kept by those nodes that are in the old square. This way the moved node leaves a ”forwarding address” in its old grid.

A.4.13 void updateLoc(int level, char* upd_reason = HGPS_NOREASON)

UpdateLoc() generates three update packets and sends them to the three grids of hierarchy level *level* that, together with the grid the node itself is in, make up the *level*+1 hierarchy square.

A.5 LocTable

Sources can be found in `locserver{.h/.cc}`. This class is the prototype for the location cache, the location table and the live connections cache.

A.5.1 void addUpdateEntry(nsaddr_t id, location loc, double timeout)

Adds or updates an entry for *id*.

A.5.2 void deleteEntry(nsaddr_t id)

Deletes an entry from the table.

A.5.3 loc_entry* getEntry(nsaddr_t id, double now)

Returns the entry for *id*, if it exists.

A.5.4 bool findClosestinRT(nsaddr_t myid, location myloc, nsaddr_t dst, struct hgps_route_entry *nt, loc_entry &entry, bool in_same_grid) bool findClosest(nsaddr_t myid, location myloc, nsaddr_t dst, struct hgps_route_entry *nt, loc_entry &entry, double lastupdate_time) bool findClosestInGrid(nsaddr_t myid, location myloc, nsaddr_t dst, location dst_loc, struct hgps_route_entry *nt, loc_entry &entry, double lastupdate_time) void findClosestExceptDst(nsaddr_t myid, location myloc, nsaddr_t dst, struct hgps_route_entry *nt, location update_loc, loc_entry &entry)

Returns the node closest in id-space to a given destination (includes: closer than the node itself) that is a member of the table and still valid. May be restricted to nodes that are in the same square as the node or nodes that are not the destination themselves.

A.6 QuadTree & HGPSRequestTable

The code can be found in `quadtree{.h/.cc}` and `hgpsrequesttable{.h/.cc}`. While requesttable is an adaptation of the request table used in DSR and therefore used to restrict sending of queries, quadtree is only used in a special debug case of HGPS (ideal routing), that we did not study. Functions in requesttable.cc have not been modified and their use is limited to add, remove, get and process entries from this table.

B Changes to the HGPS Code

We now present a closer look at the modifications we made and why we made them. This includes bugfixes as well as additions and alterations. However, this section will not include all the changes needed to make the HGPS source work with ns-2 version 2.1b8a. Neither will it list all the syntactic modifications we did to the source to clean it up or fix some minor issues with the programming style. Most of these are commented in the code itself.

B.1 Trace System

The traces implemented in the HGPS code were inconsistent in form - which made them hard to analyze automatically - and did not provide all the information we needed. Therefore, we deactivated or removed most of them and introduced our own extended traces, which added HGPS dependent information to a scheme based on the normal packet based ns-2 traces. We also introduced some additional fields to the HGPS packet header (e.g. hop counters) that, while not influencing the protocol, could be used to transport additional information along with the packets. Throughout the code you will find *trace()* and *exttrace()* function calls, that do the actual tracing in a way similar to C's sprintf() function. For the most part a *hgps_verbose* followed by a *trace()* is the original code, while *exttrace()* was introduced by us. The router function *sendOutPktWithLocation()* has been modified extensively, because the original HGPS code classified nearly every possible routing failure as 'no route available', whereas we wanted to know a little more about why the packet could not be routed. For a detailed description of what was traced and how to read the traces, please refer to Appendix D.

B.1.1 void exttrace(Packet *p,char* fmt, ...)

As part of the rewrite of the traces, we introduced this function, which accepts the additional parameter of a packet. It evaluates some standard header fields, that might be of interest and prints them in a common format to the trace file in front of the user supplied fields. This makes the actual traces more consistent and assures that no important data is forgotten.

B.2 Wake/Sleep Code

The original HGPS code used a wake/sleep system that only allowed to set a wakeup time with a duration for staying awake. This seemed inappropriate for our purposes. So we removed the old system and introduced seperate wake and sleep commands, that could be called in TCL at will, to activate or deactivate nodes.

B.3 Test Query Only Mechanism

In test query only mode (i.e. no data packets are sent, they are just generated as dummies to initiate a query cycle and inquire a location) queriy packets were sent, even if we found a location in the src node. We changed this default behavior, since for our analysis, we needed to know, how many queries were generated and where the response aquired its location information.

B.4 Timing

Some timings had to be changed in the original code, to avoid unwanted effects of simulation. The biggest change would be, that beacons were sent every BCAST_INTERVAL_ seconds and, due to the synchronizing nature of a simulator, tended to disrupt each other, because they were sent at the exact same time. So we introduced a random jitter to their sends, that delayed broadcasts.

B.5 Routing Loop Detection

We tried different solutions for detection of routing loops, but without storing additional information in the packet, no easy solution could be found, that might not compromise the functionality at some point. We ended up using a scheme nearly identical to the original one, but we decided to forward packets 4 times without target modification instead of the original 3 times.

B.6 Bugfixes

There were also some serious bugs in HGPS that we had to fix in order to complete our studies:

- In hgpsagent.cc in the function *locnotification()* the destination of the location notification packet was evaluated and its information added to the locaton cache. The destination, however, would be the node itself (since it received the packet) and a location notification is supposed to inform the destination of positional changes of the source. We corrected it to evaluate the source information.

- Though not really a bug, we should mention that contrary to the paper about the HGPS code, there is no periodical sending of update packets. The function *checkPeriodicUpdateLoc()* only sends updates if the node exceeded its update distance or it predicts that others might think so and discard its positional information.

- In hgpsrouter.cc in the function *xmitFailed()* the extraction and handling of packets in the IFQ was flawed. Due to pointer problems all packets were extracted, but only the first packet was handled correctly. All other packets were lost and posed a memory leak.

- The routing loop detection code in the function *sendOutPktWithLocation()* in hgpsrouter.cc used a $variable - 1$ calculation as an index for an array without checking that $variable$ is ≥ 1. This resulted in invalid memory accesses, because $variable$ could be 0.

- Throughout hgpsrouter.cc the rt_1 and nt arrays had the implementaion problem, that, if full they searched for an invalid entry they could replace. However, if all entries were valid - which might happen quite often for large simulations - no free slot for a new entry could be found. Since no handling for this case was implemented in the original code, the *addUpdateEntry()* function would then simply try to access the invalid default index of the array and cause a segmentation fault. The workaround we used consists of increasing the size of these arrays for our simulations (to avoid the problem) and adding some code that just drops the new entry if no slot is available. This may introduce functional errors to the protocol, but a better solution requires a better data structure.

C Pattern Generation

To study HGPS we wrote some scenario and traffic file generators and used them to create the patterns we set up the simulator with. This section will show how to generate file with them.

C.1 Scenarios

The scenario generator is called setdest since it is mostly based on the setdest code, that comes with the ns-2. For faster setup of the simulations it is recommended to use the ad-hockey tool to reverse the ordering of the scenario that is generated, as it decreases loading times, but has not yet been implemented into the scenario generator itself. Setdest uses the following parameters:

-n ⟨nodes⟩ Specifies the number of nodes that should be generated and moved.

-p ⟨pause⟩ Specifies the pause time, that nodes should rest, when they've reached their destination, before choosing the next destination (see *Random Waypoint Mobility Model*).

-s ⟨speed⟩ Specifies the maximal speed that nodes should have. On average nodes will move with about half the speed that is specified here.

-t ⟨time⟩ Specifies the simulation time in seconds; or rather the time during which the nodes should continue to select targets and move towards them. Simulations can be shorter or longer.

-x ⟨width⟩ Specifies the width of the simulated are in meters. This means it is the maximal x value that nodes may have.

-y ⟨height⟩ Specifies the height of the simulated are in meters. This means it is the maximal y value that nodes may have.

-d ⟨range⟩ Specifies the maximal distance in meters that nodes may choose their destination in if movement inhibition is activated. Used to enable a localized movement pattern, where nodes never travel far and thus have smoother movement (using more waypoints).

-v ⟨probability⟩ Specifies if movement inhibition should be enabled and how strong it shall be. It requires a value between 0 and 1, where 0 is global movement and 1 is local movement. E.g. -v 0.5 will make nodes choose 50% of their destinations on the global plain and 50% of their destination in a loacalized neighbourhood specified by -d.

-r ⟨range⟩ Specifies the radio range in meters, that will be used in the simulation. Only useful if shortest path calculations are enabled to enable checking for possible communication between two nodes.

-c ⟨on/off⟩ Specifies if the Floyd Warshall algorithm should be used to calculate reachability values for all nodes. This may take up *considerable* calculation time and will give a table at the end of the scenario file, that includes information about link changes and other shortest path results. The scenario will also contain TCL commands to feed the calculated information to the ns-2 GOD object.

For example: ”*setdest -n 2 -p 2 -s 5 -t 5 -x 10 -y 10 -d 10 -v 0 -c 0 -r 10*”
may give something like this:

```
#
# nodes: 2, pause: 2.00, max speed: 5.00 max x = 10.00, max y: 10.00
#
$node_(0) set X_ 4.968836821805
$node_(0) set Y_ 1.240468477411
$node_(0) set Z_ 0.000000000000
```

$node_(1) set X_ 8.553700948872
$node_(1) set Y_ 2.051855275323
$node_(1) set Z_ 0.000000000000
$ns_ at 2.000000000000 "$node_(0) setdest 9.839499284851 2.464489211415 0.335089215216"
$ns_ at 2.000000000000 "$node_(1) setdest 9.020593817345 9.120296117890 2.408430733315"
$ns_ at 4.941269531656 "$node_(1) setdest 9.020593817345 9.120296117890 0.000000000000"

C.2 Traffic

The traffic generator is called trafgen. It can generate traffic in five different formats and has many parameters to tune the traffic that is to be sent in the simulation:

-n ⟨nodes⟩ Specifies the number of nodes that will be in the simulation and should communicate with each other. The communication partners are chosen from this pool at random.

-t ⟨time⟩ Specifies the simulation time in seconds; or rather the time during which the communications should be initiated. It is recommended to set this parameter lower than the intended simulation time to enable open connections to finish on their own.

-c ⟨connections⟩ Specifies the number of connections (i.e.data flows) that should be initiated by each node. The number of packets sent in each flow is specified seperately.

-m ⟨mode⟩ Specifies the type of connection that will be used (i.e.the application that will send the packets):

0 Query Mode. Generates test query files for HGPS. These files can not be used by any other protocol.

1 CBR (Old). Generates constant bitrate traffic (CBR) files in the old format. ns-2 still accepts them, but it is recommended to use the new format.

2 CBR (New). Generates constant bitrate traffic (CBR) files in the new format. CBR is UDP traffic (one-way, unacknowledged).

3 TCP. Generates TCP traffic files, assuming the application to be an FTP client/server.

4 Ping. Generates ping like traffic file. Ping uses our implementation of a two-way application. It accepts the same parameters as CBR, but sends an echo packet back to the source.

-w ⟨time⟩ Specifies the startup wait time, that makes the generator delay connection initialization until after the simulator has reached this timestamp. All connections will therefore be within the time window given by **-w** and **-t**. It is intended to allow bootstrapping mechanisms of some protocols to take effect, before communication is scheduled. Optional parameter.

-i ⟨on/off⟩ Specifies if an inverted timescale should be used. 1 is the defualt value, because inverted timescale files are read faster by the ns-2 parser. In case you do not want to have the traffic file reversed, specify 0. Optional parameter.

-r ⟨rate⟩ Specifies how many packets should be send per second and connection. It therefore defines the interval between packet sends. Not applicable for test-query mode. Optional parameter.

-s ⟨size⟩ Specifies the size in bytes that one data packet should have when it is generated by the application. Not applicable for test-query mode. Optional parameter.

-p ⟨packets⟩ Specifies the number of packets that will be send per connection. Large numbers increase the overall connection time (if the rate stays the same). Not applicable for test-query mode. Optional parameter.

-o ⟨size⟩ Specifies the TCP window size that should be used per connection. Only valid for tcp mode. Optional parameter.

For example: *”trafgen -n 2 -t 5 -c 1 -m 4 -w 3 -r 2 -s 128 -p 2”*
may give something like this:

```
#
# nodes: 2, max conn: 1, send rate: 0.500000, seed: 1
#

#
# 3.466233979172 0 -¿ 1
#
set pings_(0) [new Agent/Ping]
$ns_ attach-agent $node_(0) $pings_(0)
set pingr_(0) [new Agent/Ping]
$ns_ attach-agent $node_(1) $pingr_(0)
$pings_(0) set packetSize_ 128
$pings_(0) set interval_ 0.500000
$pings_(0) set maxpkts_ 2
$pings_(0) pingLog $pingLog
$ns_ connect $pings_(0) $pingr_(0)
$ns_ at 3.4662339791717867 ”$pings_(0) ping 1”
#
# 4.365762962592 1 -¿ 1
#
set pings_(1) [new Agent/Ping]
$ns_ attach-agent $node_(1) $pings_(1)
set pingr_(1) [new Agent/Ping]
$ns_ attach-agent $node_(1) $pingr_(1)
$pings_(1) set packetSize_ 128
$pings_(1) set interval_ 0.500000
$pings_(1) set maxpkts_ 2
$pings_(1) pingLog $pingLog
$ns_ connect $pings_(1) $pingr_(1)
$ns_ at 4.3657629625922763 ”$pings_(1) ping 1”
```

D Tracefile Description

Covered in this section are HGPS specific traces that were used in the evaluation of the protocol. Although data from the general ns-2 traces was also used, it will not be covered here and is assumed to be known from the ns-2 documentation.

D.1 Packet Traces

Fields:

1	[2a 2b]	[3a 3b]	[4a 4b]	:	5	6	7	[8a 8b]	[9a 9b]	10

Legend:

1. HGPS Packet Type
2. Location Informations
   (a) Source Node Location
   (b) Destination Node Location
3. Requested/Aquired Node Information

   (a) DST Node ID (may be -2 for *unknown)*
   (b) DST Node Location (may differ from 2b)
   0 for Query Packets
   DST Location for Response Packets

4. Update Destination Information
   (a) Target Location Square
   (b) Mask (implying the order of the square)

5. HGPS Internal Drop Reason (valid for dropped packets):

   NORSN No Reason (Default; should not happen)

   RLOOP Routing Loop Detected

   OUTLOC Outdated Location, i.e. destination is no longer at the position we routed to

   NRTE No Route, i.e. no greedy neighbor

   TTL Time-to-Live expired

   MESLP Sending Node is asleep (Should not happen)

   HIMSLP Target Node is asleep (Should not happen)

   NOSRVL No Next Location Server at the node who generated the packet

   NOSRVF No Next Location Server at some node who should forward the packet

   LOCERR Outdated Location for query target, i.e. next locserver is no longer at the position we routed to

6. Packet Destination Node ID (should equal 3a)

7. Callback Count (inacurate; only used to see if packet has been in a callback situation)

8. Forwarding Information
   (a) Common Header Forwarding Counter (cmn→num_forwards_)
   (b) HGPS Header Forwarding Counter (hgps→hop_trans_)

9. Connectivity Analysis (Needs enabled path analysis in GOD object)
   (a) Greedy Path from this node to SRC or DST[9].
   (b) Shortest Path from this node to SRC or DST[10].

10. HGPS Internal Update Reason (valid for update packets)
    Possibilities:

    XING Square Crossing. Set if the node sends an update because it has changed its order-n square

    DIST Distance. Set if the node sends an update because it has exceeded the given update distance

    PRED Prediction. Set if the node sends an update because others may predict that it has exceeded its given update distance

    DNP Distance and Prediction. Set if both DIST and PRED are true

[9] SRC if packet was received; DST if the packet is send or dropped.
[10] SRC if packet was received; DST if the packet is send or dropped.

D.2 Agent Traces

Fields:

HGPS	1	_2_	RTR	3	4	—	[5a:5b 5c:5d 5e 5f]	6	7	8	[9a 9b]

Legend:

1. Simulator Timestamp
2. Node ID
3. Packet UID (Unique Identifier)
4. Packet Size [Bytes]
5. IP Header Information
   (a) SRC ID
   (b) SRC Port
   (c) DST ID
   (d) DST Port
   (e) Time-to-Live
   (f) Next Hop
6. HGPS Packet Type
7. HGPS Packet Operation (influences 8a to 8c)

   DEVI Deviation Analysis
   UPDRCV Update Reception
   CONN Connectivity Analysis
   CLBK Callback from MAC Layer
8. HGPS Packet Additional Info
   (a) Information 1

   DEVI Location Source
   - **0** 2-Hop Neighbor
   - **1** Location Cache Entry before Query Send
   - **2** Location Table Entry before Query Send
   - **3** Location Cache Entry after Query Send
   - **4** Location Table Entry after Query Send
   - **5** Direkt Answer to Query from Query Target (No Location Server passed)
   - **6** Direkt Answer to Query from Query Target (Passed Location Servers)

   UPDRCV Hops taken from SRC to Node
   CONN Connectivity for
   - **0** Data Packet, that is about to be send
   - **1** Query Packet, that is about to be send

   CLBK Transmission Failure Reason

   (b) Information 2

   DEVI Positional Deviation [meters]
   UPDRCV Greedy Path from Node to SRC
   CONN Greedy Path from Node to DST

CLBK 0

(c) Information 3

DEVI Positional Information Age [seconds]
UPDRCV Shortest path from Node to SRC
CONN Shortest path from Node to DST
CLBK 0

E Evaluation Tools and Scripts

E.1 Ad-Hockey

Ad-Hockey is a visualization tool developed by the CMU to evaluate their wireless extensions to ns-2. It was written in perl and, while not distributed with the ns-2 source, can still be used to do so. In the course of this study we extended and modified it to make it compatible with version 2.1b8a and help us in some respects. However, some features are only preliminary and buggy. The most important changes included:

- Changed the numbering scheme of ad-hockey to work correctly with node 0. Old ns-2 versions started node numbering at 1.
- Extended screencap capabilities to automatically capture and save screenshots regularly. The resulting files can then be used to create a scenario movie.
- Added color-keying for HGPS packets. This enables easier tracking of different types of HGPS packets and helped in understanding how HGPS works. NOTE: If the switch for ”Trace HGPS Only” is activated no other protocol packets will be displayed.
- Fixed and modified the trace file parsing to conform with new traces, that we introduced.
- Added new features, that can be de-/activated or used at the command line. For details see usage section below.

We will now give an explanation of the command line switches, that ad-hockey uses. Most options, however, are accessible via the GUI and are straight-forward in their use. Thus, they will not be covered here. Usable parameters are:

-help | **-h** Displays all available command line options.

-slowdown | **-sl** ⟨**scale**⟩ Specifies the default speed scale for processing.

-autostart Automatically starts the simulation display after the file parsing.

-autorewind Automatically rewinds the timescale, after the simulation display has finished.

-hgps Enable HGPS packet tracing for color-keying.

-show-range | **-sr** Draws range circles by default.

-grid-size | **-gs** ⟨**size**⟩ Draw a superimposed grid structure on the field, with the given border length for the smallest square. Useful to evaluate HGPS and estimate distances.

-ext-scaling | **-es** Scales x- and y-length values to better fit the display area. Useful for scenarios, that differ much from the 1:1 rato of the standard ad-hockey display.

-range ⟨**range**⟩ Specifies the radio range that ad-hockey should use for range circles and connectivity graphs.

-convert | **-co** Converts scenario files from random TCL to an inverse time sorted format, that might be parsed faster by the ns-2 simulator, thus decreasing set-up time. NOTE: with -convert a scenario file parameter is mandatory.

⟨**scenario file**⟩ The first name given without an option ”-” is assumed to be the scenario file. If none is given ad-hockey starts up clean.

⟨**trace file**⟩ Should a scenario file have been given, then a trace file may also be specified as the next parameter. The traffic file should contain traffic matching the scenario.

E.2 Scripts

To automate the tracefile evaluation process or other simulation related things we wrote several scripts that might aid us. Two of them are essential to reproducing the results we made, which is why we will present them in this section.

E.2.1 run.tcl

Run.tcl is a script used to automate the simulation set-up process. Used in conjunction with the ns command many parameters can easily be modified – either on the command line or in the script itself. Since all command line parameters can be set to a default value in the script itself, we will first specify the HGPS related command line switches and append information that can only be changed in the file itself at the end. Following the *ns run.tcl* call may be:

-nn ⟨**nodes**⟩ Number of nodes participating in the simulation.

-stop ⟨**time**⟩ Duration of the simulation.

-x | **-y** ⟨**dimension**⟩ Field dimensions that nodes will move in.

-adhocRouting ⟨**protocol**⟩ Routing protocol to be used. For GLS, *-adhocRouting HGPS* needs to be specified.

-zip ⟨**0/1**⟩ Should the trace file be zipped on the fly to save space or avoid the filesize limit. Valid switches: 0 and 1.

-out ⟨**file**⟩ Trace file, that should be used to pool all messages any ns trace objects might provide.

-cp ⟨**file**⟩ Connection pattern (traffic file) to be used in the simulation.

-sc ⟨**file**⟩ Movement pattern (scenario file) to be used in the simulation.

-on_off ⟨**file**⟩ Wake/Sleep pattern (activity file) to be used in the simulation.

-pingLog ⟨**file**⟩ Log file for ping statistics, if ping traffic is used. If not specified no log file will be generated.

-upd ⟨**distance**⟩ Update distance that should be used for estimation of needed HGPS update packets.

-tqo ⟨**0/1**⟩ If only queries are to be tested, instead of actual traffic, specify 1. All data packets (if there are any and the traffic file is not a test-query traffic file) will be dropped as soon as their destination location has been aquired.

Other important parameters to set in the script itself are:

Agent/HGPSAgent→broadcast_random_ If set to 0 beacons will no longer be jittered. WARNING: This will most likely have the effect of synchronization, that will lead to heavy beacon loss.

God→path_analysis_ If set the GOD object will calculate greedy and shortest hop paths if requested to do so by a trace. While this was used in our evaluation and is necessary for some parts of the evaluation script, it does increase simulation times by some order of magnitude, because of the run-time behaviour of the used algorithms.

Node/MobileNode→movtrace_ Should be set to 1 to get accurate speed information from the evaluation script. Movement traces – if unwanted – should be deactivated by setting val(movtrc). However, if val(movtrc) does not deactivate movement traces, then this may do the trick.

Phy/WirelessPhy→bandwidth_ The bandwidth of the wireless interface may be specified with this parameter. It is given in bits per second.

Phy/WirelessPhy→Pt_ This specifies the power used to transmit and therefore the radio range. Some useful values may be taken from the comments, but to calculate others one has to read the source of the wireless interface implementation.

val(agttrc) Should traces on the AGT level be written to the trace file.

val(rtrtrc) Should traces on the RTR level be written to the trace file.

val(mactrc) Should traces on the MAC level be written to the trace file.

It should also be noted, that many more ns-2 parameters can be configured with this file, which were not used in this study.

E.2.2 evaluate.pl

All our evaluation was done with a script called evaluate.pl. It's a perl script that parses one or more trace files and gives statistics on the important information. If multiple files are processed averages are computed, but the user needs to make sure that all files provided match each other (i.e. simulations should not have used different node numbers). The following data is provided:

General Statistics Information on the node number, the file count, the used MAC, the simulation field, simulation duration and movement characteristics.

Packets Information on the different packet types that were sent, resent (following a MAC callback), received or dropped and percental information on the delivery rate. NOTE: Broadcast packets will have invalid values for receive and delivery, since one sent packet will be received by all nodes within the radio range.

Packet Drops Information on why packets have been dropped. This is useful to analyze which part of the HGPS process is causing the most faults and gives hints as to what parts might best be improved.

Connectivity May be used to analyse how many packets, that were dropped for no route reasons could have reached their target with a better routing algorithm. NOTE: Only usable in conjunction with activated path analysis in the GOD object.

Deviation Lists where a location query was answered. Additional information is for the average deviation the entry had and how old it was.

Update Packets Information on how many updates were sent for what reason. This may be used to learn more about the update process and how to improve it.

Movement Information on how long nodes stayed in the squares of different hierarchies.

Bandwidth Information on how many bytes per second and node had to be sent. This helps in learning the costs of the routing algorithm and the location service.

Callback Information on how many callbacks happened. Useful in learning about the MAC congestion.

F Linux and NS-2

During our simulations we encountered some problems that were not related to the GLS protocol or HGPS implementation, but still prevented us from aquiring correct results.

The first was a problem with the **memory management of linux**. While the linux kernel in its newer incarnations is perfectly capable of handling large memory sizes, standard distributions tend to focus on home environments and limit the maximal RAM size to 1 GB. Large simulations (e.g. our 400 nodes runs) use up more than this amount. It is therefore imperative to check that the used linux kernel can handle bigger RAM sizes correctly without cutting them off. Since RAM is limited it is also important to use a linux version that no longer limits the size of the swap space, since a lot of it might be needed if the system runs low on memory.

Another problem were two lines of code in the IEEE 802.11 MAC protocol implementation of ns-2. The lines in question reset the retry counter of a failed RTS packet, whenever a data packet is received. In dense networks, with a lot of traffic this may lead to an infinite blocking of packets on a node that receives packets, but is unable to deliver any. This is caused by an RTS of one packet, that does not get through (e.g. the target moved away) and is infinitely retransmitted. Therefore, at some point the interface queue overflows and new packets simply get dropped. In our studies this caused a massive break down in performance for very moderate MAC loads and is – according to the IEEE 802.11 standard section 9.2.4 – an implementation bug.

Acknowledgments– Parts of this work were done within the framework of the 'Fleetnet – Internet on the Road' project which is partly funded by the German Ministry of Education and Research (BMB+F) under grant 01AK025D. H. H. would like to thank his colleague Xavier Pérez Costa for valuable comments on drafts of this paper.

References

[BBC+01] L. Blazevic, L. Buttyan, S. Capkun, S. Giordano, J. Hubaux, and J. Le Boudec. Self-organization in mobile ad-hoc networks: the approach of terminodes. *IEEE Communications Magazine*, 2001.

[BMJ+98] J. Broch, D. Maltz, D. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proc. MOBICOM'98*, pages 85–97, Dallas, TX, USA, 1998.

[BMSU99] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *Proc. of 3rd ACM DIAL M99*, pages 48–55, 1999.

[gri] Grid modules for ns-2. http://www.pdos.lcs.mit.edu/grid/sim/index.html.

[Kar00] B. N. Karp. *Geographic Routing for Wireless Networks*. PhD thesis, Harvard University, 2000.

[LJC+00] J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proc. MOBICOM'2000*, pages 120–130, Boston, MA, USA, 2000.

[Man] The IETF Working Group on Mobile Ad Hoc Networks (MANET). http://www.ietf.org/html.charters/manet-charter.html.

[MWH01] M. Mauve, J. Widmer, and H. Hartenstein. A survey on position-based routing in mobile ad hoc networks. *IEEE Network Magazine*, 15(6):30–39, November 2001.

[ns] Network Simulator (ns), version 2. http://www.isi.edu/nsnam/ns.

[TK84] H. Takagi and L. Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *IEEE Trans. on Comm.*, 32(3):246–257, March 1984.