E2DSR: Preliminary Implementation Results and Performance Evaluation of an Energy Efficient Routing Protocol for Wireless Ad Hoc Networks

Vahid Talooki, Hugo Marques, Jonathan Rodriguez[1], Hugo Água, Nelson Blanco, and Luís Miguel Campos [2]

[1] Instituto de Telecomunicações,
Campus de Santiago, Aveiro
{vahid,hugo.marques,jonathan}@av.it.pt
[2] PDM&FC,
Av. Conde Valbom n. 30, Piso 3, Lisboa
{hugo.agua,nelson.blanco,luis.campos}@pdmfc.com

Abstract. The energy consumption in mobile devices has been identified as a growing problem in network access. Maintaining devices connected to the network for long periods of time quickly depletes their already limited battery capacity. This problem gets worst in wireless ad hoc environments where devices besides the energy consumption associated with normal behavior, also support routing functions, consuming additional energy for packet relaying functions. In order o improve the lifetime and hence the performance of wireless ad hoc networks, this paper discusses the specification for an energy aware wireless routing protocol called E2DSR and describes the protocol implementation both in ns-2 and sensor hardware. E2DSR uses some mechanisms of Dynamic Source Routing (DSR) protocol, but defines a new structure for control packets, changes the routing behavior in nodes, implements a new "Energy Table" and creates a whole new algorithm for route cache and selection.

Keywords: DSR, E2DSR, energy efficiency, flat routing, load balancing, mobile ad hoc networks, routing protocol, wireless networks.

1 Introduction

In pure wireless ad hoc networks there is no infrastructure and as such all nodes actively participate in packet routing. This approach makes these networks easy to implement at any place and at any time, independent from electrical power source availability. In order have a fully functional wireless ad hoc network, every node in the network is eligible to execute packet relaying functions, supporting routing for the network. However, packet relaying is not fairly distributed amongst nodes in a network, some nodes due to its position in the network, get unfairly burden with traffic, relaying packets to multiple nodes in the network. This load has two implications: traffic bottleneck and energy consumption for the node itself. Traffic

J. Rodriguez, R. Tafazolli, C. Verikoukis (Eds.): MOBIMEDIA 2010, LNICST 77, pp. 151–165, 2012.

bottleneck means lower network performance, higher delay and eventually higher jitter. Energy consumption leads to earlier node failure in the overloaded nodes, causing an even higher unstable network topology that can lead to network partitioning, decreasing route reliability [1].

The authors didn't found any wireless routing protocols capable of conveying these issues without causing drawbacks such as requiring global topology information, increase delay or even create a blocking issue [3,4,5,6] (the blocking issue happens for example, when a source node is impeded by a timeout timer to start a data transmission before receiving all replies for a route request message).

Taking this in consideration, this paper discusses the Energy Efficient Dynamic Source Routing (E2DSR) protocol [7], an energy efficient routing protocol for wireless ad hoc networks. E2DSR uses some mechanisms of Dynamic Source Routing Protocol (DSR) [8], defines a new structure for control packets, changes the routing behavior in nodes, implements a new "Energy Table" and creates a whole new algorithm for route cache and selection. With this new approach, all known routes to a destination are evaluated with respect to three metrics: i) length of route; ii) freshness of route and iii) energy level. The route that better satisfies these metrics is the selected route to be used. Preliminary results from an early version of this protocol showed that performance metrics such as traffic balancing, power consumption balancing, and average end-to-end delay were improved when compared to other protocols [9,10].

The remainder of this paper is organized as follows: section 2 makes an overview of E2DSR; Section 3 describes current implementation work, both on network simulator 2 (ns-2) and hardware implementation in sensors; this section also presents the preliminary results of implementation work, finally Section 4 presents conclusions and future work.

2 E2DSR Overview

E2DSR is a hybrid source routing protocol. It's hybrid because it uses source routing only to create and maintain a *routing cache*, data packets are forwarded according to destination address and not to source routing. The *routing cache* in E2DSR is basically similar to a routing table in routing protocols. The *routing cache* however does not contain only the best route to a destination, instead it contains all *K* better routes to a destination, being *K* a configurable variable, related to energy. *Routing cache* is populated with routes discovered in the route discovery phase of E2DSR.

2.1 Route Discovery

In E2DSR, routes to unknown destinations can be learned by two different strategies: (i) by overhearing routing control packets from neighbor nodes or (ii) by broadcasting (flooding) a *Route Request* (RREQ) control packet to neighbor nodes. If using the later, the RREQ includes the source address (S), destination address (D), a hop count

field and an energy field (in form of an array) that includes the energy level of every node in the path. In this case, the energy field will only include the energy level of S.

Upon receiving a RREQ, each node needs to process it accordingly. In E2DSR there are three types of nodes, *Source* nodes, *Intermediary* nodes and *Destination* nodes.

2.1.1 Behavior of Intermediate Nodes

Contrary to DSR, in E2DSR the intermediate node forwards a maximum of *K* RREQs (from S to D). When the first RREQ from S arrives to an intermediate node it stores it in its *request table* (as in DSR), extracts the energy array to calculates the energy parameter of path (energy parameter is discussed in section 2.4.4), being E_1 the energy parameter of first RREQ, and then processes it to check if it already knows a path to D. If a path is known, then the node creates a Route Reply (RREP) message back to S that includes its own address and the addresses of all other nodes in the path to reach D, and its own energy level as well as energy level for each node in the path to D.

If this node doesn't have any route to D, it will change the received RREQ packet by putting its own address and energy level in the correspondent address and energy fields of the RREQ message. The RREQ is then broadcasted to all of its neighbors, with the objective of reaching D (flooding process).

The intermediate node then caches all subsequent received RREQs (during a configurable time window) and will choose the *K-1* routes with the highest energy level to reply. To do that, the intermediate node will extract the energy array of every RREQ and then compares it to E_1. Only the RREQs that have a better energy level than E_1 should be forwarded. To achieve this goal, we save *E* value (which is E_1 plus a threshold) in its request table (*E* is introduced in equation 1). The value of threshold is controlled by a coefficient which is set to a reasonable value C_e=0.2 in equation 1. A bigger coefficient yields a bigger threshold. The time at which the first RREQ is received (T_1) will also be saved on the request table and a related timeout clock ($T_{Inter.Wait}$) will be started.

$$E = E_1 + C_e(1 - E_1)$$
$$0 \le E \le 1;\ \ 0 \le E_1 \le 1;\ \mathrm{C_e} = 0.2 \tag{1}$$

When the *ith* route request ($RREQ_i$) arrives, the intermediate node will again calculate E_i (energy parameter of $RREQ_i$) and if $E_i > E$ then it forwards $RREQ_i$ and also updates E by E_i (i.e. $E=E_i$) otherwise it simply drops $RREQ_i$. By using this RREQ processing mechanism, each intermediate node will forward *K* RREQ at maximum; also the intermediate node will not forward RREQs that have arrived after $T_{Inter.Wait}$ timer expires. By varying *K* and $T_{Inter.Wait}$, the behavior of the protocol can be adjusted to be more efficient. The chosen value for *K* is 3. Higher values for *K* imply forwarding and processing more RREQs which have a significant impact in traffic overhead and energy consumption. A smaller value for *K* limits the number of redundant routes in a node's *routing cache*, which can lead to higher delays and also to more traffic in the network, due to the discovery phase flooding mechanism.

Also, by default $T_{Inter.Wait}$ is set to 2 seconds because it's assumed that a RREQ which is received after this reasonable long time, has encountered problems inside its path (such as traffic congestion or interference), so it should be ignored.

2.1.2 Behavior of Destination Nodes

In E2DSR, destination nodes will immediately reply to the first received RREQ (avoiding the blocking problem described in Section 1) and also to the subsequent *K'*-1 received RREQs, within time window $T_{Dest.Wait}$, that have the highest energy. By changing *K'* and $T_{Dest.Wait}$, we can customize our protocol for most efficiency; for destination nodes their default values are K'=3 and $T_{Dest.Wait}$=4s.

2.1.3 Behavior of Source Nodes

After S receives first RREP to D, it immediately starts forwarding packets to D via that route. However, taking in consideration the behavior of *Intermediate* and *Destination* nodes in E2DSR, *S* can still receive other RREPs with higher energy level, shortly after receiving the first RREP. After $T_{Source.Wait}$ seconds, S runs a new E2DSR function, called *Route Priority Function*, that calculates the priority of each discovered route (see section 2.4.1). The chosen route to a destination, to use for data communication, will be the one that has the highest priority. This (re)selection process doesn't add additional delay since it can run in the background.

For scenarios where the exchange of data between S and D takes too long, the energy of the nodes in the path will quickly change and some nodes will eventually reach a critical battery level. To avoid this, during the data transmission time window, S may need to change the route to D to an alternate route, one that has better energy.

Source nodes should run *Route Priority Function* in the following cases:

- The first run should happen after receiving the first RREP, at the beginning of the communication, $T_{run1} = 0$.
- The second run is at time $T_{run2} = T_{Source.Wait} = 6s$. Note that during this time, *S* has probably received other routes to *D*.
- The next runs will take place each $T_{interval} = 180s$. This number is chosen for test setup and can be changed based on new results.

By using this mechanism, source nodes will balance the energy consumption between the best routes they have to a specific destination.

2.2 Control Packets in E2DSR

Control packets are used by E2DSR in order to discover and maintain routes. Shows how these control packets are handled.

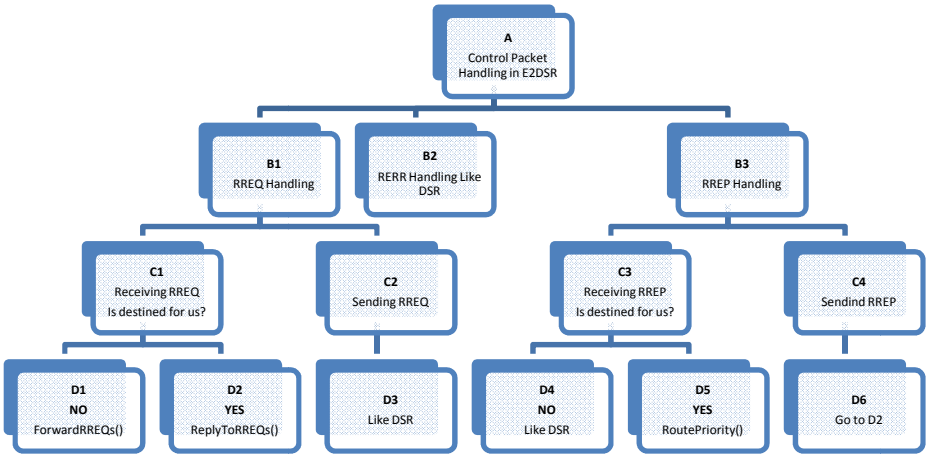

Fig. 1. Control packet handling in E2DSR

2.2.1 Structure of Route Request (RREQ) Routing Control Packets

In E2DSR the RREQ has an energy field in the form of an array. This energy array field contains the energy value (battery remaining power) of each node that forwarded the RREQ. Nodes append their energy value to the end of the energy array field. The energy value for each node is implemented by a 4 bit field which gives 16 different levels. Level zero means battery critical and level 15 means full battery. The use of 16 levels gives a good granularity level while maintaining header size at a reasonable overhead.

2.2.2 Structure of Route Reply (RREP) Routing Control Packets

In E2DSR the RREP message was also modified to include the energy array field. *Destination* nodes, upon receiving a RREQ message, create a RREP message, add an energy array field that is a copy of the one received in the RREQ message, and send the RREP to the originator of the RREQ. If a RREQ is received by an intermediate node that already has a route to destination, this node uses the energy values of that route when responding to the RREQ message.

The energy array of the new RREQ and RREP message directly shows the energy level of a route.

2.3 The Energy Table

In E2DSR, nodes create a table called *Energy Table*, used to save the energy value of discovered nodes. Energy values are extracted from received or overheard RREQs and RREPs. The *energy table* will then allow a node to determine the energy level of whole path towards a destination. Every time S listens a RREP or a RREQ it will update both the *routing cache* and *energy table*.

2.4 Route Selection

In E2DSR a *Route Priority Function* is defined to determine the priority of each discovered route. For a specific source and destination pair, the route which has the maximum priority will be selected as the best route between candidate routes. The *Route Priority Function* has three input parameters with respect to each route: (i) *length*; (ii) *freshness* and; (iii) *energy* of path. These parameters have values normalized between 0 and 1 and are explained in the next subsections.

2.4.1 Priority of a Path

The *PathPriority* function uses (2) to compute the priority of a path.

$$PathPriority(i) = \frac{K_F \cdot F(i) + K_E \cdot E(i)}{K_L \cdot L(i)} \tag{2}$$

Where K_F, K_E and K_L are the coefficients for *freshness*, *energy* and *length* of route, respectively. Desirable values for these coefficients, obtained through simulation, are K_F=1 K_E=3 and K_L=1. It's possible to achieve a higher performance regarding a special metric by giving a higher weight to the related parameter in (2). For example, for end-to-end delay the *PathPriority* function can be customized by a higher coefficient for length parameter (K_L), because delay is typically more dependent on the length of routes.

2.4.2 *Freshness* Parameter

Measuring the freshness of a route is especially important in wireless ad hoc networks due to its dynamic nature. Node's movement constantly changes the validity of a route. Therefore every entry in the node's routing cache should have information regarding how fresh that route is. A route can be old and fresh at the same time, if it was learned a long time ago but used recently. The *F(i)* parameter in (2), indicates the freshness of route *i* and can be measured according to (3):

$$F(i) = \frac{n - i + 1}{n} \tag{3}$$

The freshest route (the one learned or used more recently) has a freshness value of 1 and, for n routes, the oldest route as a freshness value of 1/n. All other routes have freshness values between 1 and 1/n.

2.4.3 *Length* Parameter

Longer routes can increase the delay in an ad hoc network. Also longer routes have more links, which leads to a higher probability of link breakage. The *L(i)* parameter in (2) indicates the length of route *i* and can be measured according to (4):

$$L(i) = \frac{Length\_of\_Route(i)}{Max\_Length} \tag{4}$$

Length_of_Route(i) is the length, in number of hops, for route *i*. *Max_Length* is the maximum length that a route can have in DSR routing protocol, default value is set to 16.

2.4.4 Energy Parameter

The energy level of a specific route is an important characteristic of that route, since it can assure that a link will not go down due to power issues. However, a route may have nodes that in average have good battery levels and, at the same time, have some nodes with low battery levels. Since there is a high probability of failure in the low battery nodes, this route is undesirable.

The *E(i)* parameter in (2) indicates the energy of route *i* and can be measured according to (5).

$$E(i) = \frac{RE(i) \cdot MRE(i)}{M(i) \cdot InitialEnergy^2} \tag{5}$$

M(i) represents the number of nodes in route *i*. *InitialEnergy* is a constant that defines the maximum energy that a node can have. *RE(i)* represents the total of the remaining energy in route *i*. *MRE(i)* is the minimum of the remaining energy between all nodes of route *i*. MRE(i) is important because it will allow the detection of the low battery nodes problem described previously. Knowing that *MRE(i)* is a part of *RE(i)* and is taken into account twice, the E*(i)* calculation presented in (5) will be modified to the one presented in (6).

$$E(i) = \frac{(RE(i) - MRE(i)) \cdot MRE(i)}{M(i) \cdot InitialEnergy^2} \tag{6}$$

2.5 Route Maintenance

Routing maintenance process in E2DSR is like DSR protocol. When a node, by forwarding a packet from S to D, discovers a link breakage it must send *Route Error* (RERR) control packet back to S. By using this mechanism, all nodes in path to the source will learn about this link breakage and update their *routing cache* accordingly.

3 E2DSR Implementation

Currently, E2DSR is being implemented in both simulation environment and real sensors. Validation through simulation is being done with ns-2 [11] since it has proven to provide trusted results and it is also widely accepted by the academic research community.

The implementation in real sensors is being done in TinyOS 2.1 [12] and being tested on Telos ultra low power IEEE 802.15.4 compliant wireless sensor modules (revision B) [13]. Code regarding E2DSR implementation in sensors is first done in TOSSIM simulator [14], where it is tested, and then exported to the sensors. Currently we have a testbed of ten sensors and have successfully implemented RREQ, RREP

and RERR primitives and *Route Priority Function*, as described in the previous sections. The testing in real sensors is allowing us to fine tune some E2DSR coefficients.

3.1 Implementation in ns-2

ns-2 [11] is a discrete event simulator where the advance of time depends on the timing of events maintained by a scheduler. *ns-2* uses two languages: (i) C++ for the object oriented simulator, and, (ii) an OTcl interpreter for writing and executing user's scripts (simulation scenarios). The OTcl can make use of the objects compiled in C++ through a TclCL (Tcl/C++ interface) linkage. The use of C++ in the simulator code has the advantage of reducing packet and event processing time, allowing it to achieve fast execution times. The use of OTcl, despite being a compact and very powerful object programming language, allows a more intuitive interface with the user, when compared to C++. The development of *ns-2* started in 1989 and has evolved substantially over the past few years; it has been supported mainly by DARPA, NSF and the constantly growing *ns-2* community. Bugs in the *ns-2* software are still being discovered and corrected; each user is responsible for verifying that his simulation is not invalidated by bugs. Additionally, due to its old code, output files are somewhat difficult to understand and users need to parse them in order to obtain the desired results. In order to improve output understanding, it is current practice to put 'debugs' and 'printfs' in *ns-2* code. Nevertheless, *ns-2* is considered a feasible simulator and as such, it's largely used by academic researchers.

Specific implementation details for the E2DSR primitives can be found in [7].

3.1.1 Simulation Scenario and Results

3.1.1.1 Testing E2DSR implementation. Up to now, E2DSR was simulated in ns-2 using a topology of 30 mobile nodes. Table 1 shows the other relevant ns-2 simulation parameters used for testing E2DSR implementation.

Table 1. Simulation parameters

Topology area	700m x 700m
Maximum mobility of nodes	10m/s
Paused time	50s
Number of nodes	30
Simulation time	200s
Traffic sources	CBR
Data packets size	512 bytes
Sending rate	8 packets/second
Maximum no. connections	10

By using this scenario, Fig. 2 shows that at time 29.7s, node 17 executes the *Route Priority Function* in order to find the best path to node 19. We can see that node 17 has in its *routing cache*, five paths to node 19. At the end the figure we can see that path 17-29-5-19 is the one with better priority; and so, is the selected by node 17 to route packets to node 19.

```
↓
Current Intermediate Node is: 17↓
vcurrent_time=29.733183↓
↓
Route Path is: [(17) 24 29 28 19 ]↓
Route_Priority=0.2584↓
Route Path is: [(17) 29 5 19 ]↓
Route_Priority=0.3975↓
Route Path is: [(17) 11 1 34 36 28 19 ]↓
Route_Priority=0.2247↓
Route Path is: [(17) 29 5 19 ]↓
Route_Priority=0.3975↓
Route Path is: [(17) 29 5 19 ]↓
Route_Priority=0.3975↓
↓
Number of Found Path=6↓
Best Route Priority is=0.3975↓
Best Route Path is: [(17) 29 5 19 ]↓
↓
```

Fig. 2. Route selection process in E2DSR

Fig. 3 shows that at time 45.5s node 17 receives a RREQ sent by node 1 and forwarded by node 9. Upon receiving this RREQ, node 17 will:

- Add self to the end of RREQ address array field. New path becomes [(1) 9 17];
- Add its energy level to the end of RREQ energy array field. The values 9, 12, and 11 (last line in figure) shows the energy level of nodes 1, 9 and 17, respectively.
- Update its *energy table*. By extracting the energy array field of RREQ, node 17 can learn or update the energy level of nodes 1 and 9.

```
↓
Current Intermediate Node is: 17↓
vcurrent_time=45.502319↓
Current Node Energy is:11↓
Route Path is: [(1) 9 ]↓
Add current node to RREQ↓
Route Path is: [(1) 9 17 ]↓
Request Energy Field Array Length is: 3↓
Eng.RREQ[0]=9  Eng.RREQ[1]=12  Eng.RREQ[2]=11  ↓
↓
```

Fig. 3. Node 17 receives a RREQ and updates its energy table

↓
Current Intermediate Node is: 17↓
vcurrent_time=4↓
Current Node Energy is: 14↓

Node Address	Node Energy	Time↓
18	15	3↓
3	15	3↓

↓

a) At time=4s

↓
Current Intermediate Node is: 17↓
vcurrent_time=45.502319↓
Current Node Energy is: 11↓

Node Address	Node Energy	Time↓
18	15	3↓
3	15	3↓
4	11	40↓
24	11	40↓
1	9	45↓
9	12	45↓
25	11	43↓
7	13	43↓
29	12	37↓
21	9	37↓
28	14	4↓
11	12	16↓
16	9	41↓
14	14	3↓
23	10	42↓
19	11	39↓
22	14	7↓
27	13	31↓

↓

b) At time=45.5s.

Fig. 4. Energy *table* of node 17

The *energy table* is dynamic through all simulation time. Fig. 4 a) and b) show the *energy table* of node 17 at time 4s and 45.5s. We can see that as simulation proceeds, node 17 learns about the presence of other nodes in the network.

3.1.1.2 Testing E2DSR Performance. In this section we will provide preliminary performance results that will allow measuring the effectiveness of the energy balancing algorithm used by E2DSR. Two Table 1 based scenarios were used:

- Scenario 1: paused time varies between 0s (nodes are always moving) and 200s (nodes don't move);
- Scenario 2: nodes' velocity varies from 0m/s (static nodes) to 18m/s. In this scenario we use a fixed paused time value of 40s.

Values are computed based on 100 iterations of the simulation, using a standard deviation of the remaining energy for all nodes. Hence the energy load for each node i, $EL(i)$, is the relation between the consumed energy in node i and the total consumed energy in all nodes in the network. $EL(i)$ can be computed according to (7).

$$EL(i) = \frac{ConsumedEnergy(i)}{TotalConsumedEnergy} \tag{7}$$

The value of the deviation will be the metric for energy consumption balancing of the protocol; the smaller the deviation, the more effective is energy balancing.

By using (7), Fig. 5 and Fig. 6 show the performance of both DSR and E2DSR protocols when using scenario 1 and 2 respectively. Our preliminary results show E2DSR has a higher performance than DSR regarding energy balancing and shows lower deviation in energy consumption.

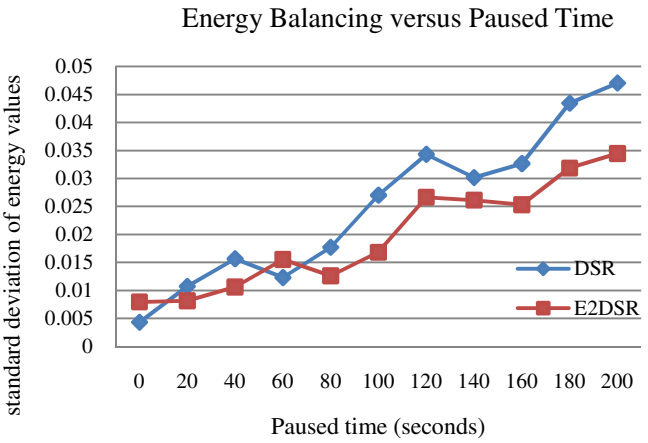

Fig. 5. Energy balancing vs. paused time

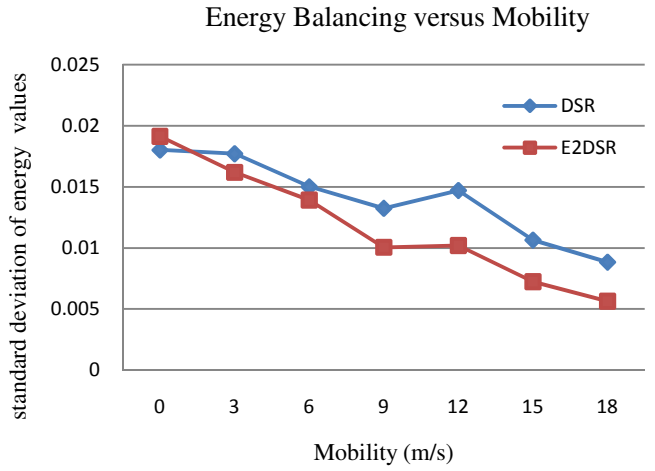

Fig. 6. Energy balancing vs. mobility

3.2 Implementation in Sensors

The E2DSR prototype was implemented in TinyOS 2.1.1 stable release. It was tested on Xbow Telos B motes and on a modified version of TOSSIM simulator. We added to TOSSIM a CC2420 radio stack model. The simulator has node mobility support, energy consumption estimation during simulations, RSSI value measurement and enhanced trace capabilities.

In order to add mobility, an application was implemented in Python language that allows customization for the movement of nodes within the network. This application reads the network changes from a text file, and the specific instants at which the changes occur.

3.2.1 Implementation Scenario and Results

We consider 3 mobile nodes, travelling randomly amongst clusters of 5 nodes, with an average connectivity time, to at least one cluster, of 1'26". At this testbed we force mobile nodes to be active seekers; they continuously send RREQ packets in order to find a route to a specific destination node. We also consider that sensors are always in a 'full on' state. After obtaining a route to the destination, the sensor sends CBR traffic, 10 packets of 35 bytes every 100 ms (approx 28Kbps).

It is important to notice that the active seeker approach does not significantly increase the energy consumption when compared to a passive seeker approach. The only relevant issue related to the active seeker approach is a higher packet loss ratio due to collisions with preambles. Simulation results measured an average of 0.26 data frames lost by neighbor nodes whenever a source node sends frames. However, the routing protocol retransmitted all lost frames and concealed the packet losses from the application.

A. Average end-to-end delay

In E2DSR the best three routes (based on the metrics described in [7]) to a destination are stored in a cache, and whenever a new route is required, the *RoutePriority* function is executed.

When E2DSR is customized to $E2DSR_l$ by giving a higher coefficient to the *length* of the route (which can decrease transfer time), it shows a better performance with respect to end-to-end delay, up to 20% fewer deviations than DSR by varying pause time or mobility, as shown in Fig. 7 and Fig. 8.

B. Throughput

Throughput is a measure of the number of bytes per second received by a mobile node. In a wireless ad hoc environment, throughput should be measured taking in consideration the frequently changing network topology of such networks. In our testbed we trigger these topology changes by varying the velocity of mobile nodes. Fig. 8 shows a comparison of the evolution of throughput when using E2DSR or DSR. During the measurements the destination node was at an average distance of 8 hops from the mobile node.

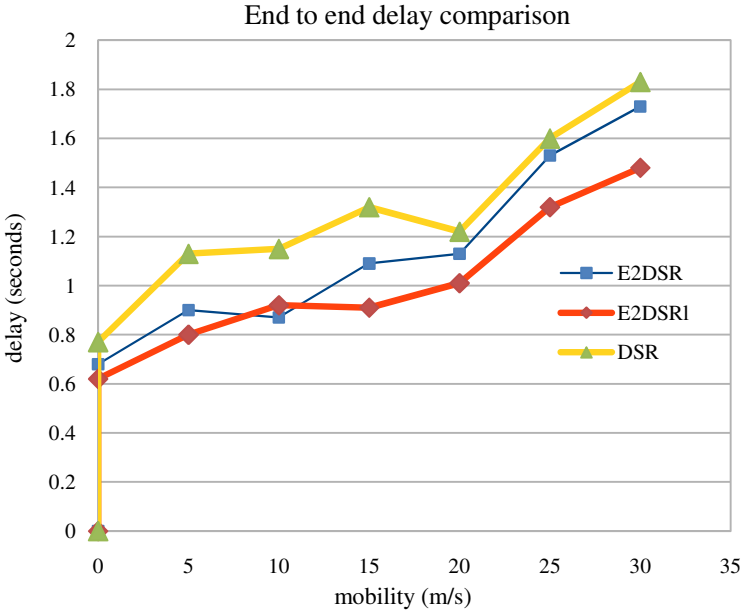

Fig. 7. End-to-end delay comparison between E2DSR and DSR

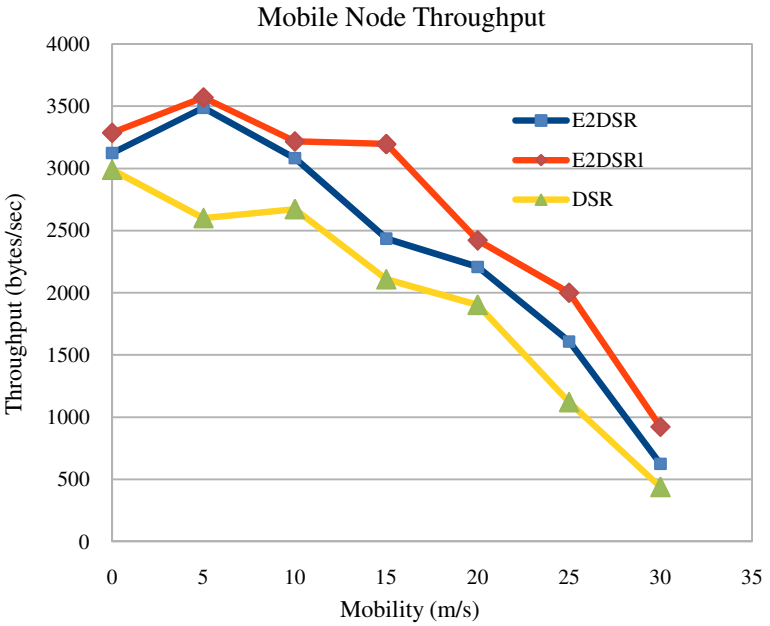

Fig. 8. Throughput comparison between E2DSR and DSR

4 Conclusions and Future Work

Even though a full evaluation of E2DSR is not possible at this moment, the preliminary results are quite promising. We shown that E2DSR algorithm performs better energy balancing and improves energy consumption when compared to well known DSR. E2DSR also shows an improvement of approximately 20% in end-to-end delay and throughput with regards to a well known protocol as DSR. These results are sufficient enough to motivate for the continued development of E2DSR. We also take in consideration that a previous version of E2DSR [9, 10][1], was already compared to other protocols, including AODV, using performance metrics such as jitter and overhead and proved to be efficient.

E2DSR is being developed to work on sensors as in other mobile devices, hence it is continuously being fine tuned according to the results received by both our simulation platform and implementation testbed; for example, operations like divisions are more demanding for the limited CPU resources of a sensor, so metric calculations needed to be rethink in order to avoid these and other complicated operations. Other limitation is related to the limited bandwidth of sensor wireless links; using pure flooding mechanisms for route discover and maintenance has a direct impact in data throughput, on the other hand, avoiding flooding mechanisms will require a complex node-to-node signaling protocol that consumes memory and CPU. The solution we present for E2DSR seems a nice balance between these considerations.

Also currently the energy field in E2DSR uses a linear quantization; however our latest studies indicate that if we give more granularity to the lower levels of battery energy, by using a non-linear quantization method, we will be able to achieve a more effective energy balance.

Acknowledgments. The work presented in this paper was undertaken in the context of the project INFSO-ICT- 225654 PEACE (IP-Based Emergency Applications and ServiCes for Next Generation Networks), which has received research funding from the European 7th Framework Programme. The authors would like to acknowledge the contributions of their colleagues from the PEACE consortium.

References

1. Woo, K., Lee, B.: Non blocking localized routing algorithm for balanced energy consumption in mobile ad hoc networks. In: IEEE MASCOTS 2001, pp. 15–18 (2001)
2. Royer, E., Toh, C.: A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks. IEEE Personal Communications 4(2), 46–55 (1999)
3. Singh, S., Woo, M., Raghavendra, C.: Power-Aware Routing in Mobile AD Hoc Networks. In: International Conference on Mobile Computing and Networking, MobiCom 1998, pp. 181–190 (1998)

[1] Load Balancing Routing Protocol (LBDSR) – is also being developed but it is not meant to work on sensors, due to memory and processor requirements.

4. Chang, J., Tassiulas, L.: Energy Conserving Routing in Wireless Ad Hoc Networks. In: Conference on Computer Communications IEEE Infocom, pp. 22–31 (2000)
5. Zhou, A., Hassanein, H.: Load-Balanced Wireless Ad Hoc Routing. In: Canadian Conference on Electrical and Computer Engineering, pp. 1157–1161 (2002)
6. Chakrabarti, G., Kulkarni, S.: Load Balancing and resource reservation in mobile ad hoc networks. Ad Hoc Networks 4(2) (2006)
7. Talooki, V., Marques, H., Rodriguez, J., Água, H., Blanco, N., Campos, L.: An Energy Efficient Flat Routing Protocol for Wireless Ad hoc Networks. In: 1st IEEE International Workshop on Convergence of Heterogeneous Wireless Systems, International Conference on Computer Communication Networks, ICCCN 2010 (2010)
8. IETEF Draft: The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4 (2007), http://tools.ietf.org/html/rfc4728
9. Talooki, V., Rodriguez, J.: Jitter Based Comparisons for Routing Protocols in Mobile Ad hoc Networks. In: IEEE Workshop on Wireless and Optical Networks (WI-OPT 2009), Saint-Petersburg, Russia (2009)
10. Talooki, V., Rodriguez, J., Sadeghi, R.: A Load Balanced Aware Routing Protocol for Wireless Ad Hoc Network. In: 16th IEEE International Conference on Telecommunications, Marrakech, Morocco (2009)
11. The Network simulator ns-2 (2010), http://www.isi.edu/nsnam/ns
12. TinyOS: open-source operating system designed for wireless embedded sensor networks, http://www.tinyos.net/
13. Polastre, J., Szewczyk, R., Culler, D.: Telos: enabling ultra-low power wireless research. In: 4th International Symposium on Information Processing in Sensor Networks. IEEE Press (2005)
14. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In: ACM International Conference on Embedded Networked Sensor Systems (SenSys), pp. 126–137 (2003)