D-sense: An Integrated Environment for Algorithm Design and Protocol Implementation in Wireless Sensor Networks

Kazushi Ikeda[1], Shunsuke Mori[1], Yuya Ota[1†], Takaaki Umedu[12], Akihito Hiromori[12], Hirozumi Yamaguchi[12], and Teruo Higashino[12]

[1] Graduate School of Information Science and Technology, Osaka University
[2] Japan Science Technology and Agency, CREST
(k-ikeda,s-mori,umedu,hiromori,h-yamagu,higashino)@ist.osaka-u.ac.jp
ohta@am.sanken.osaka-u.ac.jp[†]

Abstract. Since Wireless Sensor Networks (WSNs) are regarded as large-scale distributed systems in nature, it is (i) difficult to implement their distributed low-level codes, (ii) hard to analyze their performance and (iii) almost impossible to operate a number of nodes manually. In this paper, we propose an integrated environment called *D-sense* to solve these problems in WSN development. By providing algorithm-level APIs, D-sense tries to hide distributed, low-level operations in the NesC programming language. The algorithm-level APIs and other NesC codes can automatically be converted into simulator codes to avoid code-writing for simulation purpose. In addition, D-sense provides useful functions like monitoring, logging and debugging of distributed programs. We have implemented several known protocols and evaluated the performance by simulation and real environmental experiments to demonstrate the functions of D-sense.

1 Introduction

In Wireless Sensor Networks (WSNs), due to heterogeneity of architecture, network scale and applications, new protocols are often developed or existing protocols are tuned accordingly. Thus many protocols have been designed with different design goals [1–8]. However, protocol designers and developers face with typical problems which have been experienced in designing distributed systems. Even though the developers wish to concentrate on abstract behavior of protocols, they at last need to write target-dependent low-level codes. Then they carry out performance analysis and validation in simulated networks or real environment. However, additional effort may be required to prepare another implementation of the protocol for network simulators, since in most cases it is not compatible with real codes. Also experiments in real environment require to set up many sensor nodes, to log their behavior, and to manipulate them to validate (debug) the implementation. Obviously all of these tasks are really hard and complex.

In this paper, we design and develop *D-sense*, an integrated development environment to support protocol development in WSNs efficiently. D-sense mainly assumes NesC on TinyOS as the target language and architecture, and experiments have been carried out on Mica Motes accordingly. However, for other languages such as C or Java, D-sense's design concept can also be applied. The advantages of D-sense are three-fold. First, D-sense offers algorithm-level APIs which are derived by classifying and studying existing protocols. Thus developers can design distributed algorithms in NesC language directly with these APIs. Secondly, it enables seamless integration of simulated and real sensor networks. To accomplish this, we provide a translator from NesC codes into QualNet simulator application codes. Also the physically sensed events and sensor node status observed in real environment are made available in the simulator. These capabilities increase repeatability and fidelity of experiments. Thirdly, monitoring and run-time manipulation of sensor node behavior is possible. We will later show how this functionality can powerfully support developers in test and maintenance of WSN protocols.

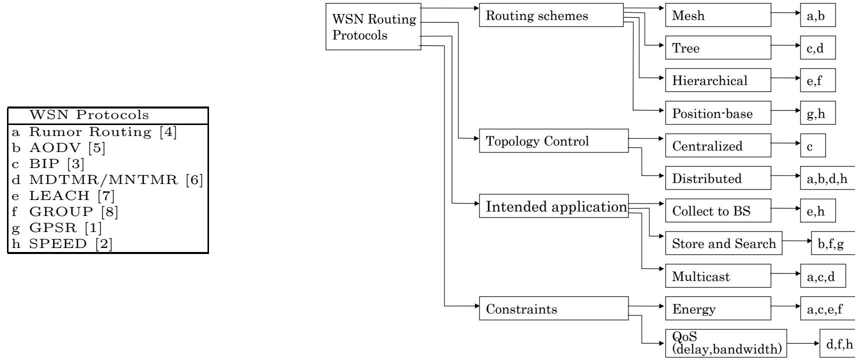
Using the D-sense APIs, we have implemented GPSR [1], SPEED [2], BIP [3] and Rumor Routing [4]. In particular, we have evaluated the performance of SPEED in both real and simulated networks and compared the results with [2] to validate the D-sense implementation. It was also confirmed how D-sense contributed to alleviate the development cost.

This paper is organized as follows. In Section 2, we address the related work and show the features of D-sense. In Section 3, we describe the functions of D-sense that support design, experiments and protocol debugging of WSN. In Section 4, we show example implementation of the existing WSN protocols by using the D-sense design APIs, and Section 5 shows the experimental results. Section 6 concludes the paper.

2 Related Work

Large-scale testbeds such as MoteLab [9] and CitySense [10] usually provide management functions like online distribution of execution codes to mitigate maintenance costs. D-sense differs from them since it is aimed at comprehensive support of design, development and performance analysis.

TinyDB [11] and COUGAR [12] support designing query processing in sensor networks. They provide SQL-like APIs to implement event acquisition and search processes. MATE [13] also provides APIs for more generic purposes, but only low-level APIs like sensing events, pushing data to stack or sending data are designed. EnviroSuite [14] is an object-based programming model framework for tracking and environmental monitoring. Meanwhile, we attempt to help high-level design of more generic protocols including geographic/random-based routing and data fusion/diffusion by extracting their typical behavior. This appropriately hides both distributed and low-level behavior so that developers concentrate on algorithm description. For example, geographic routing protocols like GPSR which employ greedy forwarding strategy need a series of the following



	WSN Protocols
a	Rumor Routing [4]
b	AODV [5]
c	BIP [3]
d	MDTMR/MNTMR [6]
e	LEACH [7]
f	GROUP [8]
g	GPSR [1]
h	SPEED [2]

Fig. 1. Criteria in Classification of WSN Protocols

atomic actions at each node; (i) obtaining positions of neighbors, (ii) computing distances between the node and the neighbors and between the neighbors and the destination, (iii) finding the neighbor which is closer to the destination than the node and is the closest to the destination among the other neighbors, and (iv) sending a packet. D-sense defines an API for each atomic action, and also provides a single API for a series of these actions by using those atomic APIs.

In summary, as far as we know, no environment has been provided that comprehensively supports algorithm design, low-level implementation, seamless use of simulator and real terminals, and online debugging/monitoring in real environment.

3 Functions of D-sense

3.1 High-level Design Support

One of the most important features of D-sense is high-level design support. Using the *D-sense design APIs*, developers can give algorithm-level NesC descriptions. Then the *D-sense design API translator* takes them as inputs, and expands the embedded APIs that are implemented as macros into pure NesC implementation automatically. In order to support as many types of protocols as possible, the D-sense design APIs are developed based on property analysis of existing typical protocols. These protocols are classified by the criteria which are inspired from [15]. A classification example by these criteria is given in Figure 1. For example, GPSR (“g” in Figure 1) is a position-based routing method and is used in GHT [16] or some other methods that employ position-based event accumulation and search mechanisms. In implementing this protocol, we may use the APIs for

Table 1. Part of D-sense Design API

Generic APIs
Get the IDs of the one hop neighbor nodes -> get_neighbors(IDs[],len_IDs) Send a packet to the designated node -> send_unicast_packet(ID,pkt)
Position base protocol APIs
Get the position of the designated node -> get_position(ID) Compute the distance between the designated two nodes -> get_distance(ID,ID)
Hierarchical protocol APIs
Get the IDs of the cluster members -> get_cluster_nodes(IDs[],len_IDs) Get the cluster head of the neighbor cluster -> get_neighbor_clusterhead()

Energy Constraint Protocol APIs
Get the residual battery of the designated node -> get_residual_energy(ID) Get the energy consumption to send a packet -> get_transmission_energy()
QoS Constraint Protocol APIs
Get the delay between the designated two nodes -> get_delay(ID,ID) Get the packet loss ratio at the designated node -> get_packet_loss_rate(ID)
Functional (Combined) API
Get the ID of the maximum residual battery node in a cluster Get the minimum delay node which has the specific data Get the packet loss ratio in the tree

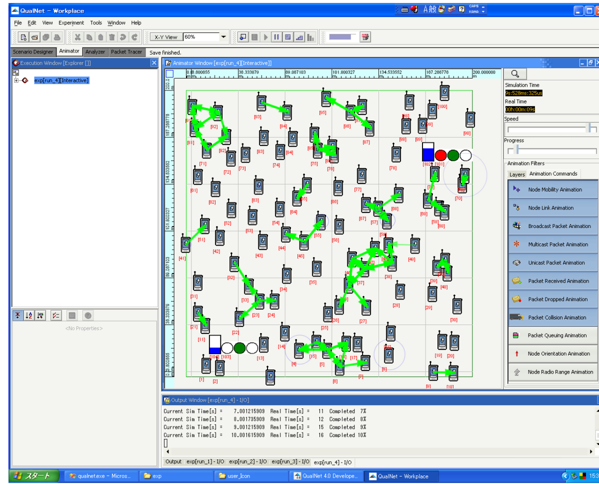


Fig. 2. A Snapshot from QualNet Simulator (Sensor Node Status is Visualized)

"position-based routing" and "store and search application". Similarly, some other known protocols are classified in the figure.

For each type in the classification, we provide type dependent APIs, and also provide generic APIs which are commonly used in all the types. Furthermore, we design more functional APIs that are realized by using these APIs. Part of them is shown in Table 1. Using these D-sense design APIs, developers can write codes concentrating on algorithmic behavior, and other low-level descriptions are hidden. The details of the APIs and their implementation are shown in our web site [17].

In Section 4, we will exemplify how typical protocols are implemented using these APIs.

3.2 Seamless Integration of Simulated and Real networks

The NesC codes derived by the D-sense design API translator can be directly executed on Mica Mote, or can further be translated into codes for QualNet simulator [18] written in C++ by the *D-sense NesC translator*. Also environmental events and sensor node status logged automatically by the D-sense debugging component in real environment and can be animated by the QualNet animator (Figure 2) in which we provide special graphics to visualize residual amount of battery and LED status (we assume Mica MOTE here) for more realistic animation.

3.3 Protocol Debugging Support

We explain our powerful support facility for online debugging. Debugging sensor node software in distributed environments requires many efforts to implement the complicated communication schemes for confirming the node status. To mitigate these efforts, D-sense offers *debug scenario*s which enable developers to get information about the nodes and to make debugging easy. At each node, we run a "debug agent", and at the base station we run a "debug controller". Each debug agent can monitor specified variables on that node, and communicate with the debug controller. The debug controller operates those agents to realize the scenario in distributed environment.

A debug scenario is composed of a *condition* followed by an *action*. As the condition, we may refer to function names which become true when the corresponding function of the NesC code is called. We may also specify boolean expressions over the variables of the NesC code. The action can be specified as a set of statements which are executed when the condition is satisfied.

An important feature is that we may specify the "owners" of the variables and functions that appear in the scenario. These owners may be a specific node or a set of nodes, which are determined statically or dynamically. For the static case, we may directly specify node ID as the prefix of each variable or function. If we wish dynamic assignment to the node ID, we may use symbol $ in the condition, which implies the first node that satisfies the condition. Also we may define a set of nodes that satisfy specified conditions. This node set can be determined statically by debug APIs, or can be defined dynamically in the action part.

In the following part, we give two examples of debug scenarios. The first one is given below.

```
$.on_loop_detected ->|
  nodeIDs = $.receive_packet->recent_visit_nodes;
  foreach u in nodeIDs {
      u.refresh_table();
  }
```

The prefix of a variable or a function indicates its owner node. Those without prefix are variables defined and held by the debug controller (at the base station). Thus, this scenario means that if a sensor node (symbolized by $) calls

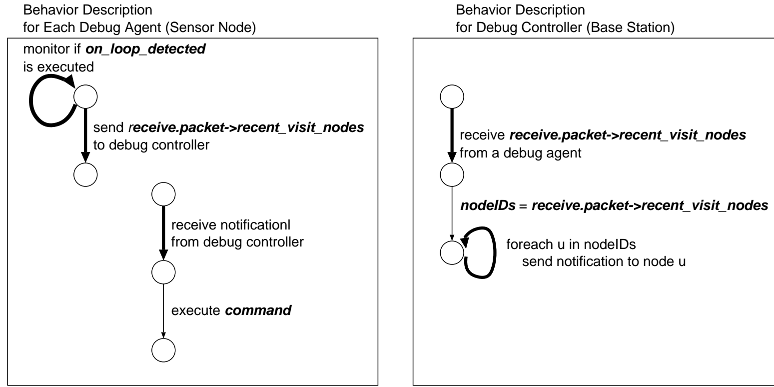


Fig. 3. Distributed Execution of Debug Scenario

on_loop_detected function, then *recent_visit_nodes* (this is a list of nodes) included in structure *receive_packet* of node $ is set on the list variable *nodeIDs* of the base station. Then each node specified in *nodeIDs* executes function *refresh_table()*.

This code can be used as an assertion and a post-condition in routing protocols. Once a loop is detected by a sensor node in forwarding a packet, the routing tables on the nodes that the packet recently traversed are refreshed.

The second example is a scenario for system monitoring and maintenance.

```
ave_energy(S = region(a,b)) < 0.2 ->|
  foreach u in S {
    u.beacon_interval = u.beacon_interval * 2
  }
```

This scenario means that if the average residual energy of the nodes in the square region defined by two corner points a and b (the set of the nodes in the region is obtained by the debug API "$region(a, b)$") is less than 20%, these nodes double their beacon intervals to extend network lifetime of the region. $ave_energy(S)$ is also a debug API that returns the average residual energy of the nodes in set S.

To realize these scenarios by the debug controller and the debug agents, we need to derive protocols to exchange necessary data or notification, and to execute statements. For example, the first example can be distributed over the agents and controller as shown in Figure 3. Formally, each debug agent (or controller) collects arguments to execute a statement that updates its own variable. For example, the first statement of the action part,

```
nodeIDs = $.receive_packet->recent_visit_nodes;
```

is executed by the controller because *nodeIDs* is the variable of the controller. However, the right-hand value is the variable held by the node symbolized by $. Since we do not know which node becomes $, the agent on each node sends this value to the controller whenever *on_loop_detected* function is executed, and the controller updates the value of *nodeIDs*.

```
01:  get_planar_graph(graph g){
02:      len = get_neighbors(neighbor_IDs, sizeof(neighbor_IDs));
03:      for (i = 0; i < len; i++){
04:          nodeID = neighbor_IDs[i];
05:          for (j = 0; j < len; j++){
06:              nodeID' = neighbor_IDs[j];
07:              if (get_distance(myID,nodeID)
                   > max(get_distance(myID,nodeID'),
                       get_distance(nodeID,nodeID')))
08:                  g.remove_edge(myID,nodeID);
09:  }
```

(a) RNG Generation

```
01:  len = get_neighbors(neighbor_IDs, sizeof(neighbor_IDs));
02:  forwardID = get_my_ID();
03:  for (i = 0; i < len; i++){
04:      nodeID = neighbor_IDs[i];
05:      if(get_distance(nodeID,targetID)
           < get_distance(forwardID,targetID))
06:          forwardID = nodeID;
07:  if(forwardID != myID) // forward toward a nearer node
08:      send_unicast_packet(forwardID,packet);
09:  else peremeter_mode == true; // perimeter mode (omitted)
```

(b) Greedy Forwarding

Fig. 4. Example Implementation of GPSR

As seen in this very simple example, we need to define the language specification to describe various scenarios design the architecture of controller, agents, and network to execute the given scenario in distributed environment. More interesting challenge is to execute the scenario in fully-distributed sensor networks where no base station is present. In such a case, sensor nodes need to collaborate to check the condition and execute the action, with less traffic and computation costs. To determine policies of distributed execution, which optimize message exchanges or some other objective functions, is part of our ongoing work.

4 Protocol Implementation Examples

In this section, we show example implementation of four existing WSN protocols; GPSR, SPEED, BIP and Rumor Routing by using the D-sense design APIs.

GPSR[1] is a position-based protocol, where each sensor node forwards a packet to the neighbor node nearest to the destination by using a planar graph. Figure 4(a) denotes an example implementation of the algorithm to make a Relative Neighborhood Graph (RNG), which is a kind of well-known planar graph. We can see that the algorithm is implemented simply by using APIs, such as listing neighbor nodes ("get_neighbor" in line 02) and getting the distance between nodes ("get_distance" in line 07). Figure 4(b) shows an example implementation of routing process of GPSR. In this code, each node lists its neighbor nodes (line 01) and forwards a packet to the node nearest to the destination in the listed nodes (lines 05–09).

SPEED[2] is also a position-based routing protocol. In SPEED protocol, Stateless Non-deterministic Geographic Forwarding (SNGF) algorithm is used

```
01: SNGF(message){
02:     my_length = get_distance(message->target_ID, myID);
03:     len = get_neighbors(neighborIDs, sizeof(neighborIDs));
04:     num_FS_first = 0; num_FS_Second = 0;
05:     for (i = 0; i < len; i++){
06:         nodeID = neighbor_IDs[i];
07:         diff = my_length
              - get_distance(message->target_ID, nodeID);
08:         if(diff > 0){
09:             if(diff / get_delay(myID,nodeID) > set_point)
10:                 FS_first[num_FS_first++] = nodeID;
11:             else
12:                 FS_second[num_FS_Second++] = nodeID;
13:         }
14:     }
15:     if(num_FS_first > 0){
16:         forwarding_probability = 0;
17:         forwarding_nodeID = FS_first[0];
18:         for(i = 0;i < num_FS_first; i ++){
19:             nodeID = FS_first[i];
20:             fp = get_probability
                  (get_distance(myNodeID,nodeID)
                   ,get_queue_size(nodeID));
21:             if(fp > forwarding_probability){
22:                 forwarding_probability = fp;
23:                 forwarding_nodeID = nodeID;
24:             }
25:         send_unicast_packet(nodeID, message);
26:     }
27:     else // forward toward the nodes in FS_second
28: }
```

Fig. 5. Example Implementation of SPEED

to select a node which is nearer to the destination and handles lighter traffic to forward packets. Figure 5 shows an example implementation of SNGF. A node receiving a packet finds nodes that are nearer to the destination than itself (lines 02–07) and classifies them into two groups. If the transmission efficiency of a node is larger than a threshold *set_point*, it is put into group *FS_first*. The other nodes are put into group *FS_second* (lines 08–13). Then a node is selected from the group *FS_first* of nodes having better transmission efficiency according to the length of the transmission path and the levels of congestion (lines 15–26).

We can see that GPSR and SPEED, both of which are position-based routing protocols, can be implemented by using similar APIs.

BIP [3] is a centralized protocol managing sensor nodes in tree topology, and is designed to minimize the total energy consumption of the network. Due to space limitations, we omit the explanation. The interested readers may refer to our web [17].

Rumor Routing [4] is a routing protocol based on mesh topology, and is designed for accumulation and search of data. Figure 6 shows an example implementation. In Rumor Routing, an agent manages event tables kept in sensor nodes as follows. A node receiving an agent adds information written in the agent to its event table. The information consists of the number of hops to the event *num_hops*, and the direction and the hop count from the node that sends the agent *source_ID* to each event (lines 01–02). At the same time, the node puts information recorded in its event table to the agent and sends it to another node. In this process, a node where agents have not visited long time is selected

```
01:   on_agent_received(source_ID,agent_packet){
02:       event_table = set_event_distance(
                  agent_packet->num_hops,source_ID);
03:       agent_packet = set_event_info(
                  agent_packet, event_table);
04:       if(agent_packet->ttl-- > 0)
05:           send_unicast_packet(
                get_not_visited_neighbor(agent_packet),
                agent_packet);
06:   }
07:
08:   on_query_received(source_ID,query_packet){
09:       query_packet->ttl--;
10:       if(get_num_hops(event_table,query_packet->data)==0)
11:           doQuery(query_packet)
12:       else if (get_hops(query_packet->data) > 0)
13:           send_unicast_packet(query_packet,
            get_forwarding_direction(event_table,query_packet))
14:       else
15:           send_unicast_packet(
                get_not_visited_neighbor(query_packet),
                queryPacket)
16:   }
```

Fig. 6. Example Implementation of Rumor Routing

(lines 03–06). When a node receives a query packet, the node searches the path to the event queried by the packet using its event table (line 10). If the node has the target event information itself, it processes the query, otherwise the node searches a direction to forward it to (lines 11–13). If the node has no information regarding the query at all, the query is forwarded to a node where the query packet has not visited recently (lines 14–15).

At last, in order to show the effectiveness of the D-sense design APIs, we counted the LOC (lines of code) of SPEED codes implemented (1) by using the design APIs, (2) in C++ for QualNet simulator and (3) in NesC for MOTE terminals.

	(1) Design API	(2) C++	(3) NesC
LOC	221	1044	1147

Without using APIs, the implementation required more than 1000 lines. On the other hand, by using the APIs, the LOC is decreased to about 200 lines. Thus, much effort dedicated to implementation was reduced.

5 Performance Evaluation

In order to validate the D-sense implementation and show its usefulness, we evaluated the performance of the SPEED protocol in simulation and real environment by using D-sense, and compared the performance in the simulation to the performance reported in [2].

We used the same scenario as [2]. This scenario is aimed at testing the congestion avoidance capability of the SPEED protocol. A few nodes are randomly selected from the left side of the terrain and send periodic data to the base

station at the right side of the terrain. Each sender generates one CBR flow with 1 packet/second. To create congestion, two randomly chosen nodes in the middle of the terrain create a flow between them at half time of the 150 second experiment. In order to evaluate the congestion avoidance capability under different congestion levels, the rate of this flow is increased by 10 from 0 up to 100 packets/second over several simulations. We have evaluated the delay and loss ratio of the packets to the base station.

We show the experimental environment in Table 2. Because of the limitation on the number of MOTEs, we evaluated the SPEED protocol with 25 nodes in real environment. To compare the reported performance with the real environmental performance, simulation experiments were also conducted in the same configurations. We adjusted the wireless ranges of MOTEs and simulator according to the network scale. Figure 7 shows a snapshot from the experiments in real environment where MOTE terminals were uniformly arranged.

Table 2. Experimental Environment

	Reported	Simulation	Real Env.
PHY & MAC	802.11	802.11	802.15.4
Bandwidth	200 Kb/s	200Kb/s, 250Kb/s	250Kb/s
Payload Size	32 Bytes	32 Bytes	32 Bytes
Terrain	(200m, 200m)	(200m, 200m), (20m, 20m)	(20m, 20m)
# of Nodes	100	100, 25	25
Node Placement	Uniform	Uniform	Uniform
Radio Range	40m	40m, 8m	8m

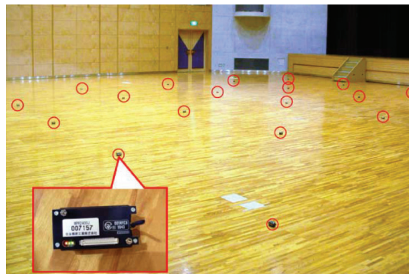
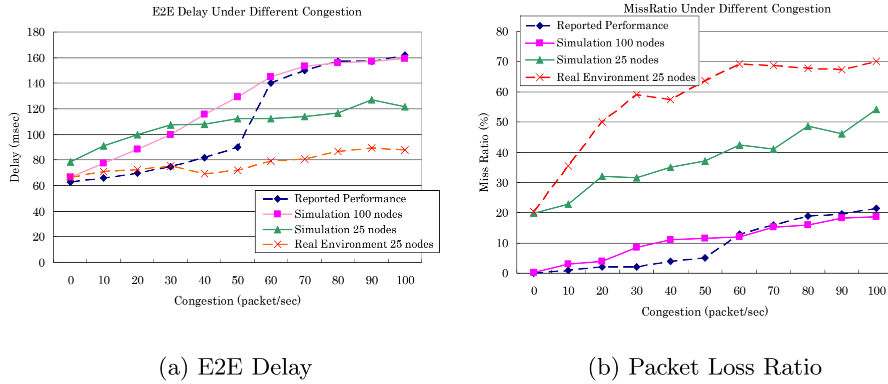


Fig. 7. Arrangement of MOTEs in Real Environment

Figure 8(a) shows the end to end delay. In the experiments with 100 nodes, the performance observed in the simulation well follows the reported performance although small difference is seen around 40 packet/sec congestion. We observed the same level delays in the experiments with 25 nodes as observed in those with 100 nodes. In each congestion level, delays in real environment were smaller than those in simulation.

Figure 8(b) shows packet loss ratio (the ratio of packets that failed to reach the base station). In the experiments with 100 nodes, the simulation performance is nearly equal to the reported performance. In the experiments with 25 nodes, the packet loss ratio is greatly higher than that in the experiments with 100 nodes. This is mainly because each node had too few nodes in its neighbor table to avoid the congestion area at the center of the network in the experiments with 25 nodes. In particular the packet loss ratio is much higher in real environment than that in the simulation.

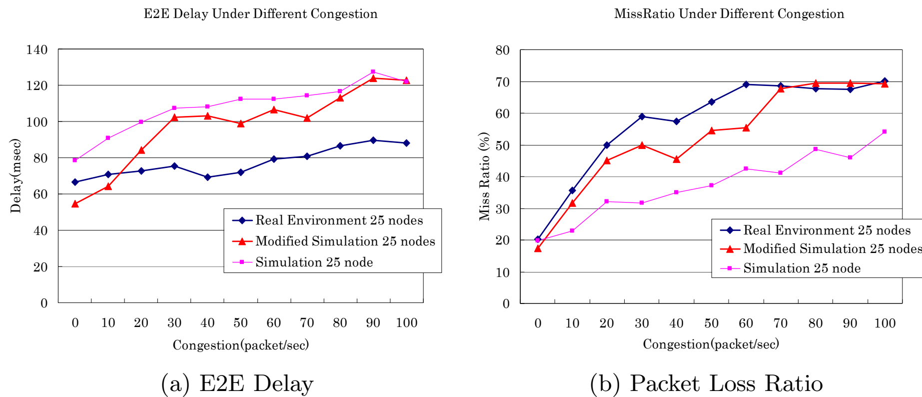
As shown in Figure 8, compared to the simulation results, we can see small delays and large packet loss ratio in real environmental results. We attribute these differences to large fluctuation of radio ranges in real environment. In



(a) E2E Delay (b) Packet Loss Ratio

Fig. 8. Performance of SPEED Protocol

real environment, nodes can receive beacons from further nodes and store the node IDs in its neighbor table. Then, nodes send packets to those further nodes, which have both lower delays and higher probability of packet loss. To solve this problem, we should improve the scheme of neighbor table management. Nodes which receive beacons do not add the IDs of the sender nodes to their neighbor tables until they observe higher success ratio of beacon reception from those nodes than a certain threshold.



(a) E2E Delay (b) Packet Loss Ratio

Fig. 9. Result of modified simulation

On the other hand, as shown in Figure 9, the simulation result gets closer to the real environment one by enlarging the radio range and choosing a proper radio model and height of antenna according to the logs obtained from the experiments in real environment. This shows that we can realize realistic simulation experiments that are closer to the experiments in real environment.

From these performance evaluations, we could validate the D-sense implementation. In addition, we could find some real environmental problems and

their causes, discuss their solutions, and improve reality of the simulation by considering them. This shows the importance of implementing and evaluating WSN protocols in real environment, and also shows that D-sense well supports these activities.

6 Conclusion

In this paper, we have designed and developed an integrated environment called D-sense for supporting development of WSNs. D-sense supports protocol design by high-level design APIs. Also it provides seamless collaboration of simulated and real networks for performance evaluation, and powerful distributed debugging scheme. We have conducted performance evaluation of the SPEED protocol in simulation and real environment to show the effectiveness of D-sense. For now, we have designed the specification of D-sense and implemented a part of its functions. Our ongoing work includes developing a complete set of design/debug APIs and related tools, and opening them to public domain.

References

1. Karp, B. and Kung, H. T.: GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In: 6th Annual International Conference on Mobile Computing and Networking (MobiCom 2000), pp. 149–160 (2000)
2. He, T., Stankovic, J. A., Lu, C. and Abdelzaher, T.: SPEED: A Stateless Protocol for Real-time Communication in Sensor Networks. In: 23rd International Conference on Distributed Computing Systems (ICDCS2003), pp. 46–55 (2003)
3. Wieselthier, J. E., Nguyen, G. D. and Ephremides, A.: On the Construction of Energy-efficient Broadcast and Multicast Trees in Wireless Networks. In: 19th Annual Conference on Computer Communications (INFOCOM2000), pp. 585–594 (2000)
4. Braginsky, D. and Estrin, D.: Rumor Routing Algorithm for Sensor Networks. In: ACM International Workshop on Wireless Sensor Networks and Applications (WSNA2002), pp. 22–31 (2002)
5. Perkins, C. and Royer, E.: Ad-hoc On-demand Distance Vector Routing. In: 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA1999), pp. 90–100 (1999)
6. Wei, W. and Zakhor, A.: Multiple Tree Video Multicast over Wireless Ad Hoc Networks. IEEE Transactions on Circuits and Systems, vol. 17, no. 1, pp. 2–15 (2007)
7. Heinzelman, W. R., Chandrakasan, A. and Balakrishnan, H.: Energy-efficient Communication Protocol for Wireless Microsensor Networks. In: 33rd Annual Hawaii International Conference on System Sciences (HICSS-33), pp. 1–10 (2000)
8. Liyang, Y., Neng, W., Wei, Z. and Chunlei, Z.: GROUP: A Grid-Clustering Routing Protocol for Wireless Sensor Networks. In: 2nd International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM2006), pp. 1–5 (2006)
9. Werner-Allen, G., Swieskowski, P. and Welsh, M.: MoteLab: a Wireless Sensor Network Testbed. In: 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005), pp. 483–488 (2005)

10. CitySense Project: CitySense - An Open, Urban-Scale Sensor Network Testbed. http://www.citysense.net/
11. Madden, S., Franklin, M., Hellerstein, J. and Hong, W.: TinyDB: An Acquisitional Query Processing System for Sensor Networks. ACM Transactions on Database Systems (TODS), vol. 30, no. 1, pp. 122–173 (2005).
12. Bonnet, P., Gehrke, J. and Seshadri, P.: Towards Sensor Database Systems. In: 2nd International Conference on Mobile Data Management (MDM2001), pp. 3–14 (2001)
13. Levis, P. and Culler, D.: Mate: a Tiny Virtual Machine for Sensor Networks. In: 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X2002), pp. 85–95 (2002)
14. Luo, L., Abdelzaher, T. F., He, T. and Stankovic, J. A.: EnviroSuite: An Environmentally Immersive Programming Framework for Sensor Networks. Trans. on Embedded Computing Sys., vol. 5, no. 3, pp. 543–576 (2006)
15. Al-Karaki, J. N. and Kamal, A. E.: Routing Techniques in Wireless Sensor Networks: a Survey," IEEE Transactions on Wireless Communications, vol. 11, no. 6, pp. 6–28 (2004).
16. Ratnasamy, S., Karp, B., Yin L., Yu, F. and Estrin, D: R. Govindan and S. Shenker, GHT: A Geographic Hash Table for Data-centric Storage in Sensornets. In: First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA 2002), pp. 78–87 (2005)
17. D-sense Web, D-sense: An Integrated Environment for Algorithm Design and Protocol Implementation in Wireless Sensor Networks, APIs. http://www-higashi.ist.osaka-u.ac.jp/software/WSN/D-sense/.
18. Scalable Network Technologies, Inc., "Qualnet Simulator," http://www.scalable-networks.com/.