The Mote Connectivity Protocol

Young-ri Choi, Mohamed G. Gouda, and Moon C. Kim
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-0233, U.S.A.
Email: {yrchoi, gouda, mckim}@cs.utexas.edu

Anish Arora
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277, U.S.A.
Email: anish@cis.ohio-state.edu

***Abstract*— An attractive architecture for sensor networks is to have the sensing devices mounted on small computers, called motes. Motes are battery-powered, and can communicate in a wireless fashion by broadcasting messages over radio frequency. In mote networks, the connectivity of a mote *u* can be defined by those motes that can receive messages from *u* with high probability and those motes from which *u* can receive messages with high probability. In this paper, we describe a protocol that can be triggered by any mote in a mote network in order that each mote in the network computes its connectivity. The protocol is simple and has several energy saving features. We implemented this protocol over TinyOS and discuss the results of some execution runs of this implementation.**

I. Introduction

A sensor network consists of a large number of sensing devices capable of detecting waves of sound, heat, magnetism, … in their immediate regions [9], [10], [8], [11]. Each sensing device needs to collaborate with its neighboring devices to perform a given task by disseminating data through the network. An attractive architecture for sensor networks is to have each sensing device mounted on a small computer, called a mote [4]. Motes are battery-powered, and can communicate in a wireless fashion by broadcasting messages over radio frequency [1]. They are energy constrained and the range of their radio transmission is limited to their neighborhood. Thus, they form a multi-hop network to deliver information.

One of the challenging problems in designing mote networks is for any mote *u* to identify every mote that can receive messages from *u* with high probability, and identify every mote from which *u* can receive messages with high probability. Note that because the lists of these motes change over time, mote *u* needs to recompute these lists every so often.

In this paper, we give a formal definition of mote connectivity and devise a protocol that can be triggered by any mote in a mote network in order that each mote in the network computes its connectivity. In our protocol, each mote periodically broadcasts hello messages. Also each mote keeps track of the number of hello messages the mote receives from every other mote during a certain duration, and computes its connectivity based on the number of received hello messages.

In the past, several protocols have been proposed to compute the connectivity of each node in a network, whether the network is wired or wireless. The most prominent examples of these protocols are the Ping protocol in the Internet, the Hello protocol in the OSPF standard, and the channel history protocol in wireless networks. Next, we briefly describe each of these protocols and highlight the differences between each protocol and the mote connectivity protocol.

The Ping protocol [6] allows a node *u* to check if it can exchange messages with another arbitrary node *v* in the Internet. In this protocol, node *u* sends several echo-request messages to node *v*, and *v* replies by sending back an echo-reply message for each echo-request message it receives from *u*. If *u* receives the echo-reply messages from *v*, *u*'s conclusions are sharp: *v* can receive messages from *u and u* can receive messages from *v*. On the other hand, if *u* does not receive the echo-reply messages from *v*, *u*'s conclusions are vague: *v* cannot receive messages from *u or u* cannot receive messages from *v*. Thus, this protocol cannot be used to identify asymmetric patterns of communications between *u* and *v*, where *v* can receive messages from *u* but *u* cannot receive messages from *v*, or vice versa. Because these asymmetric communication patterns are common in mote networks, the Ping protocol cannot be used to measure connectivity in mote networks. (Note that the asymmetric communication patterns are not common in the Internet; this should explain the effectiveness of the Ping protocol in the Internet.)

The Hello protocol in the OSPF standard [5] requires that each node (in this case a router) periodically sends hello messages to every other node in its subnetwork. The periodic exchange of hello messages between two neighboring nodes allows each of the two nodes to identify the pattern of possible communication between the two nodes, even if this pattern is asymmetric where only one of the two nodes can receive messages from the other node. Our mote connectivity protocol is similar, but not identical, to this hello protocol. Our protocol differs from the hello protocol in two important ways:

i) The hello protocol can detect whether or not a node *u* can receive messages from a neighboring node *v*. However, it cannot measure the percentage of messages that *u* can receive from *v*. By contrast, our mote connectivity protocol accurately measures the percentage of messages that *u* can receive from *v*.

ii) The hello protocol is designed to be executing all the time since the nodes (i.e. routers in the Internet) do not need to save their energy.

> By contrast, the mote connectivity protocol is executed for a short period of time and only when one or more motes in the mote network need to update their connectivity information. Thus, our connectivity protocol permits the motes to save their energy.

These differences make our mote connectivity protocol more complex, and also more interesting, than the hello protocol.

In the channel history protocol [2], a master node in a wireless network (usually the base station in the network, if one exits) collects information concerning the communication errors in all channels in the network and uses the collected information to determine whether the states of scheduled channels are good or bad. Because mote sensor networks have a distributed architecture, rather than a centralized one, this protocol may not be useful in such networks.

II. Connectivity in Mote Networks

Let u and v be two motes in a mote network. The *in-connectivity* of u with respect to v is an integer k in the range 0..100 such that k percent of the messages that v sends are received correctly by u. Similarly, the *out-connectivity* of u with respect to v is an integer k in the range 0..100 such that k percent of the messages that u sends are received correctly by v.

The next relation follows from these definitions of in-connectivity and out-connectivity.

The in-connectivity of u w.r.t. v =
The out-connectivity of v w.r.t. u

We refer to this relation as the *mote connectivity relation*.

The (in- and out-) connectivity of a mote with respect to any other mote in its network may change over time, especially if the two motes are moving and the distance between them and their relative orientation change over time. The mote connectivity protocol, described in this paper, is executed by every mote in a mote network when any mote needs to compute its current in-connectivity and out-connectivity with respect to every other mote in the network. Note that the (in- and out-) connectivity of a mote is computed periodically (rather than merely once) so that any change in the mote connectivity can be reflected in future computations of the connectivity.

Let u and v be two motes in a network and assume that the in-connectivity of u with respect to v is zero while the in-connectivity of v with respect to u is larger than zero. According to the mote connectivity protocol, discussed below, each of the two motes accurately computes its own in-connectivity and attempts to report it to the other mote. (This is because the in-connectivity of each mote equals the out-connectivity of the other mote, by the mote connectivity relation.) However, only mote u succeeds in reporting its in-connectivity, or the out-connectivity of v, to mote v (since the in-connectivity of v is larger than zero). Mote v, on the other hand, does not succeed in reporting its in-connectivity, or the out-connectivity of u, to mote u (since the in-connectivity of u is zero). In this case, mote u cannot determine its out-connectivity and so it assumes that its out-connectivity is zero.

In summary, if the in-connectivity of a mote u with respect to another mote v is zero, then mote u cannot determine its out-connectivity with respect to v and arbitrarily assumes it is zero. In this case, the mote connectivity relation may not hold between u and v. (Fortunately, this is the only case where the mote connectivity relation may not hold between two motes.)

Let u and v be two motes in a network. Mote v is an *in-neighbor* of u iff the in-connectivity of u with respect to v is relatively high, 50% or higher. Mote v is an *out-neighbor* of u iff the out-connectivity of u with respect to v is relatively high, 50% or higher. Mote v is a *bi-neighbor* of u iff v is both an in-neighbor and out-neighbor of u.

If the in-connectivity of u with respect to v is 100%, then u concludes easily that u can hear all messages from v. On the other hand, if the in-connectivity of u with respect to v is 0%, then u concludes easily that u cannot hear any message from v at all. Thus, the in-neighbors of u can be defined as the motes from which u can receive more than 50% of messages. Note that a mote can send each message twice if it needs to ensure that the message is received by all its out-neighbors.

In the next section, we describe the mote connectivity protocol that can be triggered by any mote in a mote network so that every mote in the network can compute its in-neighbors, out-neighbors, and bi-neighbors.

III. Specification of the Connectivity Protocol

A mote u that participates in the mote connectivity protocol partitions its time into a sequence of slots of t seconds each. At the beginning of each slot, mote u executes its timeout action as follows. First, u broadcasts a hello message. Second, u schedules its timeout action to be executed next after t seconds. (A small random jitter can be added to the value of t to reduce the probability that two or more nearby motes broadcast their hello messages at the same time causing the messages to collide and not be received by any mote in the network.)

Because each mote in the network broadcasts a hello message at the beginning of each slot, a mote u can compute its in-connectivity with respect to another mote v by keeping track of the number of hello messages that u received from v during its last 10 slots. Also, each mote includes its in-connectivity (computed based on the number of hello messages that the mote has received during the last 10 slots) in each hello message it broadcasts. Thus, when a mote u receives a hello message from a mote v and finds in the message the in-connectivity of v with respect to u, u stores this information as its own out-connectivity with respect to v.

Each mote u stores its in-connectivity in an array named inc, and stores its out-connectivity in an array named outc. These two arrays are declared in u as follows,

```
var inc, outc : array [0..n-1] of 0..100
```

In this declaration, n is the total number of motes in the network. Each element inc[v], where $v \neq u$, in the first array stores the in-connectivity of *u* with respect to *v*. Similarly, each element outc[v], where $v \neq u$, in the second array stores the out-connectivity of *u* with respect to *v*.

Each mote *u* also has an array named slot to keep track of the number of hello messages that *u* received from every other mote during the last 10 slots that preceded the current slot named x.

```
var slot : array [0..n-1, 0..10] of integer,
       x : 0..10
```

Thus, the element slot[v,x] stores the number of hello messages that *u* receives from *v* during the current slot. In general, the element slot[v, x-k mod 11] stores the number of hello messages that *u* received from *v* during the k-th slot that preceded the current slot.

When the value of element slot[v,x] is zero, the sum of all *v*-elements, slot[v,0] + ... + slot[v,10], is the number of hello messages that mote *u* received from mote *v* during the last 10 slots. In this case, this sum can be used to compute the in-connectivity inc[v] of *u* with respect to *v* as follows.

```
inc[v] :=
      10 * min(slot[v,0]+ .. + slot[v,10], 10)
```

Note that in some cases, the sum slot[v,0] +...+ slot[v,10] is larger than 10, and in those cases, the sum is reduced to 10 (by the function min) before it is multiplied by 10 to produce the in-connectivity inc[v] of *u* with respect to *v*.

When a mote *u* participates in the mote connectivity protocol, mote *u* executes at most $D + 10$ slots of this protocol, where D is an upper bound on the network diameter. Thus, each mote *u* maintains a variable named ns to store the number of remaining slots that *u* needs to execute in its current execution round of the mote connectivity protocol. Variable ns is declared as follows in mote *u*.

```
var ns : 0..D+9
```

Note that when ns becomes 0, mote *u* terminates its current execution round of the mote connectivity protocol. At this instant, the two arrays inc and outc in mote *u* have "correct" values and mote *u* records the current time in an integer variable named recordt. We assume that for the next T seconds after a mote *u* terminates its execution round of the mote connectivity protocol, the two arrays inc and outc in mote *u* have correct values and *u* has no reason to initiate another execution round of the protocol. Later on, however, when the current time exceeds the value of recordt by more than T seconds, the values in the two arrays inc and outc in mote *u* become old (and possibly incorrect), and *u* may need to initiate the next execution round of the protocol.

Each hello message that is broadcasted by a mote *u* has three fields as follows

```
hello(u,s,c)
```

The first field is the name *u* of the mote that broadcasts the message. The second field is the current value of variable ns in mote *u*. The third field is the current value of array inc in mote *u*.

When a mote *u* receives a hello(v,s,c) message, *u* checks whether the value of its ns variable is 0. In one hand, if the value of its ns is larger than 0, then *u* recognizes that it is participating in the current execution round of the mote connectivity protocol. In this case, *u* uses the received message to update its two arrays outc and slot. In the other hand, if the value of ns equals 0, then *u* recognizes that it is not participating in the current execution round of the protocol. In this case, depending on the value of s in the received message, *u* decides whether to join the current execution round of the protocol. If s < 10, *u* decides that there is not enough remaining slots in the current execution round to justify joining (the current execution round of the protocol). In this case, *u* discards the received hello(v,s,c) message. If s ≥ 10, *u* joins the current execution round of the protocol.

Below, we give a formal specification of the program of a mote *u* in the mote connectivity protocol. This specification is written using a notation similar to the Abstract Protocol notation described in [3]. A program is specified in this notation as a *set* of actions. Each *action* is of the form

```
<condition> -> <statement>
```

The `<statement>` of an action is executed only when the `<condition>` is true. Each `<statement>` of an action is a sequence of

```
assignment statements,
broadcast statements,
if ... fi statements, and
for ... do .. od statements
```

The program of a mote *u* consists of three actions. In the first action, *u* detects that it can initiate the next execution round of the protocol and it does so. In the second action, *u* times-out and executes the next slot in the current execution round of the protocol. In the third action, *u* receives a hello(v,s,c) message and processes it.

The program of mote *u* is specified next. Note that this program specification contains three statements named `INITIALIZE, SCHEDULE`, and `RECORD`. The definitions of these three statements are given below the program specification.

```
mote u:0..n-1

var inc, outc : array [0..n-1] of 0..100,
    slot      : array [0..n-1, 0..10] of integer,
    x         : 0..10,
    ns        : 0..D+9,
    recordt   : integer,
    t         : integer,
    v         : 0..n-1,
    s         : 0..D+9,
    c         : array [0..n-1] of 0..100

begin
    ns = 0 and recordt+T < "currenttime" ->
       ns := D+9;
       INITIALIZE;
       broadcast hello(u,ns,inc);
       SCHEDULE

[]  timeout is activated ->
       x := x+1 mod 11;
       for every v, v in 0..n-1 and v != u do
           slot[v,x] := 0;
           inc[v] := 10 *
           min(slot[v,0]+ .. + slot[v,10], 10);
           if inc[v]=0 -> outc[v] := 0
           [] inc[v]>0 -> skip
           fi
       od;
       ns := ns-1;
       broadcast hello(u,ns,inc);
       if ns>0 -> SCHEDULE
       [] ns=0 -> recordt := "currenttime"
       fi

[]  rcv hello(v,s,c) ->
       if ns>0            -> RECORD
       [] ns=0 and s<10   -> skip
       [] ns=0 and s>=10 ->
           ns := s-1;
           INITIALIZE;
           RECORD;
           inc[v] := 10;
           broadcast hello(u,ns,inc);
           SCHEDULE
       fi
end
```

The three statements INITIALIZE, SCHEDULE, and RECORD that appeared in this specification are defined as follows.

```
INITIALIZE
       x := 0;
       inc := 0;
       outc := 0;
       slot := 0

SCHEDULE
       activate timeout in t seconds

RECORD
       outc[v] := c[u];
       slot[v,x] := slot[v,x]+1
```

Using this protocol, mote *u* can decide at time r that another mote *v* is its neighbor as follows.

i) *v* is an in-neighbor of *u* at r iff
in *u*, recordt+$T \geq$ r and inc[v] ≥ 50

ii) *v* is an out-neighbor of *u* at r iff
in *u*, recordt+$T \geq$ r and outc[v] ≥ 50

iii) *v* is a bi-neighbor of *u* at r iff
v is an in-neighbor and out-neighbor of *u* at r

In Section 4, we discuss some energy saving features of this protocol. In Section 5, we discuss how we use an implementation of this protocol to investigate the connectivity properties of a class of motes, called Mica motes [7].

Ad-hoc On Demand Distance Vector Routing (AODV) protocol [12] uses hello messages to maintain the local connectivity of nodes in a network. The AODV protocol is similar to our protocol, but our protocol is different in the following ways:

i) The AODV protocol computes only the bi-neighbors of each mote. By contrast, the mote connectivity protocol computes the in-neighbors, out-neighbors and bi-neighbors of each mote.

ii) Similar to the hello protocol in the OSPF standard [5], the AODV protocol is designed to be executing all the time. By contrast, the mote connectivity protocol executes only on-demand (to save energy).

IV. Energy saving features

The above mote connectivity protocol has four features intended to reduce the energy consumption of motes. These four features are as follows.

- On-demand execution
- Short execution time
- Small number of messages
- Small message size

Next, we discuss how each of these features contributes to save the mote's energy.

a) On-demand execution: A mote does not start to execute the mote connectivity protocol until the mote needs its current neighbor information and until the connectivity values stored in the mote are no longer valid. If no mote needs neighbor information for a long period of time, no mote needs to send hello messages during that period. Once a mote starts the mote connectivity protocol, every mote executes a fixed number of slots and then terminates the protocol. After the protocol is terminated, the mote considers the computed connectivity values valid for the next T seconds.

b) Short execution time: A mote executes at least 10 slots and at most $(D+10)$ slots in one run of the mote connectivity protocol, where D is an upper bound on the network diameter. If the duration of each slot is t seconds, the execution time of one run of the mote connectivity protocol is between $10 * t$ seconds and $(D + 10) * t$ seconds. If a mote sends hello messages every 5 seconds in the mote network, and if the network diameter is 50, each mote executes one run of the mote connectivity protocol in between 50 seconds and 300 seconds. Thus, the average execution time of a mote is 175 seconds.

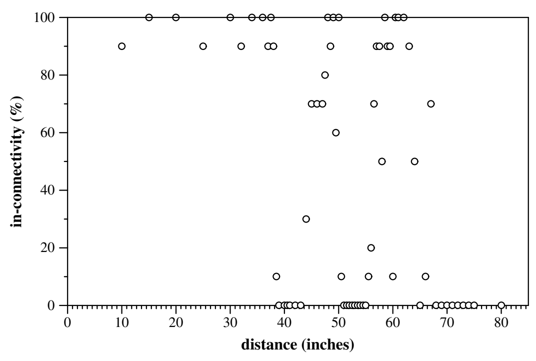

Fig. 1. In-connectivity of mote0 with respect to mote1.

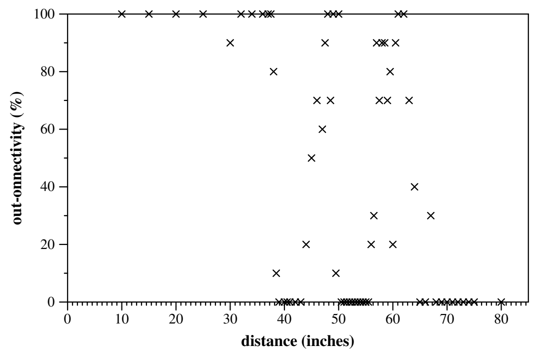

Fig. 2. Out-connectivity of mote0 with respect to mote1.

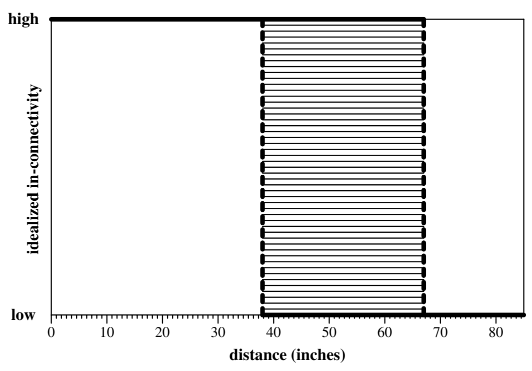

Fig. 3. Idealized in-connectivity of mote0 with respect to mote1.

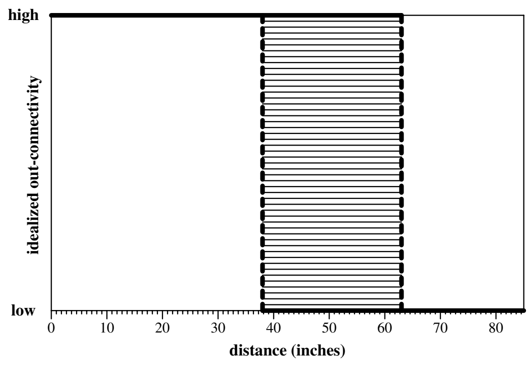

Fig. 4. Idealized out-connectivity of mote0 with respect to mote1.

c) Small number of messages: A mote sends at least 10 hello messages and at most $(D+10)$ hello messages in one run of the mote connectivity protocol, where D is an upper bound on the network diameter. If the diameter of a network is 50, a mote in the network sends at most 60 hello messages and at least 10 hello messages in one run of the protocol. Thus, the average number of hello messages for a mote to send is 35.

d) Small message size: In the specification of Section 3, the third field of each hello message is the current value of array inc whose size is proportional to the total number of motes in the network. We can reduce the size of each hello message, specially when a network is large, such that a mote only includes the in-connectivity of motes whose value is bigger than zero. Thus, the size of the third field becomes proportional to the number of the motes whose in-connectivity is bigger than zero.

V. Connectivity Measurements

We have implemented the above mote connectivity protocol over TinyOS and embedded our implementation on Mica motes [7]. We used the implemented protocol, with t around 5 seconds, to measure the connectivity of two motes, named mote 0 and mote 1, over distances that range from 10 inches to 80 inches. The two motes were supplied with relatively new batteries to ensure that the difference in batteries does not cause any asymmetric communication patterns. The experiments were carried out in a lab without any obstacles between the two motes. The relative orientation between the two motes remained the same during the experiments.

Fig. 1 and 2 respectively show the in-connectivity and out-connectivity of mote 0 with respect to mote 1. Each mark in Fig. 1 represents a value of the in-connectivity of mote 0 after executing 50 slots. Similarly, each mark in Fig. 2 represents a value of the out-connectivity of mote 0 after executing 50 slots.

Referring to Fig. 1, one observes that the in-connectivity (of mote 0 with respect to mote 1) partitions the range of possible distances (between mote 0 and mote 1) into three intervals:

i) In the first interval, the distance is less than 38 inches. In this interval, the in-connectivity is consistently high between 90% and 100%.

ii) In the second interval, the distance is between 38 inches and 67 inches. In this interval, the in-connectivity is unstable in the following sense. A slight change in the distance within this interval leads to a large change in the in-connectivity. For example, if the distance is changed from 60 inches to 60.5 inches, the in-connectivity changes from 10% to 100%.

iii) In the third interval, the distance is larger than 67 inches. In this interval, the in-connectivity is consistently low between 0% and 10%.

From these observations, the relationship between the distance and in-connectivity, depicted in Fig. 1, can be "idealized" as shown in Fig. 3. Note that in Fig. 3, if the distance is less than 38 inches, then the in-connectivity is "high". Moreover, if the distance is between 38 inches and 67 inches, then the in-connectivity can have any value. Finally, if the distance is larger than 67 inches, then the in-connectivity is "low".

Similarly, the relationship between the distance and out-connectivity, depicted in Fig. 2, can be "idealized" as shown in Fig. 4. Note that in Fig. 4, if the distance is less than 38 inches, then the out-connectivity is "high". Moreover, if the distance is between 38 inches and 63 inches, then the out-connectivity can have any value. Finally, if the distance is larger than 63 inches, then the out-connectivity is "low".

Figures 3 and 4 show that the idealized in-connectivity and out-connectivity are consistent, i.e. both are high or both are low, provided that the distance is outside the interval of instability between 38 inches and 67 inches.

VI. Concluding Remarks

In this paper, we gave a formal definition of connectivity in mote networks: let u and v be two motes in a mote network. The *in-connectivity* of u with respect to v is an integer k in the range 0..100 such that k percent of the messages that v sends are received correctly by u. Similarly, the *out-connectivity* of u with respect to v is an integer k in the range 0..100 such that k percent of the messages that u sends are received correctly by v.

We presented a protocol that can be triggered by any mote in a mote network in order that each mote in the mote network computes its connectivity. Our protocol can identify asymmetric communication patterns (i.e., a mote u can receive messages from another mote v but v cannot receive from u) as well as symmetric communication patterns. Our protocol has several energy saving features: on-demand execution, short execution time, small number of messages and small message size. Finally, we implemented our protocol over TinyOS and measured the connectivity of two motes over a variety of distances.

A mote sends hello messages every t seconds during the execution of the mote connectivity protocol. It is possible that a mote piggybacks hello messages on any application messages sent by the mote at this time. In this way, a mote can reduce the number of messages to send and save energy.

In the mote connectivity protocol, a mote considers the computed connectivity values valid for the next T seconds, after the mote terminates the protocol. The range of T is between 0 and infinity. If T equals 0, then a mote needs to execute the protocol whenever the mote needs neighbor information. If T equals infinity, then a mote needs to execute the protocol only once during its lifetime. The value of T can be selected based on the type of the mote network. On one hand, if the network is stationary, then motes can execute the protocol less frequently (i.e., by choosing T large). On the other hand, if the network is mobile, then the motes need to execute the protocol frequently (i.e., by choosing T small).

The neighbor information provided by the mote connectivity protocol can be utilized on many applications in mote networks. Several applications that can utilize the protocol were discussed in [13].

References

[1] D. E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A network-centric approach to embedded software for tiny devices. In *EMSOFT 2001: First International Workshop on Embedded Software*, October 2001.

[2] S. Deb, M. Kapoor, and A. Sarkar. Error avoidance in wireless networks using link state history. In *INFOCOM*, pages 786–795, 2001.

[3] M. G. Gouda. *Elements of Network Protocol Design*. John Wiley and Sons, Inc., New York, New York, 1998.

[4] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278, 1999.

[5] J. Moy. The OSPF specification RFC 1131, 1989.

[6] J. Postel. Internet control message protocol RFC 792, 1981.

[7] TinyOS. http://today.cs.berkeley.edu/tos/.

[8] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. In *Proceedings of the 7th annual international conference on Mobile Computing and Networking* (MobiCom 2001), ACM Press, New York, pages 189–199, 2001.

[9] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, Elsevier Science, Vol. 38, No. 4, pages 393–422, March 2002.

[10] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, Vol. 40, No. 8, pages 102–114, August 2002.

[11] L. Zhou and Z. J. Hass. Securing Ad Hoc Networks. *IEEE Network*, Vol. 13, no. 6, pages 24–30, 1999.

[12] Charles E. Perkins and Elizabeth M. Royer. Ad hoc On-Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[13] Young-ri Choi, Mohamed G. Gouda, Moon C. Kim and Anish Arora. The Mote Connectivity Protocol. Technical Report TR03-08, Department of Computer Sciences, the University of Texas at Austin, 2003