Security of the Grid Routing Protocol In Sensor Networks

Lydia Miller

Honors Thesis

CS379H

Department of Computer Sciences
The University of Texas at Austin

2003-2004

First Reader
Professor Mohamed Gouda

Second Reader
Professor Simon Lam

The Grid Routing Protocol

Lydia Miller

Advised by Professor Mohamed Gouda

Acknowledgements

I would like to thank Dr. Mohamed Gouda for his time and attention to this research project. The Grid Routing Protocol was developed by Dr. Mohamed Gouda, his graduate student Young-Ri Choi, Dr. Anish Arora at Ohio State University and his graduate student Vinayak Naik. I used the slides of some presentation of this group as a starting point for this thesis. I then added many security features to this protocol. I am also thankful to Dr. Simon Lam for his comments which improved the presentation a great deal.

Abstract

A sensor network consists of a large number of tiny computers; each capable of sensing magnetism, movement, heat, or sound in their vicinities. The computers in this network collaborate to achieve a global sensing objective. The characteristics of each computer usually include constrained battery power, small memory resources, and limited computational power. Moreover, the failure rate of each tiny computer is high. The computers communicate wirelessly through radio signals, and because each computer has a limited communication radius, the sensor network is in fact a multi-hop system. Sensor networks may be used in various applications including monitoring and surveillance, object tracking and inventory keeping, military awareness, natural disaster detection in remote places, monitoring sensitive areas, and so on. Because of the nature of these applications, it is essential for the message transmission in these networks to be secure from an adversary who might attempt to compromise the network effectiveness.

In this thesis, we formally specify a simple routing protocol, called the grid routing protocol, that can be used for routing data messages in a sensor network. We also specify several enhancements of this protocol to make it secure against mote impersonation and infiltration attacks.

Table of Contents

1 The Grid Routing Protocol

The Logical Grid Routing Protocol is a protocol for routing data messages in sensor networks. [2], [4] This protocol overcomes random message loss and mote failure.

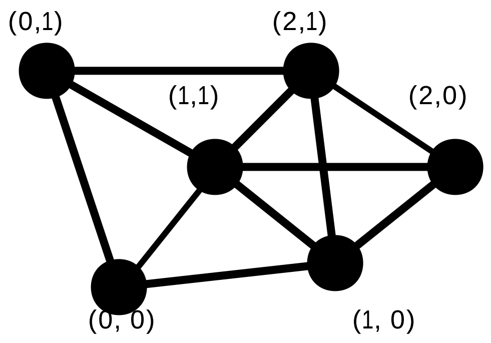
1.1 The Grid

We consider a network that consists of a number of tiny computers called motes. The characteristics of each mote usually include constrained battery power, small memory resources, and limited computational power. Moreover, the failure rate of each mote is high. These motes communicate wirelessly through radio signals, and because each mote has a limited communication radius, this network is in fact a multi-hop system.

The motes in the network are placed in an arbitrary physical configuration, but they are labeled as if they are points in an M*N logical grid. Therefore, each mote has a label (i, j) where i := 0 .. M – 1 and j := 0 .. N – 1.

It is required that the physical connectivity of the motes is a superset of the grid connectivity implied by the mote labels. As an example,

Figure 1



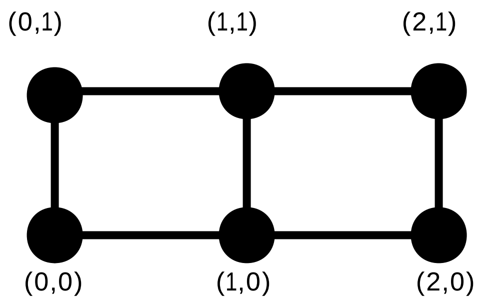


Figure 2

Figure 1 shows a network with six motes; the edges between the motes describe which motes can communicate with which other motes. Thus, Figure 1 describes the physical connectivity in the network. Figure 2 describes the logical connectivity in the same network based on the chosen labels of the motes. Note that the edges in Figure 1 (i.e. the physical connectivity of the network) are a superset of the edges in Figure 2 (i.e. the grid connectivity of the same network).

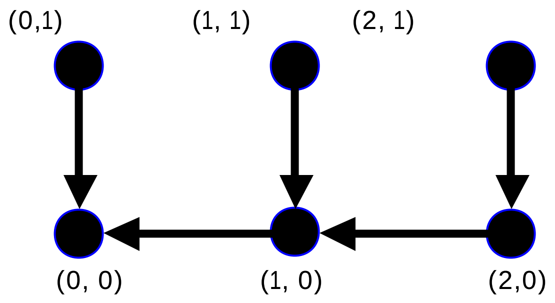
1.2 Logical Neighbors

Each mote [i, j] in the middle of the grid has four logical neighbors. These logical neighbors are mote [i, j + 1], mote (i, j – 1), mote [i + 1, j] and mote [i - 1, j]. Two of those four motes are called low neighbors, and the other two are called high neighbors. The two low neighbors of mote [i, j] are motes [i, j – 1] and [i – 1, j], and the two high neighbors are motes [i + 1, j] and [i, j + 1]. If mote [i, j] is located on the boundaries of the grid, then it has a smaller number of logical neighbors. There are four cases to consider. In the first case, where i = 0, mote [0, j] has only one low neighbor, namely mote [0, j – 1]. In the second case, where j = 0, mote [i, 0] has only one low neighbor, namely mote [i – 1, 0]. (Therefore, mote [0, 0] has no low neighbors.) In the third case, where i = M – 1, mote [M – 1, j] has only one high neighbor, namely mote [M – 1, j + 1]. In the fourth case, where j = N – 1, mote [i, N – 1] has only one high neighbor, namely mote [i + 1, N – 1]. (Therefore, mote [M – 1, N – 1] has no high neighbors.)

1.3 Communication Pattern

Each mote [i, j] in the network broadcasts messages whose ultimate destination is mote [0, 0]. To facilitate communication between motes, the motes need to maintain an incoming spanning tree whose root is mote [0. 0]. As an example, Figure 3 shows a spanning tree for the grid network in Figure 2. Each mote in the network maintains a pointer to its parent in the tree, and these pointers form the spanning tree. When a mote receives a message, it forwards the message to its parent in the tree. This process continues, until the message reaches the tree root, namely mote [0, 0].

Figure 3

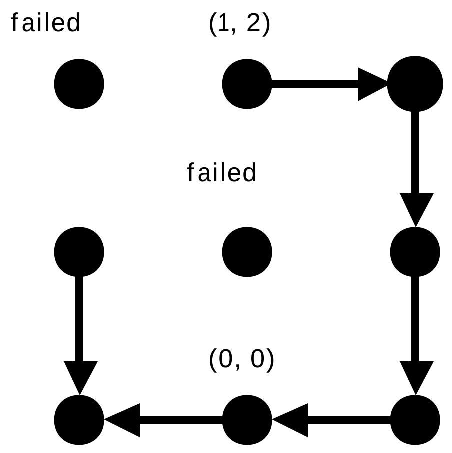


The parent of a mote in the spanning tree is chosen from the logical neighbors of this mote so that the parent of the mote is within the communication range of the mote. When a mote [i, j] broadcasts a message, the message reaches to all motes within the communication range of [i, j]. All motes within the communication range, excluding the parent, discard the message, and the parent forwards the message to its parent, and so on, until the message reaches mote [0. 0]. Clearly, if all logical neighbors of a mote fail, then this mote cannot have a parent in the spanning tree and so cannot broadcast any messages to mote [0. 0].

1.4 Choosing the Parent

Each mote [i, j] chooses its parent from its logical neighbors based on the distance of each logical neighbor. The logical distance of a mote [i, j] is defined to be i + j. It is advantageous for a mote [i, j] to choose its parent with as small a distance as possible. Therefore, when a mote chooses its parent from its logical neighbors, the mote prefers a low neighbor. However, if both low neighbors of a mote fail, then the mote chooses one of its high neighbors to be its parent. This situation is called an inversion. Thus, an inversion is defined to be an occurrence where a mote chooses one of its high neighbors to be its parent. As an example, one inversion occurs at mote (1, 2) in the network in Figure 4, because both low neighbors of mote (1, 2) have failed.

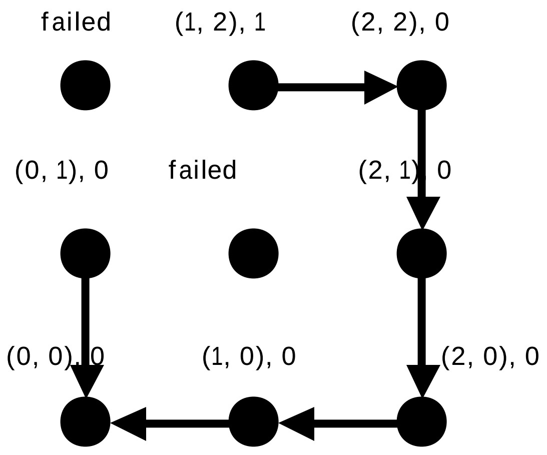
Figure 4



1.5 Inversion Count

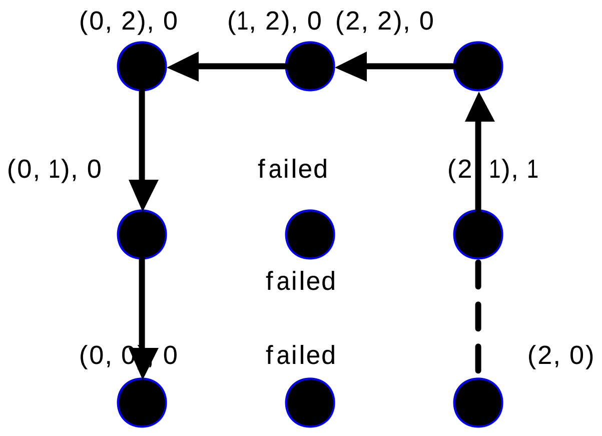
The inversion count of a mote [i, j] is defined to be the number of inversions that occur along the spanning tree route from mote [i, j] to mote [0. 0]. As an example, Figure 5 shows the inversion counts of each mote in the network in Figure 4: the inversion count of mote [1, 2] is 1, whereas the inversion count of every other mote is 0.

Figure 5



The inversion count of a mote has an upper bound, namely M + N, and a lower bound, namely 0. We can limit the upper bound of the inversion count artificially to be a smaller number. One advantage of having an artificial upper bound on the inversion count is that a message has a shorter distance to the root. A disadvantage of having this artificial upper bound is that a mote could potentially be isolated from the network and therefore unable to forward messages to mote [0. 0]. As an example, in Figure 6, if cmax equals 1, then mote (2, 0) is disconnected from the network and cannot forward any messages.

Figure 6



1.6 Protocol Message

When, and only when, a mote [i, j] has a parent (x, y), every t seconds mote [i, j] broadcasts a connected message to all motes within its communication range. This connected message has three fields: (i, j, c), where (i, j) is the label of the mote broadcasting the message and c represents its current inversion count. All motes within the communication range of mote [i, j], excluding the neighbors of mote [i, j], discard the connected (i, j, c) message broadcasted by mote [i, j]. Mote [0. 0] broadcasts a connected (0, 0, 0) message every t seconds.

If all logical neighbors of mote [i, j] fail, then mote [i, j] has no parent, and it will not broadcast any messages. In this case, mote [i, j] becomes isolated from the spanning tree and can no longer forward any message to mote [0. 0].

1.7 Acquiring a Parent

Initially, mote [i, j] has no parent. When mote [i, j] has no parent and receives a connected (x, y, d) message from a neighbor (x, y), mote [i, j] chooses this neighbor as its parent in two cases. The first case occurs when mote [x, y] is a low neighbor. The second case occurs when mote [x, y] is a high neighbor and the inversion count d is less than cmax.

When mote [i, j] receives a connected (x, y, d) message where (x, y) is a low neighbor of mote [i, j], mote [i, j] chooses this neighbor to be its parent and computes its inversion count using c := d. If the parent [x, y] is a high neighbor and d is less than cmax, mote [i, j] computes its inversion count using c := d + 1.

Consider the network in Figure 6. Mote [0. 0] broadcasts the message connected (0, 0, 0) every t seconds. When mote [0, 1] receives this message, it recognizes that [0. 0] is its low neighbor, and so it chooses (0. 0) to be its parent in the spanning tree and makes its own inversion count equal 0. From this point on, mote [0, 1] broadcasts the message connected (0, 1, 0) every t seconds. When mote [0, 2] receives the message connected (0, 1, 0) and recognizes that mote [0, 1] is its low neighbor, it chooses mote [0, 1] to be its parent and its inversion count to be 0, and the cycle repeats. When finally mote [2, 1] receives the message connected (2, 2, 0) and recognizes that mote [2, 2] is its high neighbor, it chooses mote [2, 2] to be its parent and its inversion count to be 1. From this point on, mote [2, 1] starts to broadcast the message connected (2, 1, 1) every t seconds. When mote [2, 0] receives the message connected (2, 1, 1), it will not choose mote [2, 1] to be its parent because that would make its own inversion count 2, violating the upper bound of 1 on the inversion count. Thus, mote [2, 0] remains without a parent, cannot join the spanning tree, and cannot forward any message toward mote [0. 0].

1.8 Keeping the Parent

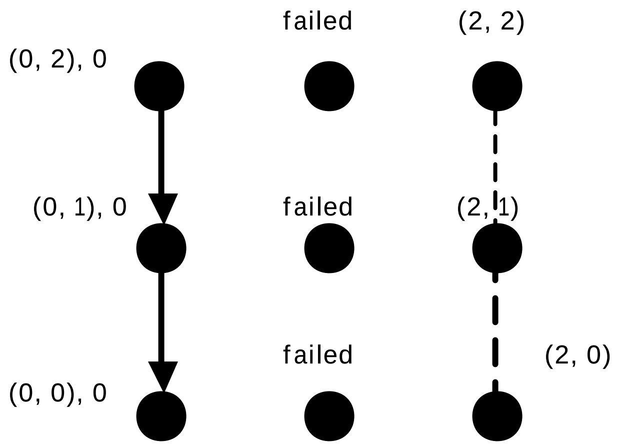
Once a mote [i, j] has a parent (x, y) and receives any connected (x, y, d) message, mote [i, j] uses the received e to update its own inversion count c as follows. The inversion count c is calculated as c := d if the parent (x, y) is a low neighbor of [i, j], and it is calculated as c := d + 1 if the parent (x, y) is a high neighbor of [i, j] and d is less than cmax. When the parent (x, y) is a high neighbor and e equals cmax, then mote [i, j] loses its parent.

1.9 Losing the Parent

There are two cases where a mote [i, j] loses its parent (x, y). In the first case, mote [i, j] receives a connected (x, y, cmax) message from its parent (x, y) and the parent is a high neighbor of [i, j]. In the second case, mote [i, j] does not receive a connected (x, y, d) message from its parent (x, y) for 4*t seconds.

Consider the network in Figure 6. Based on this network, mote [2, 2] has its parent to be mote [1, 2]. In Figure 7, mote [2, 2] loses its parent, after receiving no message connected (1, 2, 0) for 4*t seconds. From this point on, mote [2, 2] is disconnected from the network, so it is unable to broadcast connected (2, 2, 0) messages. Since mote [2, 1] does not receive any connected (2, 2, 0) message from its parent for 4*t seconds, it also loses its parent and is disconnected from the network as shown in Figure 7.

Figure 7



1.10 Replacing the Parent

There is one case where mote [i, j] replaces its parent (x, y) with a new parent (u, v). This case occurs only when mote [i, j] has a parent (x, y) and receives a connected (u, v, f) message, where (u, v) is the label of a neighbor and f is the inversion count of this neighbor. Mote [i, j] chooses (u, v) as its parent, if using f to compute the inversion count c of mote [i, j] reduces the value of c. In this case, mote [i, j] chooses its parent to be (u, v) and computes its inversion count using f. Mote [i, j] always chooses the parent with the smallest inversion count in order to minimize the length of the route from mote [i, j] to mote [0. 0] in the incoming spanning tree.

Consider the network in Figure 6 and assume that mote [1, 1] wakes up. When mote [1, 1] wakes up, receives a message connected (0, 1, 0), and recognizes that mote [0, 1] is its low neighbor, mote [1, 1] chooses (0, 1) to be its parent and makes its inversion count 0. From this point on, mote [1, 1] broadcasts a connected (1, 1, 0) message every t seconds. When mote [2, 1] receives the connected

(1, 1, 0) message and recognizes that mote [1, 1] is its low neighbor, mote [2, 1] replaces its parent (2, 2) with mote [1, 1} and reduces its inversion count from 1 to 0. From this point on, mote [2, 1] broadcasts a connected (2, 1, 0) message every t seconds. When mote [2, 0] receives a message connected (2, 1, 0) and recognizes that mote [2, 1] is its high neighbor, mote [2, 0] chooses mote [2, 1] to be its parent and sets its inversion count to 1. The resulting spanning tree is shown in Figure 8.

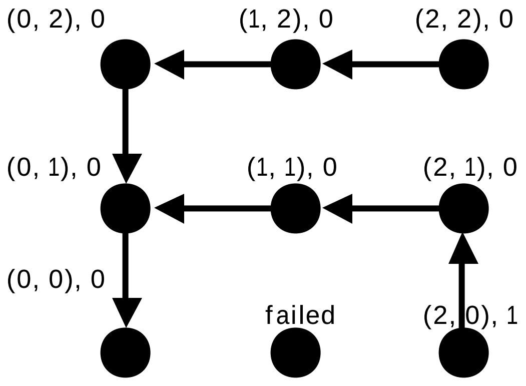


Figure 8

2 Formal Definition of the Grid Routing Protocol

It is beneficial to specify the above grid routing protocol using an abstract notation, called the SP (Sensor Protocol) notation [3] to explain the protocol precisely. The English language, or any natural language for that matter, is too imprecise to describe such a protocol. Precision provides a clearer understanding of the protocol. Using the SP notation, we model each mote in the network as a process. Each process has constants, inputs, variables, and actions.

2.1 Constants, Inputs and Variables

Each process mote [i, j] has one constant, cmax, that represents the maximum possible value for the inversion count of mote [i, j]. The constant cmax is declared as follows:

```
const        cmax        :        integer
```

Each process mote [i, j] has two set inputs. The first is called 'L', and it contains the indices of the low neighbors of mote [i, j]. The second set is called ‘H’, and it contains the high neighbors of mote [i, j]. These two sets are declared as follows:

```
inp     L    :    set {(i, max  (j – 1, 0)), (max  (i – 1, 0), j),
        H    :    set {(i, min  (j + 1, n – 1)), (min  (i + 1, m – 1), j) }
```

Each process mote [i, j] has five variables. The first variable, named pid, stores the identity of the parent of mote [i, j]. Recall that the value of pid is to be taken from the set L U H. The second variable, named trc, is discussed in the next paragraph. The third variable, named c, stores the inversion count of mote [i, j]. The fourth variable, named d, stores the inversion count of the parent of mote [i, j]. The fifth variable, named tout, is a timeout variable. The timeout variable and the concept of timeout is explained below in sections 3.2 and 3.3.

The second variable trc represents the remaining time period that mote [i, j] has a parent and is connected to the spanning tree. Each time mote [i, j] receives a connected message from its parent, trc is set to 4. Each t seconds trc is decremented by 1. Thus, if mote [i, j] does not receive a connected message from its parent for a period of 4t seconds, trc becomes zero, and mote [i, j] is no longer connected to the spanning tree. In section 2.9, Losing a Parent, recall that one way a mote loses its parent is to receive no connected message from its parent for 4t seconds.

The five variables of each process mote [i, j] are declared as follows:

```
    var     pid    :    L U H
            trc    :    0 .. 4
            c, d   :    0 .. cmax
            tout   :    boolean
```

In the initial state of the network, mote [0. 0] has the following values for its variables:

pid = (0, 0)
trc = 4

c = 0
tout = **true**

Every other mote [i, j] has the following values for its variables:
trc = 0
c = 0
tout = **false**

2.2 Timeout Variables

In the initial state of the protocol, variable trc has the value 4 in mote [0, 0] and has the value 0 in every other mote. If a mote [i, j] receives a connected message from a neighbor and chooses that neighbor to be its parent, the value of trc in mote [i, j} becomes 4, and variable tout is set to true. While tout in mote [i, j] is true, mote [i, j] times out and broadcasts a connected message every t seconds, the timeout action decrements the value of trc by 1 and activates another timeout after t seconds, until the value of trc becomes 0. Once the trc variable becomes zero, the mote loses its parent, is no longer connected to the spanning tree, and finally tout becomes false and no timeout is activated. Recall that a mote loses its parent if the mote receives no connected message for 4t seconds.

As an example, consider a network where a mote [i, j] receives a connected message (x, y, d) and chooses mote [x, y] to be its parent. The pid of mote [i, j] becomes (x, y), the trc variable becomes 4, and tout becomes true. Mote [i, j] waits to receive another connected message. After t seconds, the timeout action is executed, and trc is decremented by 1. If mote [i, j] waits t more seconds and receives no connected message, once again the timeout action is executed, and trc is decremented by 1. Finally, when trc becomes zero, mote [i, j] loses its parent and is disconnected from the spanning tree. The tout variable is declared as follows:

tout : **boolean**

In the initial state of the network, for mote [0, 0], tout = **true.** Otherwise, tout = **false**.

2.3 Timeout Actions

The timeout action in mote [i, j] can be viewed as consisting of two parts. In the first part, mote [i, j] checks whether it is mote [0, 0]. If not, mote [i, j] sets its trc variable to the maximum of trc – 1 and 0. If so, mote [0, 0] does nothing. This first action of the timeout is defined as follows:

```
timeout →if i ‡ 0 ^ j ‡ 0 → trc = max (trc – 1, 0)
            [] i = 0 ^ j = 0 → skip
            fi;
```

In the second part of the timeout, each mote [i, j] broadcasts a connected (i, j, c) message and activates the timeout to expire next after t seconds. If trc equals 0, the value of tout becomes false. The second part of the timeout action is defined as follows:

```
if trc > 0 → broadcast conn (i, j, c);
             activate timeout in t seconds
[] trc = 0 → tout := false
fi
```

The entire timeout action is as follows:

```
timeout →if i ‡ 0 ^ j ‡ 0 → trc = max (trc – 1, 0)
            [] i = 0 ^ j = 0 → skip
            fi;
            if trc > 0 → broadcast conn (i, j, c);
                         activate timeout in t seconds
            [] trc = 0 → tout := false
            fi
```

2.4 Receive Action

The receive action of each mote [i, j] has one if … fi statement that consists of branches. This action is specified as follows:

```
[] rcv conn (x, y, d) →

        if ~((x, y) in L U H) → skip            {discard conn}

        [] (x, y) in L ^ (trc = 0) →            < statement S0 >
        [] (x, y) in L ^ (trc > 0) →                < statement S1 >

        [] (x, y) in H ^ (trc = 0) →                < statement S2 >

        [] (x, y) in H ^ (trc > 0) →                < statement S3 >

        fi
```

Note that statement S_0 is executed if the received conn (x, y, d) was broadcast by a low neighbor of mote [i, j], and mote [i, j] currently has no parent. Statement S_1 is executed if the received conn (x, y, d) was broadcast by a low neighbor of mote [i, j], and mote [i, j] currently has a parent. Statement S_2 is executed if the received conn (x, y, d) is broadcast by a high neighbor of mote [i, j], and mote [i, j] currently has no parent. Statement S_3 is executed if the received conn (x, y, d) was broadcast by a high neighbor of mote [i, j], and mote [i, j] currently has a parent.

The four statements S_0 through S_3 can be specified as follows:

```
Statement S0 → pid, trc, c := (x, y), 4, d;
               **if** ~tout → activate timeout in t seconds; tout := **true**
               [] tout → **skip**
               **fi**

Statement S1 → **if** pid = (x, y) → trc, c := 4, d
               [] pid ‡ (x, y) ^ ( d < c) → pid, trc, c := (x, y), 4, d
               [] pid ‡ (x, y) ^ (d ≥ c) → **skip** {discard message}
               **fi**

Statement S2 → **if** d < cmax → pid, trc, c := (x, y), 4, d + 1;
                     **if** ~tout → activate **timeout** in t seconds; tout := **true**
                     [] tout → **skip**
                     **fi**
               [] d = cmax → **skip**            {discard message}
               **fi**

Statement S3 → **if** pid = (x, y) ^ (d < cmax) → trc, c := 4, d + 1
               [] pid = (x, y) ^ (d = cmax) → trc := 0
               [] pid ‡ (x, y) ^ (d + 1 < c) → pid, trc, c := (x, y), 4, d + 1
               [] pid ‡ (x, y) ^ (d + 1 ≥ c) → **skip**      {discard conn}
               **fi**
```

2.5 Formal Specification

The Grid Routing Protocol can be specified formally as follows:

```
process mote [i: 0..m – 1, j: 0 .. n – 1]

const           cmax    :       integer

inp             L       :       set {(i, max  (j – 1, 0)), (max  (i – 1, 0), j),
                H       :       set {(i, min  (j + 1, n – 1)), (min  (i + 1, m – 1), j) }
                x, y    :       integer

var             pid     :       L U H
                trc     :       0 .. 4
                c, d    :       0 .. cmax
                tout    :       boolean

begin
timeout →if  i ‡ 0 ^ j ‡ 0 → trc = max (trc – 1, 0)
           [] i = 0 ^ j = 0 → skip
           fi;
          if trc > 0 → broadcast conn (i, j, c);
                       activate timeout in t seconds
          [] trc = 0 → tout := false
           fi
      [] rcv conn (x, y, d) →
              if ~((x, y) in L U H) → skip             {discard conn}
              [] (x, y) in L ^ (trc = 0) → pid, trc, c := (x, y), 4, d;
                                      if ~tout → activate timeout in t seconds; tout := true
                                      [] tout → skip
                                      fi
              [] (x, y) in L ^ (trc > 0) →  if pid = (x, y) → trc, c := 4, d
                                            []  pid ‡ (x, y) ^ ( d < c) → pid, trc, c := (x, y), 4, d
                                            []  pid ‡ (x, y) ^ (d ≥ c) → skip       {discard message}   p16
                                            fi
              [] (x, y) in H ^ (trc = 0) → if d < cmax → pid, trc, c := (x, y), 4, d + 1;
                                                 if ~tout → activate timeout in t seconds; tout := true
                                                 [] tout → skip
                                                 fi
                                           [] d = cmax → skip            {discard message}
                                           fi
              [] (x, y) in H ^ (trc > 0) → if  pid = (x, y) ^ (d < cmax) → trc, c := 4, d + 1
                                           [] pid = (x, y) ^ (d = cmax) → trc := 0
                                           [] pid  ‡ (x, y) ^ (d + 1 < c) → pid, trc, c := (x, y), 4, d + 1
                                           [] pid ‡ (x, y) ^ (d + 1 ≥ c) → skip
```

```
                              fi
                    fi
end
```

initial state: if i = 0 ^ j = 0 then pid = (0, 0), trc = 4, c = 0, tout = **true** and timeout is activated
            else trc = 0, tout = **false**, and the timeout is not activated

3 Using the Grid Routing Protocol

The function of the Grid Routing Protocol, discussed in the last section, is for each mote [i, j] to maintain two variables trc and pid. When trc > 0 then mote [i, j] is connected to the spanning tree via its parent mote [pid]. In this case, mote [i, j] can forward any data message to its parent until the data message reaches the network root, namely mote [0, 0]. When trc = 0, then mote [i, j] can forward any data message to its parent which in turn forwards it to its parent until the data message reaches the network root, mote [0, 0]. When trc = 0, then mote [i, j] is not connected to the spanning tree and can no longer forward data messages toward the network root.

The data transfer protocol, that uses the grid routing protocol, can be specified as follows:

```
process mote [i: 0 .. m – 1, j: 0 .. n – 1]

inp     L           :       set {(i, max  (j – 1, 0)), (max  (i – 1, 0), j),
        H           :       set {(i, min  (j + 1, n – 1)) (min  (i + 1, m – 1), j) }
        pid         :       L U H           {from grid routing protocol}
        trc         :       0 .. 4          {from grid routing protocol}

var     t           :       integer
        x, y        :       integer

begin
        trc > 0 → t := any; broadcast data (pid, t)
        rcv data  (x, y, t)  →  if  (x ‡ i) v (y ‡ j) v (trc = 0) ^ (x ‡ 0 v y ‡ 0) → skip      {discard data}
                                []  (x = i) ^ (y = j) ^ (trc > 0) ^ (x ‡ 0 v y ‡ 0) → broadcast data (pid, t)
                                []  (x = i) ^ (y = j) ^ (trc > 0) ^ x = 0 ^ y = 0 → skip        {store data}
                                fi
end
```

4 Adversarial Actions

In order to secure the grid routing protocol, it is necessary to consider what types of scenarios the adversary could use to disrupt the protocol. One of these scenarios is mote impersonation. Mote impersonation occurs when an adversary removes one of the legitimate motes out of the network and replaces it with an illegitimate adversarial mote. Another scenario is network infiltration. This scenario occurs when an adversary adds an adversarial mote to the network. We describe these scenarios in more detail next.

4.1. Impersonating motes in a network without shared secrets

One action that an adversary may take to disrupt the Grid Routing Protocol is to remove a legitimate mote from the network and replace it with an illegitimate foreign mo te. This action is called impersonating a mote. As an example, consider a network where an adversary removes mote [0, 1] from the network. A foreign mote takes the place of mote [0, 1]. The foreign mote then broadcasts connected (0, 1, 0) messages every t seconds. In this network, assume motes [0, 2] and [1, 1] choose the foreign mote to be their parent, when they receive a connected (0, 1, 0) message from the adversary that is impersonating mote [0, 1] and recognize that mote [0, 1] is their low neighbor. Mote [0, 2] then broadcasts a connected (0, 2, 0) message every t seconds, and mote [1, 1] broadcasts a connected (1, 1, 0) message every t seconds. Mote [0, 3] chooses mote [0, 2] to be its parent; motes [1, 2] and [2, 1] choose mote [1, 1] to be their parent and so on.

Each parent is the root of a subtree. In this example, the root of a subtree is mote [0, 1], and the subtree includes all of the motes that have chosen mote [0, 1] to be their parent, namely motes [0, 2] and [1, 1], all of the motes that have chosen mote [0, 2] or [1, 1] to be their parent, and all of the motes that have chosen motes [0, 3], [1, 2], or [2, 1] to be their parent and so on.

All the motes in this subtree forward their messages to their parent, until they get to the foreign mote. One action that the adversary may take to disrupt the network is to have the foreign mote discard all of the messages it receives. This action is extremely disruptive, since all of the messages in this subtree instead of being forwarded to the root are discarded. Clearly the result of this action is message loss. Another action the adversary can take is to receive a connected message, alter it, and then broadcast the altered message. This action results in message corruption.

4.2 Impersonating Motes in a Network with Shared Secrets

To counter the adversarial attack called mote impersonation, it is necessary to enhance the grid routing protocol by adding shared secrets. Each mote shares a separate secret with each of its neighbors. Thus, a mote has the same number of shared secrets as it has neighbors. Each mote computes a value using each of its shared secrets, one value per secret. These values are appended to the connected message before broadcasting it. When a mote receives a connected message from one of its logical neighbors, the mote uses the shared secret to authenticate the connected message.

In this more secure network, the adversary may once again remove a mote from the network and replace it with a foreign mote. The legitimate mote broadcasts a connected message every t seconds, but in this network the shared secret provides an authentication value verifiable by the receiver. For example, in this more secure network, mote [0, 1] broadcasts a connected (0, 1, 0) message and appends a message digest value computed using the shared secret. A neighbor of the mote, mote [0, 2], receives the message, recognizes that the message contains the computed value using its shared secret, verifies this value, and accepts the message. Mote [0, 2] recognizes that mote [0, 1] is a low neighbor and chooses mote [0, 1] to be its parent in the spanning tree. This process continues throughout the network; a mote chooses mote [0, 2] to be its parent and so on. In this network, before the foreign mote removes mote [0, 1] from the network, the foreign mote accepts one of the connected (0, 1, 0) messages from mote [0, 1] with the appended value computed using the shared secret. The foreign mote then removes mote [0, 1] from the network and replays this connected (0, 1, 0) message with the computed values appended for each neighbor using their respective secret keys. Thus, when mote [0, 2] receives this message, it chooses the foreign mote to be its parent, since the value mote [0, 2] computed using the shared secret is the same as the received value. Just as in the example of mote impersonation in a network without shared secrets, mote [1, 1] chooses this foreign mote to be its parent when mote [1, 1] receives the connected (0, 1, 0) message, verifies the appended message digest value, and recognizes that mote [0, 1] is its low neighbor, and the process continues. This second scenario also disrupts the grid routing protocol, since the integrity of the subtree whose root is the foreign mote is compromised by message replay.

4.3 Infiltrating the Network

In this attack, the adversary adds an illegitimate mote to the network which broadcasts wrong data messages containing the identity of a legitimate mote. The legitimate motes continue to broadcast their data messages, but the infiltrator also broadcasts data messages with inaccurate data.

For example, assume that mote [2, 1] is alive and is broadcasting a data (1, 1, t) message. Then an illegitimate mote infiltrates the network and pretends to be this mote. The adversary mote broadcasts a data (1, 1, t_a) message, where t_a is the data fabricated by the adversary. Mote [1, 1] receives the data message and forwards it to its parent, mote [1, 0], who forwards it to mote [0, 0]. Clearly the integrity of the data transfer protocol is compromised.

5 Countering Mote Impersonation

In order to provide integrity to the Logical Grid Routing Protocol, it is necessary to add shared secrets. Each mote [i, j] has one shared secret for each of its neighbors.

5.1 Enhancing the Network

In this more secure protocl, mote [i, j] and its high neighbor mote [i, j + 1] share one secret. This shared secret is called N in mote [i, j]. Mote [i, j] and its other high neighbor, mote [i + 1, j], share another secret. This secret is called E in mote [i, j]. In the same manner, mote [i, j] shares a different secret with mote [i, j – 1]; this secret is called S in mote [i, j] Mote [i, j] shares yet another secret with its other low neighbor, mote [i - 1, j]. This last secret is called W in mote [i, j]. Therefore, mote [i, j] has one shared secret for each of its neighbors. Each time that mote [i, j] broadcasts a connected message, this mote computes a message digest function (MD) over the message text and each shared secret that mote [i, j] shares with a neighbor.

A message digest MD is a function that computes a unique integer value for any data item d (of type integer), a data item MD(d) in the range of 0 .. k – 1 such that the following condition is satisfied:

Finger Printing: There is no efficient algorithm that computes, for any MD(d), a data item d' such that MD(d) = MD(d'). Thus, MD(d) is a finger print of data item d. [1]

Before broadcasting a message, each mote [i, j] computes a message digest of i, j, c, and the shared secret for each neighbor. The value MD(i|j|c|sc) represents the message digest of the concatenation of i, j, c, and the shared secret. Each time that a mote broadcasts a message, the mote computes the message digest for each neighbor, using the shared secret of that neighbor. For example, if a mote [i, j] has four neighbors, assume S represents the secret mote [i, j] shares with mote [i, j – 1], W represents the secret mote [i, j] shares with mote [i - 1, j], N represents the secret mote [i, j] shares with mote [i, j + 1], and E represents the secret mote [i, j] shares with mote [i + 1, j]. The message mote [i, j] broadcasts would look like the following. Let sd = MD(i|j|c|S) (where a vertical line represents concatenation), wd = MD(i|j|c|W), nd = MD(i|j|c|N), and ed = MD(i|j|c|E). The message that mote [i, j] broadcasts will be connected (i, j, c, sd, wd, nd, ed). If mote [i, j - 1] receives the connected message, it will compute a message digest function for x, y, d, and the shared key, where the shared key for mote [i, j – 1] is N. So, mote [i, j - 1] will have a value nd' = MD(x|y|d|N). If nd' = sd, mote [i, j - 1] will accept the connected message; if nd' ‡ sd, mote [i, j - 1] will discard the connected message. If mote [i - 1, j] receives the connected (x, y, d, sd, wd, nd, ed) message from mote [i, j], mote [i - 1, j] uses its shared secret E to compute a message digest function of x, y, d, and the shared secret. Mote [i - 1, j] will have a value ed' = MD(x|y|d|E). If ed' = wd, then mote [i - 1, j] receives the connected message, and if ed' ? wd, then mote [i - 1, j] discards the connected message. If mote [i, j + 1] receives the connected (x, y, d, sd, wd, nd, ed) message, mote [i, j + 1] uses its shared secret S to compute a message digest function of x, y, d, and the shared secret. Mote [i, j + 1] computes the value sd' = MD(x|y|d|S). If sd' = nd, then mote [i, j + 1] receives the message; otherwise mote [i, j + 1] discards the message. Finally, if mote [i + 1, j] receives the connected (i, j, c, sd, wd, nd, ed) message from mote [i, j], it computes a value wd' using the message digest function with x, y, d, and W concatenated. Thus, wd' = MD(x|y|d|W), and mote [i + 1, j] receives the message if wd' = ed and discards it otherwise.

This protocol provides a solution to the first attack, impersonating a mote in a network without shared secrets. However, this protocol provides no solution to the attack where the foreign mote replays one of the messages from the mote that it replaces in the network. Upon receiving a message where the message digest value equals its own computed value, i.e. sd' = nd, wd' = ed, nd' = sd, or ed' = wd, each mote will assume that this connected message is from a legitimate mote. Thus, a foreign mote may disrupt the protocol by replaying one of these connected messages, causing other motes to choose the adversarial mote as their parent and form a subtree, giving the adversary an opportunity to discard or alter messages.

5.2 Formal Definition of the Enhanced Protocol

In order to prevent the adversary from impersonating a mote in the network, the grid routing protocol needs added security. The description of the constants, inputs, variables, and statements added to the grid routing protocol to provide security follows.

When the grid routing protocol is enhanced to include shared secrets, the inputs added represent the secrets mote [i, j] shares with each of its neighbors: S, W, N, and E. These inputs are declared as follows:

S, W, N, E :	**integer**	{S.(i, j) = N.(i, j – 1), N.(i, j) = S.(i, j + 1) , W.(i, j) = E.(i – 1, j), E.(i, j) = W.(i + 1, j)}

Recall that the secret mote [i, j] shares with mote [i, j – 1] is the S secret for mote [i, j] and the N secret for mote [i, j – 1]. Similarly, the secret mote [i, j] shares with mote [i – 1, j] is W for mote [i, j] and E for mote [i – 1, j]. The secret mote [i, j] shares with mote [i, j + 1] is N for mote [i, j], but the same secret for mote [i, j] is S. Finally, the secret that mote [i, j] shares with mote [i + 1, j] is E for mote [i, j] and W for mote [i + 1, j].

The additional variables representing the message digest values are these:

sd, wd, nd, ed	:	**integer**
sd’, wd’, nd’, ed’	:	**integer**

The timeout inputs have the secrets S, W, N, and E, as well as the variables representing the message digest values sd, ed, nd, and wd. The guard for the timeout actions is still the same, and the first statements are also unchanged.

```
if i ‡ 0 v j ‡ 0 → trc = max (trc – 1, 0);
[] i = 0 ^ j = 0 → skip                        {store data}
```

Next, mote [i, j] compares the value of its trc variable to 0. If the value is more than 0, then mote [i, j] calculates its message digest values, broadcasts a message including these values, and activiates its timeout to expire next after t seconds. The statement added is the one where mote [i, j] computes the message digest values.

```
if trc > 0 → sd, wd, nd, ed: = MD(i|j|c|S), MD(i|j|c|W), MD(i|j|c|N), MD(i|j|c|E);
             broadcast conn (i, j, c, sd, wd, nd, ed);
             activate timeout in t seconds
```

The timeout is unchanged, except for the additional computation of the message digest values.

```
[] trc = 0 → tout := false
```

The complete timeout action of the secure grid routing protocol is:

```
timeout → if i ‡ 0 v j ‡ 0 → trc = max (trc – 1, 0);
          [] i = 0 ^ j = 0 → skip
          fi;
          if trc > 0 →
                sd, wd, nd, ed: = MD(i|j|c|S), MD(i|j|c|W), MD(i|j|c|N), MD(i|j|c|E);
                broadcast conn (i, j, c, sd, wd, nd, ed);
                activate timeout in t seconds
          [] trc = 0 → tout := false
          fi
```

There are several local guards added to the original grid routing protocol, in order to provide security using shared secrets. The reason for adding these guards is each time that a connected message is received from a neighbor, mote [i, j] has an extra step to identify by which neighbor the message was broadcast, so that mote [i, j] can determine whether to use S, W, N, or E in its message digest function.

Instead of receiving a connected (x, y, d) message, now mote [i, j] receives a connected (x, y, d, sd, wd, nd, ed) message from its neighbor. Thus, mote [i, j] determines which neighbor the message is from and then computes its own message digest value to compare with the MD value received. For example, if the message is from mote [i, j – 1], then mote [i, j] uses the shared secret S to calculate a value called sd', where sd' = MD(x|y|d|S). The L and H for high neighbors and low neighbors are no longer specific enough to use, since each high neighbor has a different secret, and each low neighbor has a different secret. The added local guards are as follows:

```
[] ((x, y) = (i, j – 1)) ^ (trc = 0) →
[] ((x, y) = (i - 1, j)) ^ (trc = 0) →
[] ((x, y) = (i, j – 1)) ^ (trc > 0) →
[] ((x, y) = (i - 1, j)) ^ (trc > 0) →
[] ((x, y) = (i, j + 1)) ^ (trc = 0) →
[] ((x, y) = (i + 1, j)) ^ (trc = 0) →
[] ((x, y) = (i, j + 1)) ^ (trc > 0) →
[] ((x, y) = (i + 1, j)) ^ (trc > 0) →
```

Each process mote [i, j] has the additional statements needed to compute its message digest value using the secret that mote [i, j] shares with the neighbor broadcasting the message and to compare that value to the one received from the neighbor. This is the additional statement, when mote [i, j] receives a message from its low neighbor, mote [i, j – 1].

```
sd' := MD(i|j|c|S);
if sd' = nd → $S_0$
```

There is a similar statement for each message received by mote [i, j] from motes [i + 1, j], [i, j – 1], and [i – 1, j]. Thus, the enhanced protocol includes the constants, inputs, variables, and actions of the original grid routing protocol along with other enhancements to ensure security with respect to mote impersonation.

5.2.e Formal Definition of Enhanced Protocol

```
process mote [i: 0..m – 1, j: 0 .. n – 1]

const       cmax               :     integer

inp         L                  :     set {(i, max  (j – 1, 0)), (max  (i – 1, 0)}
            H                  :     set {(i, min  (j + 1, n – 1)), (min  (i + 1, m – 1), j)}
            S, W, N, E         :     integer          {S.(i, j) = N.(i, j – 1),
                                                  W.(i, j) = E.(i – 1, j),
                                                  N.(i, j) = S.[i, j + 1] ,
                                                  E.(i, j) = N.(i, j)}
var         pid                :     L U H
            trc                :     0 .. 4
            c, d               :     0 .. cmax
            tout               :     boolean
            sd, wd, nd, ed     :     integer
            sd', wd', nd', ed' :     integer

begin
   timeout → if i ‡ 0 v j ‡ 0 → trc = max (trc – 1, 0);
             [] i = 0 ^ j = 0 → skip
             fi;
             if trc > 0 →
                  sd, wd, nd, ed: =  MD(i|j|c|S), MD(i|j|c|W), MD(i|j|c|N),  MD(i|j|c|E);
                  broadcast conn (i, j, c, sd, wd, nd, ed);
                  activate timeout in t seconds
             [] trc = 0 → tout := false
             fi
 [] rcv conn (x, y, d, sd, wd, nd, ed) →
             if ~((x, y) in L U H) → skip                {discard conn}
             [] ((x, y) = (i, j – 1)) ^ (trc = 0) → sd' := MD(x|y|d|S);
                        if sd' = nd → pid, trc, c := (x, y), 4, d;
                              if ~tout → activate timeout in t seconds; tout := true
                              [] tout → skip
                              fi
                        [] sd' ‡ nd → skip                          {discard conn}
                        fi
```

```
[] ((x, y) = (i - 1, j)) ^ (trc = 0) → wd' := MD(x|y|d|W);
                 if wd' = ed → pid, trc, c := (x, y), 4, d;
                            if ~tout → activate timeout in t seconds; tout := true
                            [] tout → skip
                            fi
                 [] wd' ‡ ed → skip                         {discard conn}
                 fi
 [] ((x, y) = (i, j – 1)) ^ (trc > 0) → sd' := MD(x|y|d|S);
                 if (sd' = nd) ^ (pid = (x, y)) → trc, c := 4, d
                 [] (sd' = nd) ^ (pid ‡ (x, y)) → if d < c → pid, trc, c := (x, y), 4, d
                                                 [] d ≥ c → skip     {discard conn}
                                                 fi
                 [] (sd' ‡ nd)   → skip                      {discard conn }
                 fi
 [] ((x, y) = (i - 1, j)) ^ (trc > 0) → wd' := MD(x|y|c|W);
                 if (wd' = ed) ^ (pid = (x, y)) → trc, c := 4, d
                 [] (wd' = ed) ^ (pid ‡ (x, y)) → if d < c → pid, trc, c := (x, y), 4, d 32
                                                 [] d ≥ c → skip    {discard conn}
                                                 fi
                 [] (wd' ‡ ed) → skip      {discard conn}
                 fi
[] ((x, y) = (i, j + 1)) ^ (trc = 0) → nd' := MD(x|y|c|N);
                 if ( nd' = sd) ^ (d < cmax) → pid, trc, c := (x, y), 4, d + 1;
                                 if ~tout → activate timeout in t seconds; tout := true
                                 [] tout → skip
                                 fi
                 [] ( nd' ‡ sd) v (d = cmax) → skip                 {discard conn}
                 fi
[] ((x, y) = (i + 1, j)) ^ (trc = 0) → ed' := MD(x|y|c|E);
                 if ( ed' = wd) ^ (d < cmax) → pid, trc, c := (x, y), 4, d + 1;
                                 if ~tout → activate timeout in t seconds; tout := true
                                 [] tout → skip
                                 fi
                 [] ( ed' ‡ wd) v (d = cmax) → skip                 {discard conn}
                 fi
 [] ((x, y) = (i, j + 1)) ^ (trc > 0) → nd' := MD(x|y|c|N);
                 if  (nd' = sd) ^ (pid = (x, y)) ^ (d < cmax) → trc, c := 4, d + 1
                 [] (nd' = sd) ^ (pid = (x, y)) ^ (d = cmax) → trc := 0
                 [] (nd' = sd) ^ (pid  ‡ (x, y)) ^ (d + 1 < c) → pid, trc, c := (x, y), 4, d + 1
                 [] (nd' = sd) ^ (pid ‡ (x, y)) ^ (d + 1 ≥ c) → skip
                 []  nd' = sd → skip                                {discard conn}
                 fi
 [] ((x, y) = (i + 1, j)) ^ (trc > 0) → ed' := MD(x|y|c|E);
                 if (ed' = wd) ^ (pid = (x, y)) ^ (d < cmax) → trc, c := 4, d + 1
                 [] (ed' = wd) ^ (pid = (x, y)) ^ (d = cmax) → trc := 0
```

```
                        [] (ed' = wd) ^ (pid ‡ (x, y)) ^ (d + 1 < c) → pid, trc, c := (x, y), 4, d + 1
                        [] (ed' = wd) ^ (pid ‡ (x, y)) ^ (d + 1 ≥ c) → **skip**
                        [] ed' ‡ wd → **skip**                          {discard conn}
                        **fi**
            **fi**
**end**
```

initial state: if i = 0 ^ j = 0 then pid = (0, 0), trc = 4, c = 0, tout = **true** and timeout is activated
else trc = 0, tout = **false**, and the timeout is not activated

6 Countering Network Infiltration

There is a way to secure a sensor network against network infiltration using a message digest function. In the data transfer protocol, a message digest value is added using a shared secret in order to provide security. This way, any message that originates from an adversarial mote can easily be detected. This concept is similar to that of the secure grid routing protocol.

6.1 Enhancing the Protocol

In the data transfer protocol, in order to prevent mote infiltration, a message digest value is added to the data message. Thus, there are these added inputs:

```
        PS, S, W, N, E          :               **integer**
```

PS is the secret that mote [i, j] shares with its parent in the spanning tree.

There are also some added variables. The variable d represents the message digest value computed by mote [i, j], before it broadcasts the data message. There are six additional variables in the enhanced protocol called sd, wd, nd, ed, pd, and d. The variable pd stores the message digest value that mote [i, j} computes before broadcasting data to its parent. Therefore, pd is the message digest of pid concatenated with t concatenated with PS. The variable d stores the message digest value calculated by the broadcasting mote. These variables are declared as follows:

```
    pd, sd, wd, nd, ed, d    :              integer
```

In the first action, the only statement added to the data transfer protocol is the statement where mote [i, j] computes the message digest value using the secret shared with its parent. This variable is called pd. The added statement is this: pd := MD(pid|t|PS), and the pd value is included in the data message. The first action including its statements is defined as follows:

```
    if trc > 0 → t := any; pd := MD(pid|t|PS); broadcast data (pid, t, pd)
```

In the second action, mote [i, j] computes the message digest values sd', wd', nd', and ed' to verify the authenticity of the message it has received. Next, mote [i, j] compares sd', wd', nd', and ed' with d to see whether the values are equal. If not, the message is discarded. Otherwise, if the message digest value is equal to that computed by mote [i, j], and if (i, j) is other than (0, 0), then the message is forwarded to the parent as in the original data transfer protocol. If (i, j) equals (0, 0), then the message is stored; thus mote [i, j] skips. The second action including its statements is as follows:

```
    rcv data (x, y, t, d) →
        if (x ‡ i) V (y ‡ j) V (trc = 0) → skip                {discard data}
        [] (x = i) ^ (y = j) ^ (trc > 0) →
                sd, wd, nd, ed := MD(x|y|t|S), MD(x|y|t|W), MD(x|y|t|N), MD(x|y|t|E);
            if ((sd = d) V (wd = d) V (nd = d) V (ed = d)) ^ (x ‡ 0 v y ‡ 0) →
                    pd := MD(pid|t|PS); broadcast data (pid, t, pd)
            [] ((sd = d) v (wd = d) v (nd = d)  v (ed = d)) ^ x = 0 ^ y = 0 → skip {store data}
            [] (sd ‡ d) ^(wd ‡ d) ^ (nd ‡ d) ^ (ed ‡ d) → skip              {discard data}
            fi
        fi
```

Thus, the secure data transfer protocol has the same number of actions as the original protocol. It uses the same message digest function that is in the secure grid routing protocol.

6.3 Formal Definition of Enhanced Protocol

```
process mote [i: 0 .. m – 1, j: 0 .. n – 1]

inp     L                   :       set {(i, max  (j – 1, 0)), (max  (i – 1, 0), j),
        H                   :       set {(i, min  (j + 1, n – 1)) (min  (i + 1, m – 1), j) }
        pid                 :       L U H
        trc                 :       0 .. 4          {from grid routing protocol}
        PS, S, W, N, E      :       integer         {from grid routing protocol}


var     t                   :       any
        x, y                :       integer
        pd, sd, wd, nd, ed, d:      integer


begin
        trc > 0 → t := any; pd := MD(pid|t|PS); broadcast data (pid, t, pd)
        rcv data (x, y, t, d) →
                if (x ‡ i) v (y ‡ j) v (trc = 0) → skip                      {discard data}
                [] (x = i) ^ (y = j) ^ (trc > 0) →
                                sd, wd, nd, ed := MD(x|y|t|S), MD(x|y|t|W), MD(x|y|t|N), MD(x|y|t|E);
                        if  ((sd = d) v (wd = d) v (nd = d) v (ed = d)) ^ (x ‡ 0 v y ‡ 0) →
                                pd := MD(pid|t|PS); broadcast  data (pid, t, pd)
                        [] ((sd = d) v (wd = d) v (nd = d)  v (ed = d)) ^ x = 0 ^ y = 0 → skip    {store data}
                        [] (sd ‡ d) ^ (wd ‡ d) ^ (nd ‡ d) ^ (ed ‡ d) → skip                  {discard data}
                        fi
                fi
end
```

7 Concluding Remarks

The grid routing protocol provides a straightforward abstraction of the sensor network. In this abstraction, each mote is defined to be a process. Each mote broadcasts messages to its neighbors, chooses one of them to be its parent in the spanning tree, and forwards data toward the root of the spanning tree, via its parent. Due to the sensitive nature of most applications of the sensor network, the protocol requires security. The secure grid routing protocol and the secure data transfer protocol protect the sensor network from mote impersonation. The addition of the shared secrets between neighbors and the message digest function provides a solution to the problem of one adversarial mote replacing a legitimate mote in a sensor network. This shared secret enhancement also secures the network against mote infiltration.

One attack that the adversary may make against the network with shared secrets is that of message replay. One solution to provide security to the network against anti-replay is to add sequence numbers and a dynamic memory window in which to store whether or not a message has been received. Using this protocol, a mote may determine whether a message is new or is being replayed. However, the description of this protocol is beyond the scope of this paper.

References

[1] M.G. Gouda, *Elements of Network Protocol Design,* John Wiley & Sons, New York, New York, 1998

[2] Young-ri Choi, Mohamed Gouda, Moon C. Kim, and Anish Arora "The Mote Connectivity Protocol", in Proceedings of 12th International Conference on Computer Communications and Networks (ICCCN 2003), pages 522 – 538, Dallas, Texas, October 20 – 22, 2003

[3] Young-Ri Choi, Sensor Network Protocols, Ph. D. thesis, in progress, 2003

[4] Mohamed Gouda, Young-ri Choi, Anish Arora, and Vinayak Naik, The Logical Grid Routing Protocol, Presentation given in PI meeting of NEST program in DARPA, July, 2003