FORMAL MODELING OF A SINK-BASED SPANNING TREE CONSTRUCTION PROTOCOL FOR WIRELESS SENSOR NETWORK

Andrianantenaina Z.1, Randriamaroson R.2, Rakotomiraho S.3

1 Laboratoire de Recherche Systèmes Embarqués, Instrumentation et Modélisation des Systèmes et Dispositifs Electroniques (LR-SE-I-MSDE)
2 École Doctorale en Sciences et Techniques de l’Ingénierie et de l’Innovation (ED-STII)
3 École Supérieure Polytechnique - Université d’Antananarivo, BP 1500 - Antananarivo 101 – Madagascar

ABSTRACT

More and more wireless sensor networks (WSN) applications are proposed and made available nowadays. Network nodes can be highly heterogeneous. Most of the time, nodes have a low processing capacity and a limited energy source. But there are cases where we have within the WSN a node (the sink of the WSN as for example) with better processing performance and a larger source of energy. A major part of the tasks can be assigned to that node. In, this paper, we propose a centralized (sink-based) protocol for a spanning tree construction.

Keyword: wireless sensor networks, spanning tree construction, formal model

1. INTRODUCTION

Wireless sensor networks (WSN) are currently a very active field of research. It covers a variety of applications. A wireless sensor network is made up of several nodes which are, among other things, electronic modules with low capacity in terms of memory and computation. Nodes are often powered by batteries. Nodes have application-specific sensors and perform scheduled measurements which are then sent regularly to the sink for further processing. The sink acts as a concentrator.

The use of WSN in critical applications is currently increasingly being studied. In order to make these networks more reliable, the data exchange between nodes and sink must be optimized to avoid overloading the communication links. An adequate communication structure to address this issue is the spanning tree mechanism. A spanning tree is a structure which minimizes the number of communication links used by maintaining a single path between all pairs of nodes. There may be a multitude of possible spanning trees for a given network. The type of spanning tree constructed from a network depends on the metric to be optimized. This may for example be to minimize the communication delay between the nodes, to minimize the degree of the nodes or to minimize the construction cost in the case of a weighted tree. In this paper, we propose a protocol that minimizes the construction cost by assigning to the sink the tasks of managing the construction of the spanning tree. Sink often has better performance and more energy source compared to other nodes.

The paper is organized into six sections. In the next section, we will first give some definitions on the notion of graphs and spanning trees. Then, in section 3, we will discuss similar works that have already been done for spanning trees construction. In section 4, we will describe our protocol. A formal model of the protocol is given in section 5. A verification will be carried out on this model and we will conclude in section 6.

2. GRAPH AND TREES BACKGROUND

A wireless sensor network may assimilated as a graph. The following definitions give basic background on graphs and trees.

Definition 1 (Graph) A graph $G$ is a pair $(V,E)$ consisting of a finite, non-empty set of vertices $V$ and a set of edges $E \subseteq V \times V$. We write $G = (V, E)$ to denote such a graph. For a given graph $G$, we denote $n=|V|$ the number of vertices of the graph and $m=|E|$ the number of edges.

Definition 2 (Subgraph) Graph $G'=(V', E')$ is a subgraph of $G$ if and only if $V' \subseteq V$ and $E' \subseteq E$.
Definition 3 (Partial graph) Graph $G^\prime=(V^\prime,E^\prime)$ is a partial graph of $G$ if and only if $V^\prime \subseteq V$ and $E^\prime \subseteq E$ and $G$ is connected.

Definition 4 (Path) Let $u,v \in V$ be a pair vertices. A path from $u$ to $v$ denoted $P(u,v)$ is a sequence of vertices $<u,v_0,...,v_k,v>$ such that $\forall i, 0 \leq i < k$, there is an edge in $G$ between the vertices $v_i$ and $v_{i+1}$.

Definition 5 (Connected graph) Graph $G=(V,E)$ is said to be connected if and only if there exists at least one path between all pairs of vertices $u,v \in V$.

Definition 6 (Loop) A loop is a path $(u,v)$ such that $u=v$.

Definition 7 (Neighbour) Let $v \in V$ be a node. The set of $u$ such that $(u,v) \in E$ are called neighbours of $v$.

Definition 8 (Degree of a node) Let $v \in V$ be a node. The degree of $v$ denoted $d(v)$ is the number of neighbours of $v$.

Definition 9 (Tree) Let $T=(V,E)$ be a graph. The following definitions are equivalent to the definition of a tree:
- $T$ is connected and has no loop;
- There is only one path between any pair of vertices $u,v \in V$;
- $T$ is connected, and if one edge of $T$ is deleted $T$ is no more connected;
- $T$ has no loop, and if one edge is added, a loop is created;
- $T$ is connected and has $n-1$ edges.

Definition 10 (Spanning tree) A tree $T=(V,T,E_T)$ is a spanning tree of graph $G$ if and only if $T$ is a partial graph of $G$.

Definition 11 (Parent) Let $T=(V,T,E_T)$ be a tree rooted on $r \in V_T$, the vertex $u \in V_T$ is the parent of vertex $v \in V_T$, if $u$ is the only neighbour of $v$ on the path between $v$ and $r$, denoted as $P(v,r)$.

Definition 12 (Child) Let $T=(V,T,E_T)$ be a tree rooted on $r \in V_T$, every neighbouring vertex $v \in V_T$ which is not a parent of $v$ is called child of $v$.

Definition 13 (Level) Let $T=(V,T,E_T)$ be a tree rooted on $r \in V_T$, the level of a vertex $v \in V_T$ is the distance between $v$ and $r$ and such that the level of $r$ is 0. Two vertices $u$ and $v$ having the same distance with regard to the root are said to be on the same level.

3. SIMILAR WORKS

A large number of spanning tree construction algorithms have been proposed already. The authors of [1] have given an in-depth state of the art in this field. We give only a partial state of the art in this section.

The first to have proposed a breadth-first tree construction algorithm (BFS) are Dolev, Israeli and Moran [2]. Their algorithm works by distance propagation.

Afek, Kutten and Yung [3] designed an algorithm which builds a BFS tree in a general network where each node has an unique identifier (ID). In this algorithm, every node tries to construct a tree in the network from itself. The node of larger identity overruns nodes with lower identity and the tree of largest ID node overruns all the other trees. A node leaves its tree when it detects a local inconsistency. However, to join another tree whose ID is larger a node has to propagates a request message along the new tree branches to the root and to receive a grant message back. This mechanism prevents the necessity to know additional information such as a bound on the network size.

Huang and Lin [4] proposed an algorithm in which a node calculates its distance from all its neighbours as does Dijkstra’s algorithm. Johnen and Tixeul [5] proposed two algorithms for a spanning tree construction in a dynamic environment where the edge weight can change over time. The main contribution of their approach is their focus on the loop-free property which states that the spanning tree must adapt to changes in edge weight without disconnecting nor creating a loop. Gupta and Srimani [6] assume the same dynamism as [5]. They proposed several algorithms, including an algorithm for building an shortest path tree. As their main contribution, they provided an optimal algorithm in space and in convergence time.

Blin, Dolev, Potop-Butucaru and Rovedakis [7] were the first to propose an algorithm using a labeling scheme. Labels were used to find the nearest common ancestor in the tree of two given nodes. By using a mechanism called trains, [8] has further improved this ratio. In [9], the authors propose a memory space optimal algorithm while
remaining polynomial in number of rounds. The algorithm converges first to a spanning tree, then to a minimum spanning tree while maintaining a loop-free property. Proof labeling schemes were one of the main techniques used by the authors.

4. CONTRIBUTION

This section introduces and develops a tree construction algorithm. Given a topology of a network, the algorithm outputs a spanning tree related to that network. Unlike the algorithms mentioned in the previous section which are mostly distributed, we propose here a centrally managed protocol in the sense that there is a node (the sink) which controls the rhythm of the tree construction.

The algorithm that we propose here segregates the nodes into two types:
- The initiator node (the sink) which will become the root of the tree,
- The non-initiator nodes which will constitute the intermediate and leaf nodes of the tree.

The initiator node initializes and terminates the algorithm. Five types of message will be used: Probe, ProbeAck, Iterate, IterateAck and IterateNack. These messages are described in Table 1 below.

The algorithm takes into account the following assumptions:
- each node in the topology knows its neighbours,
- at the start of the algorithm, each node is considered as its own parent and its level value is 0,
- all messages include the identifier (ID) of the sending node,

**Table 1. Message types**

<table>
<thead>
<tr>
<th>Message types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Probe</strong></td>
<td>The <em>Probe</em> message notifies the receiver of its level. This message is broadcasted by the transmitter.</td>
</tr>
<tr>
<td><strong>ProbeAck</strong></td>
<td><em>ProbeAck</em> message is a reply to a <em>Probe</em> message and transmitted also as a broadcast.</td>
</tr>
<tr>
<td><strong>Iterate</strong></td>
<td>The <em>Iterate</em> message instructs nodes having a certain level to subsequently issue a <em>Probe</em> message in its neighbouring after receiving the <em>Iterate</em> message. This message is transmitted in broadcast</td>
</tr>
<tr>
<td><strong>IterateAck</strong></td>
<td>These two messages are generated in response to an <em>Iterate</em> message. They are issued in unicast and are sent to parents directly.</td>
</tr>
</tbody>
</table>

In the topology shown in Figure 1, the initiator will be node 0 and the other nodes will represent the non-initiator nodes. Initialization begins with a Probe message sent by the initiator to all of its
neighbours (see Figure 2). This message includes the level which will be taken into account by all the neighbours of the sink having received it. In this first message, the level has a value of 1.

![Figure 1. Initial topology.](image)

When a non-initiator node first receives a Probe message, it:

- defines the sending node of the message as parent,
- sets its level to equal the level contained in the received Probe message (equal to 1 for the first Probe message sent by the sink),
- sends a ProbeAck message to the parent (see Figure 3).

If a non-initiator node receives a ProbeAck message, it:

- removes the sender of the message from his neighbour list,
- checks if the value of its probeAckCount

![Probe(0)](image)

Figure 2. The initiator node (ID = 0) sends a Probe message

variable is equal or not to the number of its neighbours. If there is an equality, it sends an IterateAck message to its parent.

- checks if the value of the Receiver-ID contained in the message is equal to its ID. If so, it increments its probeAckCount variable by 1. If the value of probeAckCount becomes equal to the number of its neighbours, it sends an IterateAck message to its parent

![ProbeAck(3-0)](image)

Figure 3. Node 3 sends ProbeAck message to node 0. Nodes 2, 4 and 7, which are not the destination, ....

Upon receiving a ProbeAck message, an initiator node increments its probeAckCount variable by 1. If all expected ProbeAck messages (sent by its neighbours) have been received, it:

- increments the iterationRound variable by 1,
- sets the iterateAckCount and iterateNackCount variables to 0,
- sends to all its neighbours an Iterate message (Cf. Figure 4 (b)) containing the value of iterationRound

![iterateNack2-0-1] (a)
When a non-initiator node receives an Iterate message, it first checks the ID of the sending node. If this ID is equal to that of its parent, and the value of the rank contained in the Iterate message is equal to that of the node, the number of neighbours is checked. If the node has at least one neighbouring node, the value of the level contained in the Iterate message is incremented by 1 and this new value of the level is then added to the Probe message which will be sent by the node (Cf. Figure 5). Otherwise, the node sends an IterateNack message to its parent. If the value of the level transmitted in the Iterate message is greater than that of the node, the variables iterateAckCount and iterateNackCount are reset to 0 and the number of neighbours is checked.

In the event that the node has at least one neighbouring node, the Iterate message is forwarded to its neighbours as shown in Figure 7. If the node has no neighbours, it sends back an IterateNack message to its parent.

Upon receiving an IterateAck message, a non-initiator node increments its iterateAckCount variable by 1. The iterateAckCount and iterateNackCount variables are added and the sum is compared to the probeAckCount variable. If there is equality, the node sends back an IterateAck message to its parent.

When receiving an IterateNack message, a non-initiator node increments its iterateNackCount variable by 1. The iterateAckCount and iterateNackCount variables are added. If this sum is equal to the probeAckCount variable, and if iterateAckCount is equal to 0, an IterateNack message is sent to its parent.

If the value of iterateAckCount is non-zero, then the node sends to its parent an IterateAck message.
Upon receipt of an IterateNack message, the sink (initiator) increments its iterateNackCount variable by 1. The iterateAckCount and iterateNackCount variables are then added and the sum is compared to the number of neighbours of the sink. If there is equality, and if the value of iterateAckCount is non-zero, the value of iterationRound is incremented by 1 and a new iteration is initiated. In this case, the iterateAckCount and iterateNackCount variables are reset to 0 and an Iterate message containing the new value of iterationRound is broadcasted.

The algorithm ends if the initiator node no longer receives an IterateAck message, meaning that the variable iterateAckCount remains equal to 0. Figure 8 sketches the spanning tree constructed from the topology of Figure 1 at the end of the algorithm.

5. FORMAL MODELING AND VERIFICATION

In this section, a formal modeling of the protocol is done. This is to ensure that the proposed protocol is working properly. To do this, we have chosen the UPPAAL model checker.

The UPPAAL model checking tool [10] offers the possibility of implementing the model as an automaton. In UPPAAL, an automaton is defined by its states (called Location), its transitions, its invariants, guards and updates. UPPAAL allows to declare variables and arrays of variables which are particularly useful for implementing a representation of a topology of any network. The data can be local to a node or global. It is possible to write updates to the data (executed through transitions) by using functions written in a language derived from C. Synchronizations between transitions are performed on variables called channels declared as type chan in UPPAAL.

Figure 6. (a) Node 7 sends a ProbeAck message. (b) Node 3 sends an IterateAck message to node 0. (c) Node 3 becomes parent of node 7

When the sink (initiator) receives an IterateAck message, it increments its iterateAckCount variable by 1 (Cf. Figure 6 (b)). It then compares the sum of the iterateAckCount and iterateNackCount variables to the number of its neighbours. If there is equality, the iterationRound variable is incremented by 1. Before the start of the new iteration, the iterateAckCount and iterateNackCount variables are reset to 0 and an Iterate message containing the new value of iterationRound is broadcasted.

Figure 7. (a) Node 0 sends an Iterate message with iterationRound = 2
(b) Node 3 retransmits this Iterate message to nodes 4 and 7

Figure 8. Constructed spanning tree
The proposed protocol model is composed of two types of automaton: the first one is for the initiator node and the second for the non-initiator nodes.

5.1. The initiator node model

When the network is initialized, the initiator node sends a synchronization on the probe! channel. This initialization is modeled by the transition in Figure 9.

![Figure 9. Protocol initialization](image)

Once a direct neighbour receives this synchronization, it sends back a synchronization on the probeAck[id_G]! channel, id_G indicating the initiator ID. The left branch of Figure 10 represents this mechanism.

The iterate! channel is used by the initiator node to rhythm the construction of the tree. The right branch of the model in Figure 10 implements this action. The complete model of the initiator node is depicted in Figure 11.

Declaration of global variables

Global variables are common to both initiator node and non-initiator node models. They are listed in Table 2.

Local declaration for the initiator node

The variables and codes below are specific to the model of the initiating node.

![Figure 10. Modeling ProbeAck exchanges](image)

In order to control the activation of the iterate! transition, we have created a boolean variable sendIterate. sendIterate is true if and only if all the conditions for sending an msgIterate message dictated by the protocol are satisfied. Another boolean variable has also been created: algEnd. algEnd becomes true if the initiating node no longer receives an msgIterateAck message. The transition to the End location is then activated and the algorithm ends.

We have also created three functions (rcvProbeAck(), rcvIterateAck() and rcvIterateNack()) in order to control the behavior of the initiating node when synchronizations on probeAck[id_G], iterateAck[id_G], iterateNack[id_G] are activated, the variable id_G of type Node_Id being the identifier of the initiator node.
<table>
<thead>
<tr>
<th>Type/Name of the variable</th>
<th>Description</th>
<th>Type/Name of the variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int N</td>
<td>Total number of nodes</td>
<td>Node_Id</td>
<td>Type of node ID variable</td>
</tr>
<tr>
<td>int MAX_STEP</td>
<td>Max number of iteration</td>
<td>int V[Node_Id]</td>
<td>Array of neighbours</td>
</tr>
<tr>
<td>int iterationRound</td>
<td>Current iteration round</td>
<td>int MAX_LEVEL</td>
<td>Max level value</td>
</tr>
<tr>
<td>probe, iterate</td>
<td>Broadcast channels used to synchronize exchange of messages Probe and Iterate</td>
<td>probeAck[Node_Id], iterateAck[Node_Id], iterateNack[Node_Id]</td>
<td>Synchronisation channel used for ProbeAck and Iterate messages</td>
</tr>
<tr>
<td>IsC[int][int]</td>
<td>Connectivity matrix. Two nodes i and j can communicate to each other if : isC[i][j] = true</td>
<td>msgProbe, msgIterate, msgProbeAck, msgIterateAck, msgIterateNack</td>
<td>Variables representing different messages of the protocol</td>
</tr>
</tbody>
</table>

5.2. The non-initiator node model

The execution of the algorithm of a non-initiator node does not begin until a synchronization is received on the probe! channel. A ProbeAck message is then sent to the node that initiated the synchronization. In UPPAAL, the mechanism is illustrated in Figure 12.

The full model of non-initiating nodes is given in Figure 13.

![Figure 12. Modeling of ProbeAck message exchanges of non-initiator nodes](image-url)
Local declaration for the non-initiator nodes

The following variables and codes are specific to the non-initiator node model.

We have declared several boolean variables in order to have control over the different model transitions: sendProbe, sendProbeAck, sendIterate, sendIterateAck, sendIterateNack.

Trivially, the id and parent variables, of type Node_Id, designate the current node and its parent.

The functions rcvProbe(), rcvIterate(), rcvProbeAck(), rcvIterateAck(), rcvIterateNack() are respectively called when the synchronized transitions are activated by probe?, iterate?, probeAck?id?, iterateAck?id? and iterateNack?id?. Code samples of the rcvProbe() and rcvIterate() functions are given by Listing 1 and Listing 2 below.

```c
void rcvIterate() {
    if (parent == msgIterate.s_id) {
        if (rang == msgIterate.rang) {
            if (V[id]==1) {
                sendIterateNack = true;
            } else sendProbe = true;
        } else {
            if (rang < msgIterate.rang) {
                iterateAckCount = 0;
                iterateNackCount = 0;
                if (V[id]==1) sendIterateNack = true;
            } else sendIterate = true;
        }
    }
}
```

Listing 2. Function rcvIterate()

which for each node holds that its parent is a correct parent.

**Property 2** Along all paths eventually a state is reached in which for each node holds that its level is correct.

**Property 3** Along all paths eventually a state is reached in which each node selected a correct parent after MAX_STEP iterations.

**Property 4** Along all paths eventually a state is reached in which each node computed the correct level value) after MAX_STEP iterations.

6. CONCLUSION

The protocol proposed in this paper guarantees that a spanning tree will be constructed at the end of the execution of the algorithm. It is optimized for a heterogeneous wireless sensor network that has one node (the sink, as for example) having better performance compared to other nodes in the network. From the point of view of non-initiator nodes, the protocol minimizes the construction cost in the sense that the tasks of managing the construction of the spanning tree are centralized at the initiator node level. The formal verification carried out on the models proposed in this paper have confirmed that all the properties required for a correct functioning of the proposed protocol are satisfied.
REFERENCES


