

MICS Summer Internship  
Implementation of an energy efficient data  
collection algorithm in TinyOS/SlotOS

Heidi Gumpert

Supervisors: Roland Flury and Philipp Sommer  
Professor: Roger Wattenhofer  
Distributed Computer Group

ETH Zurich  
Zurich, Switzerland

October 12, 2009

## 1 Introduction

Sensor networks are spatially distributed networks of nodes which can be equipped to measure various properties about the area in which they are located. These networks are typically installed over an area to collect data over long periods of time. This requires that the applications running on these nodes are energy efficient, so that the battery replacement is minimal. Also it is often required to aggregate all measured data to one sink node, so that only a single node must be visited by a researcher in order to collect the data.

In this report an implementation of an energy efficient data collection algorithm is presented. The use of this implemented module can assist those who are writing applications for sensor networks which need to aggregate data to one sink node. This implementation takes advantage of a synchronous global network provided by the SlotOS operating system that is used. This implementation takes many of its key ideas from a previous implementation called Dozer. The main difference between the proposed algorithm and Dozer is that this algorithm exploits the synchronicity of the nodes provided by the operating system. This synchronicity allows for easier implementation and debugging.

## 2 Previous Work

A previous implementation of such a data collecting algorithm called Dozer had been proposed by [1]. This algorithm first establishes a tree topology for the set of nodes in the network and defines a sink node where all the data will be aggregated to. Next using the topology, each node establishes its communication schedule with its parent and children. Slots for communication are reserved with its parent and children and this schedule is repeated as long as the topology of the network has not changed. Most repeated communication collisions are avoided by having the length of the schedules be random and thus not having the same period as other nodes in the area. The nodes wake up when they are to send messages up to their parents or when they are to receive messages from their children. This schedule of when the nodes are to wake up to communicate is the key idea that promotes energy efficiency, since the nodes only turn on their radios when communication is scheduled to occur.

The proposed implementation is done using the SlotOS operating system [2]. The SlotOS operating system allows the user to organize their implemented modules into time slots to create a schedule of when the applications are to be executed. This schedule is then repeated cyclically. A portion of the time slots are required for operating system maintenance. This would include maintaining the global time synchronization.

## 3 Implementation Design

The proposed implementation of the data collection algorithm operates using three modules: one to create the tree topology, one to create the communication

schedule and one to manage the user provided data that is to be aggregated at the sink node.

### 3.1 Bidirectional Tree

The Bidirectional Tree module creates a tree topology to facilitate bidirectional communication. With one node set as the root, the module finds a parent for each node and creates a list of children that have that node as their parent.

The module has three phases per round.

**hello phase** Within the first portion of the module, each node that has been incorporated into the network sends out a hello beacon. Nodes that have not yet been incorporated into the network listen for these hello beacons in order to find a suitable parent to connect to. The hello beacon is sent out at a random time within this phase to avoid collisions with other nodes hello beacons.

**updating history of communication** A measure of the reliability of the communication link is needed when a node is to select a parent, or to discern if a communication link still exists between its parent or one of its children. The history variable serves this purpose and is updated after the end of the hello phase when all hello beacons have been sent out. The previous entries in the history record are shifted up and the last position is updated to true if a nearby node was heard, otherwise false.

If the last three entries are false, it is assumed that the communication to this node has been lost. If this occurs for the parent of a node, then the child will attempt to connect to another node it has heard in the subsequent ping phase. If this occurs for one of the node's children, then it will remove that child from its list of children.

**connect phase** This phase provides an opportunity for nodes to connect to a parent. An unconnected node must hear the hello beacon twice in successive rounds before it will attempt to connect to that node as its parent. This assists in ensuring that the child will reliably be able to receive communication from the parent.

To obtain a parent, the child sends a connect message to the potential parent. If the parent has not reached the maximum number of children it will return an acknowledgement message and the child knows that the parent has accepted it as one of its children. This acknowledgement message ensures that bidirectional communication between parent and child is possible.

A child can also attempt to switch parents in this phase if it has found a parent which is closer to the root node. Upon receiving the acknowledgement message from the new parent, it switches to the new parent.

The above describes the basic functionality of the Bidirectional Tree module; the bootstrapping state to create a tree. However, when the tree is created and the topology remains stable there is a lot of wasted communication. It is unnecessary to send out hello beacons every round and to listen for potential connect

messages when the nodes have all be incorporated into the tree. Additionally, when the produced tree is used for communication by other modules, then these modules, by use of the tree, can discover if the communication links in the tree are still functional. After each communication, the modules can let the tree know if the communication was successful or not and the tree can then update the communication history. This describes the established state of the tree.

The state of the topology moves into the established state after a specified number of rounds of no topology change in the bootstrapping phase. In the proposed implementation eight rounds of no topology change was used as the threshold. Since it is still possible that a nearby node might lose its parent and would require to find a new parent, each node in the established state must still send out hello beacons once in a while to provide an opportunity for a nearby node to connect to it. Even though a node must still send out some hello beacons and listen for possible connect messages, there is still quite a bit of energy saving for the rounds that the node is not sending out messages.

It is imperative that the users of this Bidirectional Tree module use the interface to let the tree know if communication between its parent and children is still successful. Additionally, users of the tree must should also query the tree to obtain the current parent and list of children in case there had been a change between rounds.

### **3.2 Data Scheduler**

The role of the Data Scheduler is to determine a schedule of when to send messages to the parent and when to expect messages from the children. This schedule between parents and children allows the node to know when the radio must be on for communication. The communication start time between two nodes is random but based on a shared random seed.

At the beginning of the module, the Bidirectional Tree module is queried to obtain the current parent and list of children and it determines if there were any changes. If there is a new parent or child, then a new random seed for the communication time is set based on the ID (the node's own ID is used if there is a new parent, or the child's ID if there is a new child). The use of random seeds in this way provide a mechanism for both parent and child to establish random yet defined communication times between each other.

After any changes in the topology are dealt with, the module then sets the timers for communication with the parent and children. Only two timers are used: one for parent communication and one for child communication. Once the child timer fires, the timer is reset for next communication with another child.

There is only one communication round between two nodes per execution slot of the module. However, during this communication round it is possible to send successive messages. Within the message that is sent to the parent, there is a variable that indicates whether the child has more to send to the parent. This variable is set to true if it has more data to send and if its communication time to its parent has not yet been interrupted by a communication round with one of its own children. The parent sends back a variable in its acknowledgement message

indicating whether it can accept more messages from the child. This is set to true while it has not been interrupted by either the start of a communication round to its own parent or the start of a communication round to another child.

At the end of the module, the Bidirectional Tree interface is used to update the success of sending messages between the nodes. If at least one message has been successfully passed, then communication between the nodes is considered to still be possible. This metric is influenced by how many messages are sent between the nodes. It can be unforgiving when only one message is required to be sent per round and this one message was not sent successfully.

### 3.3 Data Manager and Data Sender

The Data Manager and the Data Sender provide interfaces to handle the data that is to be sent to the sink node. The Data Manager is the interface for the Data Scheduler. The Data Scheduler queries the Data Manager to obtain the data to send. The Data Sender is the interface for the user to submit data to be sent. A separate interface was provided so users would have a simplified interface without unnecessary functionality and to hide implementation details.

The Data Manager takes care of most of the implementation details. The Data Scheduler queries it to obtain the next piece of data to send. It also submits data it has received from children to the Data Manager.

The Data Manager has a circular buffer in which it puts the data to send. The Data Scheduler notifies the Data Manager after messages have been sent successfully and it then removes it from the buffer. If the buffer is full, it first tries to remove an entry with the same ID as the one being inserted. If there are no entries with that node ID, then it removes an entry from a node that has multiple entries in the buffer. Both of these measures are an attempt to prevent starvation for any nodes.

The Data Manager also has a function for the Data Scheduler to query if there is any data to be sent. As explained above, this is needed to determine if there are more messages that can be sent.

The Data Sender interface provides a function to submit data to send. A user can first query if there is space left in the buffer. The user may wish to implement a back-up strategy to save data if the buffer is full. There are two events that must be implemented by users of the Data Sender interface: one that alerts the loss of communication, and one to receive the data. It is important for the node to know if data cannot be sent up towards the root. This event is currently only fired when the communication is lost between the node and its parent. Again, the user may want to implement a back-up strategy to save data while the connection is not present. The other event, receiving the data, only needs to be implemented by the root node. Accepting and sending up the data further towards the sink is not the responsibility of the user - this is taken care of by the Data Manager as internal functionality.

## 4 Results

A small test bed was set up to test the implemented modules. Although the Bidirectional Tree module had been tested and shown the correct behavior, it would have been difficult to use this module in the office environment. It is difficult to place the nodes in such a way so that there are multiple hops for the data to be passed over. Within the Bidirectional Tree module a white-list was used to restrict radio communication between nodes so that a predefined tree structure would be established. Also for the purposes of this report, restricting the test set up to an undisturbed office space reduces the likeliness of random communication interference. The test was run for approximately 16 hours, which meant that in total 2368 rounds of the modules were executed with each round being 32 seconds.

Below in Figure 1 is the set up of the tree that was used in the test. Each of the non-sink nodes attempting to send up two messages per round to the sink node.

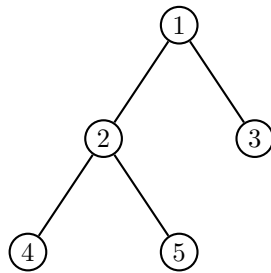


Figure 1: Tree Set-Up of Nodes

Various data during regarding the functionality of the modules were recorded. At the end of each execution of the Data Scheduler module, the number of messages sent and received were recorded. Additionally, the amount of time the radio was turned on was also calculated.

### 4.1 Messages

The number of messages each node sent and received was recorded. Below is a chart which summarizes the number of messages each node sent and received over the 2368 rounds of communication.

Table 1: Messages Received

Node	Messages Received from Node	Average Number Received	Maximum Number Received
1	2	6.15	33
	3	2.00	8
2	4	2.24	32
	5	2.29	32

Table 2: Messages Sent

Node	Messages Sent to Node	Average Number Sent	Maximum Number Sent
2	1	6.13	33
3		2.00	8
4	2	1.89	32
5		2.00	32

From Table 1 and 2, one can see that approximately two messages are sent per round. Node 1 received on average six messages from node 2 per round; one from node 2 itself and the last four from node 2's children.

However by comparing the average number of messages sent and received between node 2 and nodes 4 and 5, there appears to be a small discrepancy. Node 2 received on average 2.24 and 2.29 messages per round from nodes 4 and 5, respectively. Nodes 4 and 5 sent 1.89 and 2.00 messages per round to node 2. After inspection of the code, it appears that there was an error in the counting of the messages on the receiving end. Sequence numbers were used in order to ensure that messages were not missed and that duplicate messages were not received. Sequence numbers internally increment a counter for each node it communicates with after receiving the next message from that node. If the number of the message and the number of the counter do not match, then a message was missed or duplicated. The testing counter for messages received should have only been incremented when sequence numbers were incremented as well. This was not the case in the receiving code, so it is suspected that the counter was incremented even for duplicated messages. These additional increments of the counter could explain for the small discrepancy.

The maximum number of messages sent is in order with the message buffer. A buffer of size 32 was used. Node 1 received only a maximum of 8 messages from node 3 likely because node 3 established a connection to node 1 quickly and so never accumulated messages for more than the initial rounds. Node 2 sent a maximum of 33 messages to node 1. Although the buffer has only size for 32 messages, it is possible that there was a small overlap between switching from accepting to sending messages.

## 4.2 Radio Up-Time

The amount of time the radio was turned on for per round was recorded. This amount is the total time the radio was turned on, and so it also includes the radio time used by the operating system. Not including the operating system radio time would make this measure invalid since these modules use the time synchronization provided by the operating system.

Below Table 3 summarizes the amount of time the radio was on for each node.

The average times for the various nodes are as expected. Node 1, the sink node, has the longest average radio up-time with node 2 having the second longest average uptime. The root node must receive all messages whereas the second

Table 3: Radio Up-Time

Node	Average Time (s)	Maximum Time (s)	Minimum Time (s)	Percentage Average Usage (%)
1	3.207	9.504	0.008	10.02
2	2.475	9.393	0.017	7.73
3	0.954	5.750	0.044	2.98
4	1.756	6.835	0.016	5.49
5	1.717	5.836	0.15	5.37

node must receive messages from its two children and also send its own messages to the root.

Nodes 3, 4, and 5 are all leaves on the tree and as such do not need to do as much communication as the the internal nodes, which is reflected in the average radio up-times. It is interesting to note that nodes 4 and 5 both had their radios on approximately 0.8s more on average than node 3. After examining the data log, there are times that the node 4 or 5 disconnected from their parent node. After this occurs, they must attempt to reconnect to a parent which involves them listening for hello beacons in the Bidirectional Tree module. The radio time spent to listen for these beacons adds up considerably especially when their parent is in the established state and is not sending out beacons every round.

The maximum radio up-times is also larger for the two nodes that receive messages than for the nodes that only send messages. This is likely due their busier scheduler communicating with other nodes. The deviation in radio-up time between the maximum and the average values could arise from children sending out more messages in some rounds than others. The children attempted to send two messages per round to the sink. The Data Manager puts the messages into a buffer which the Data Scheduler attempts to clear during the time it sends data to the parent. If communication was suspended for a few rounds, this buffer would fill up and many messages would be sent once communication was re-established.

## 5 Further Work

The proposed implementation provides a good framework on which one could make enhancements to achieve further energy efficiency. Currently the main energy savings come from how the Bidirectional Tree is constructed and maintained and from the radio usage scheduling from the Data Scheduler. However, certain improvements could be made in order to improve the energy efficiency. Also included below are some ideas which would improve the quality of the implementation.

### 5.1 Potential Bidirectional Tree Improvements

Currently, there are measures done in the formation and maintenance of the tree to save energy such as the implementation of the established state. During

the established state, the radio need not be used for sending out beacons for new nodes to connect. Also, disruptions in the topology are detected by the modules using the Bidirectional Tree as they are required to update whether communication is successful or not.

One potential improvement to reduce the radio up time could be to introduce a contention window for nodes to connect such as in [1]. After a node sends out its beacon, it waits to hear if there is a busy tone. Only if it heard a busy tone would it turn on its radio in the connect phase of the tree module to let a node connect.

A suspended state should also be implemented for nodes that cannot find a parent to connect to. Currently, they repeatedly attempt to find a parent to connect to. This wastes a lot of energy whereas it may be more prudent to wait a period of time to try again, as done in [1]. The sensor node may then have some power left to still be able to take measurements and save the results to a hard drive to be collected manually later.

## 5.2 Potential Data Scheduler Improvements

One area with a short-coming in the implemented Data Scheduling module is the lack of warning when the end of the module time slot is to occur. Since the Data Scheduler can send multiple messages, it can sometimes send or receive a message after its allotted slot in the SlotOS schedule. To fix this issue, a separate timer should be added to warn when the end of the module time slot is about to end. The logic to determine if more messages can be sent or received should take into account whether this timer has fired yet. No more messages should be sent or received if the module time slot is due to occur soon.

## 5.3 Potential Data Manager Improvements

A potential improvement for the Data Manager module would be to provide a mechanism to save data to some type of external storage. This would be particularly useful when communication towards the sink has been lost and the node would need to store data for some period of time. The data would then either need to be collected afterwards if the connection was never re-established, or the data could be sent towards the sink node when the communication is re-established.

# 6 Conclusions

In conclusion, a basic implementation to collect data among a set of nodes was implemented for the SlotOS operating system. The implementation used ideas in the construction of the network topology tree and the scheduling of the data transmission to minimize radio up time in order to be energy efficient. Additionally, the implementation could be used as a framework in order to

do further improvements to achieve a more efficient and fully featured data collection module.

## References

- [1] Burri N, von Rickenbach P, Wattenhofer R. Dozer: Ultra-Low Power Data Gathering in Sensor Networks. *International Conference on Information Processing in Sensor Networks (IPSN)*, Cambridge, Massachusetts, USA, April 2007.
- [2] Flury R. 2009. Routing on the Geometry of Wireless Ad Hoc Networks. Thesis (PhD) ETH Zurich.