Aggregation Query Under Uncertainty in Sensor Networks

CS 252 Project

Yozo Hida Paul Huang Rajesh Nishtala
Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720
(yozo@cs.berkeley.edu, pbhuang@bmrc.berkeley.edu, rajeshn@cs.berkeley.edu)

Abstract

In sensor networks, aggregation is often used to obtain some form of summary of the sensor values, such as the maximum and the average. When there are hundreds of nodes, it is inevitable that some of these sensor will malfunction and report faulty sensor values or fail to route the value information, adversely affecting the aggregate result. In this paper, we describe a simple method to locally detect outliers in sensor network to make the aggregate queries more tolerant to faulty sensor readings and failed nodes. We also describe the results of implementing this method in the TinyDB [10] aggregation service running on the Nido sensor network simulator included in the TinyOS [17, 7] distribution.

1 Introduction

Wireless sensor networks have gained momentum in the research community in recent years. Large scale wireless sensor networks often require some sort of data aggregation to occur since not all sensor readings can be reported due to energy or communication constraints. The wireless devices used in such networks, called *motes*, are designed to be inexpensive; in large network some of them will inevitably fail one way or another. Some sensors may malfunction and report an abnormally high (or low) sensor values. Others may stop responding altogether because it has run out of power or its transmitter has failed. These sorts of errors can make the aggregate result meaningless.

While the number of failed sensors may be a small fraction of the total number of sensors, they can still drastically affect the result of some aggregate results. The sensitivity to node failures depends on the aggregate operator; for example if some sensor keeps reporting abnormally high temperature, the maximum will be affected much more than the average, while others such as the count or the median will be mostly unaffected.

In many cases, such as when collecting a temperature readings of a building, the underlying data we are trying to obtain from the sensor readings are continuous in time or space, or both. In this paper we take advantage of this property to perform outlier detection to make simple aggregation operation (such as MAX and AVG) more reliable under the presence of faulty or failed motes. To deal with failed nodes, we have implement a multipath algorithm so the motes has more than one route to deliver the information. We implement our algorithm on the TinyDB [10, 11] aggregation service running on Nido sensor network simulator provided by the TinyOS [7, 17] distribution. To supply the simulator with sensor values, we create a data model to map a function over the area covered by the motes.

The rest of the paper is organized as follows. In section 2 and 3 describes our outlier detection algorithm and the multipath algorithm used. Section 4 discusses the data model being used to test our modifications. Section 5 then discusses the various parameters we varied in our tests and presents the results. Section 6 discusses related work that has been done in this field. Section 7 discusses future work to be done (including quite a few things we had planned to investigate but was omitted due to time limitations[1]). Section 8 gives some concluding remarks. Finally, the appendix will give some implementation details that may be of interest to users in need of outlier detection in TinyDB.

2 Outlier Detection

2.1 Overview

We use a pair of statistical test to determine whether the incoming value is an outlier or not. If the new value passes the outlier detection, it is allowed to be aggregated as usual; otherwise it is flagged as an outlier and it is eliminated and the node waits for the next value (which may very well be faulty as well).

We assume that the data sets are smooth with respect to time and space. If there is a sensor value discontinuity with respect to space, we assume that there is some patch of sensor network that reports both side of the jump. We also assume near normal distribution of data over local space and over small time steps.

Each node will keep track of recent sensor values of all the nodes in the local neighborhood (which we call *s*phere of influence). As values of sensor nodes in the sphere of influence is reported one by one, we perform two statistical tests. First it compares against most recent values of all other nodes in the neighborhood (including itself). Next it tests against previous values (up to 5 epochs ago) of the neighborhood to take into account variance over time. If both of these tests pass, it means the value is valid; otherwise, it is flagged as an outlier and discarded from the analysis. Currently a sphere of influence of level 1, meaning the neighborhood consisting of all the nodes that can be reached in at most 1 hops, has been implemented and tested. Larger neighborhood can be used if the node density is high (or if the function is very smooth) to better detect outliers, at the cost of extra messages.

2.2 The Aggregate Value versus the Local Value

Since the outlier detection relies on the sensor readings of the immediate neighborhood of the node, we modified the aggregation so that each node will now report its local sensor reading as well as the aggregated result. We just tag this extra information onto the query result message, this does not require any extra message.

A simple example will illustrate why we need to use the local sensor value readings instead of the aggregate value. Suppose that some portion of the sensor network senses a relatively high temperature reading: a sizable cluster far away from the root reports these high values. In this

[1]In retrospect, it may have been better to test out our algorithm in a simpler simulator written from scratch rather than to attempt to directly implement it in the existing Nido simulator and TinyDB, especially since we only require a small subset of the functionality offered. Both the Nido simulator and TinyDB are still under heavy development, and therefore quite unstable. A bulk of our time was spent in debugging the various portions of the Nido simulator and TinyDB unrelated to our outlier detection.

"hot spot", these values are marked as valid, and propagated up the routing tree towards the root. If each node only reports the aggregated result, when the result from this "hot spot" reaches the the nodes near the root, where the temperature is relatively cool, if will be flagged as an outlier and hence discarded (and thus the true max will not be reported). The fact that aggregated result may come from geographically far region means that it may appear discontinuous if seen from a node away from that region.

This is easily fixed by making each node report the local value as well as the aggregate value. It is assumed that the aggregate value has passed through all the outlier detection mechanisms and is a valid value and is therefore considered the trusted value. The local value that each node reports is considered untrusted and the outlier detection mechanisms are run on these values to eliminate outliers. If that value is determined to be valid, it is allowed to be merged with the trusted aggregate value. One should note that this mechanism does not prevent malicious nodes to intentionally report false aggregate values.

2.3 Outlier Test

At each node, a data structure called *history buffer* is kept to perform outlier detection. For every expression in the query, the history buffer keeps track of the list of motes within the sphere of influence (including itself). For each of these entries, a FIFO queue of (sensor value, epoch time) pair is stored. Data is added into the queue only when the value has been determined to be valid.

The actual outlier detection proceeds in two phases, termed *n*eighbor test and *h*istory test. During the neighbor test, the mean and standard deviation of most recent sensor values from all the nodes within the neighborhood is computed. If the incoming value is outside of 2.5 standard deviations of the mean, then it is discarded. This test essentially assumes spacial continuity; if the given child node is too different from all the other neighbors, it is flagged as an outlier. If a normal distribution of the sensor data is assumed, roughly 98.8% of all the true data would be passed.

The history buffer test scans all entries in the history buffer for a given expression in the query and calculates the mean and the standard deviation. The incoming value is then compared against these means and standard deviations in much the same was as in the neighbor test. This takes into account the variability of sensor values over time, so that if the sensor values are rapidly changing, the outlier detection will not flag valid data as invalid.

In addition to using the computed standard deviation s, there is a user specified deviation σ that is used to allow certain values even if it lies outside of 2.5 standard deviation from the mean. This prevents the false outlier detection from occurring when the neighbor happens to have a very similar value. For example, if a local set of nodes stays constant at temperature 200, the computed standard deviation is essentially zero. This causes any value other than 200 to be rejected. Thus in this case we use the user supplied deviation parameter σ to allow certain value to be accepted regardless.

2.4 When Outlier Detection Needs to be Done

There are three places in which the outlier detection mechanisms must be called; only two of these places does the result of the detection actually alter the flow of events. Other case just updates the internal state (so that later inputs can be tested accurately). The three places are

1. A node receives a query result from that branch of the routing tree. In this case, we just perform the outlier detection and merge the result only if it passes our tests.

2. A node snoops a neighbor broadcasting a sensor value (and the aggregate value). This mechanism implements the level 1 sphere of influence: all the nodes within communication distance is considered. In this case, we update the internal table of neighbor sensor values, but do not update the aggregate value of this node. This prevents multiple results to be reported to the root, which can affect some aggregate operator such as the average. For other operators, such as the maximum and minimum, we can update our aggregate result.

3. A node reads in its own local sensor readings. We need to make sure that if this local sensor readings is detected as an outlier (with respect to the sphere of influence of that node), then it is not used in aggregation (but still reported up to the parent as the local value).

In each case, the internal history buffer is updated to reflect the new sensor readings from its neighbor (or itself).

3 Multipath Routing

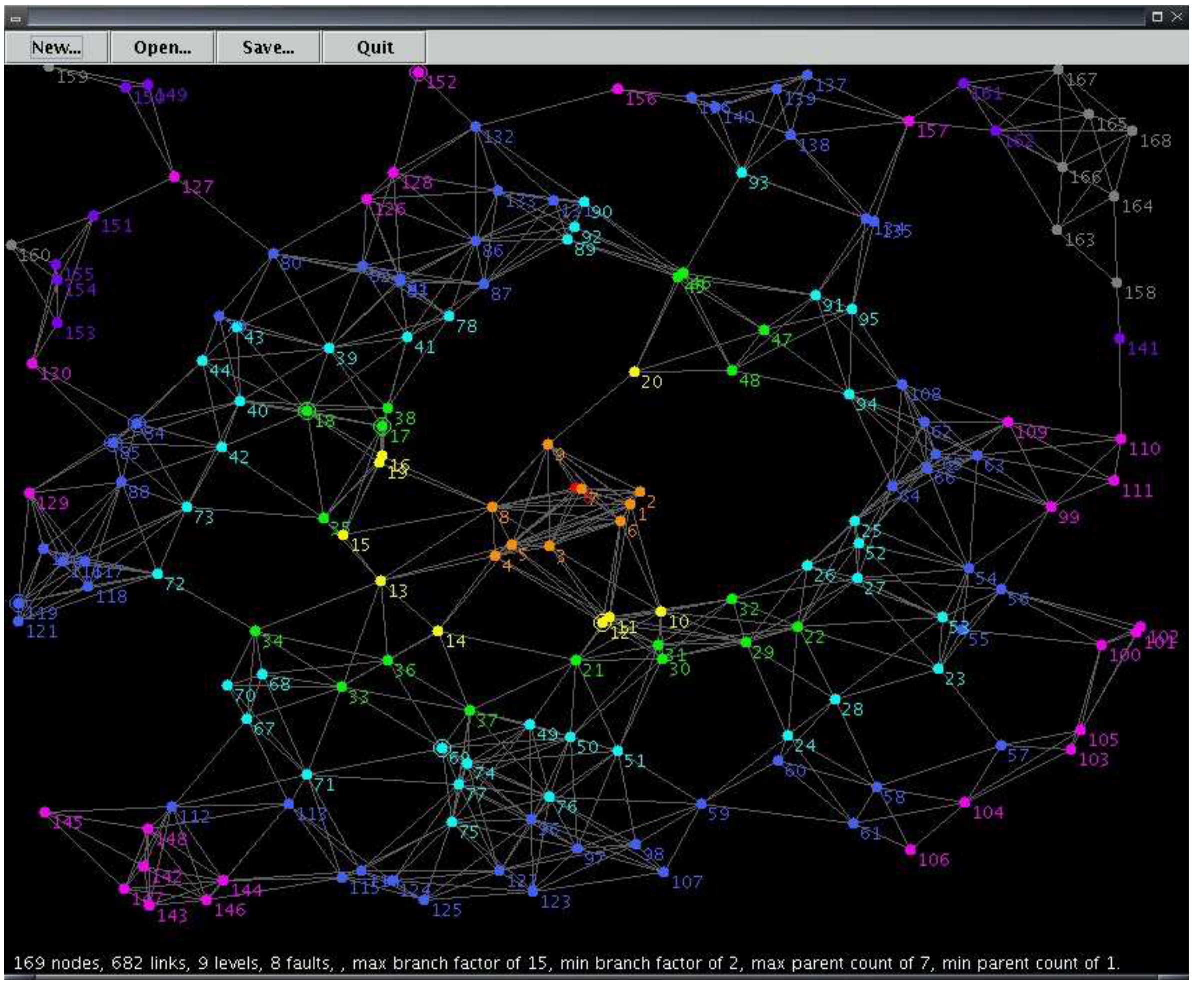
In unreliable networks, multipath routing can help in data aggregations to be more resilient to faulty motes. Multipath routing is an idea that has been used in many wireless networks [9, 14, 12]. These networks are multihop wireless networks that give little thought to energy problems in the wireless sensor networks. Multipath routing has been researched in the context of energy aware wireless sensor networks [4, 5, 15, 10]. In [5], the multipath routing algorithm has the drawback of using only one sink and source. While [15] and [4] algorithms are more concerned with energy efficiency in the wireless sensor networks, their other criteria in the algorithm design is temporal message order, which is not as important in aggregations. So we decided to use a very simple [10] algorithm and alter the re-choose parent interval.

The TinyDB algorithm [10] for setting up the routing tree is as follows. An aggregation query is injected in the root mote. The root mote is at level 0. When the root mote broadcasts the query, any mote that can hear the root mote is placed at level 1. Then the level 1 motes broadcast the query again. Any mote that can hear the level 1 message and has not been assigned a level are placed into the level 2. The process continues until the query reaches leaves of the tree.

Once the routing tree is setup, each mote keeps track of the signal strength of the neighbor motes along with time last heard. The multipath routing involves a mote to re-choose a parent every 2 epochs. The algorithm to determine a new parent is picking a parent with level lower than the mote itself and having the highest signal strength if there are ties. The parent_reselect_interval can be changed as often as 1 epoch. We want to keep the reselect interval short so that the data aggregation can report back data faster when a faulty mote appears. When a mote has not heard from its parent in 10 epochs, it will re-choose a new parent. This interval, in the variable parent_lost_interval, is adjustable.

4 Data Modeling

The inputs to the simulation are the sensor network itself (node placements and connectivity) and the sensor value it reports at a given time t. We place the nodes in 2-dimensional square



169 nodes, 682 links, 9 levels, 8 faults, , max branch factor of 15, min branch factor of 2, max parent count of 7, min parent count of 1.

Figure 1: Random distribution of nodes. Root (in red) is located in the center, and other nodes are color coded depending on the distance from the root. Those with rings are the faulty nodes.

$[-1, 1] \times [-1, 1]$ in the following three manners:

1. Square grid. Total of $n = k^2$ nodes (k odd) is placed with the root node at the center.
2. Hexagonal grid. Total of $n = 1 + 3k(k + 1)$ nodes in k concentric hexagonal rings around the root node.
3. Random point placement. We place n nodes in uniform random manner.

All of the data sets can be easily generated using the Java tool depicted in figure 1, which shows a random graph of 100 nodes.

To test a realistic data, we have tested our algorithm on several different kinds of data sets, including ones with spatial discontinuity. The following four sensor value distribution was used (actual value reported by the nodes are multiplied by 400 to make it an 16-bit integer):

1. Constant function moving over time:

$$f(x, y, t) = 1 + \frac{\sin t}{10}$$

   This is provided as the most basic example to sanity-check our algorithm.

2. Function with fixed maximum location.

$$f(x, y, t) = \frac{\cos t}{1 + r^2} \qquad \text{where } r^2 = \left[\frac{x-1}{2}\right]^2 + \left[\frac{y-1}{2}\right]^2.$$

   This tests whether our outlier detection can keep track of outliers when the function values change over space. See figure 2.

3. Function whose location of maximum varies.

$$f(x, y, t) = 1 + \frac{\cos\left(t + 2\pi\sqrt{u + 2v(1.1 + \sin t)}\right)}{\sin t + 2e^{0.5\sqrt{u+v}}}$$

   where

$$u = \left[\frac{x-1}{2}\right]^2 \qquad \text{and} \qquad v = \left[\frac{y-1}{2}\right]^2.$$

   This function displays more complex behavior, where the size of the maximum and the location of the maximum changes continuously over time. See figure 3.

4. Function with a spacial discontinuity.

$$f(x, y, t) = \begin{cases} 0 & \text{if } x^2 + y^2 \leq 0.5 \\ \frac{\sin t}{1+(x-0.5)^2+(y-0.5)^2} & \text{otherwise.} \end{cases}$$

   This function has a boundary where one side has a value of 0 while the other side has a some positive value. See figure 4.

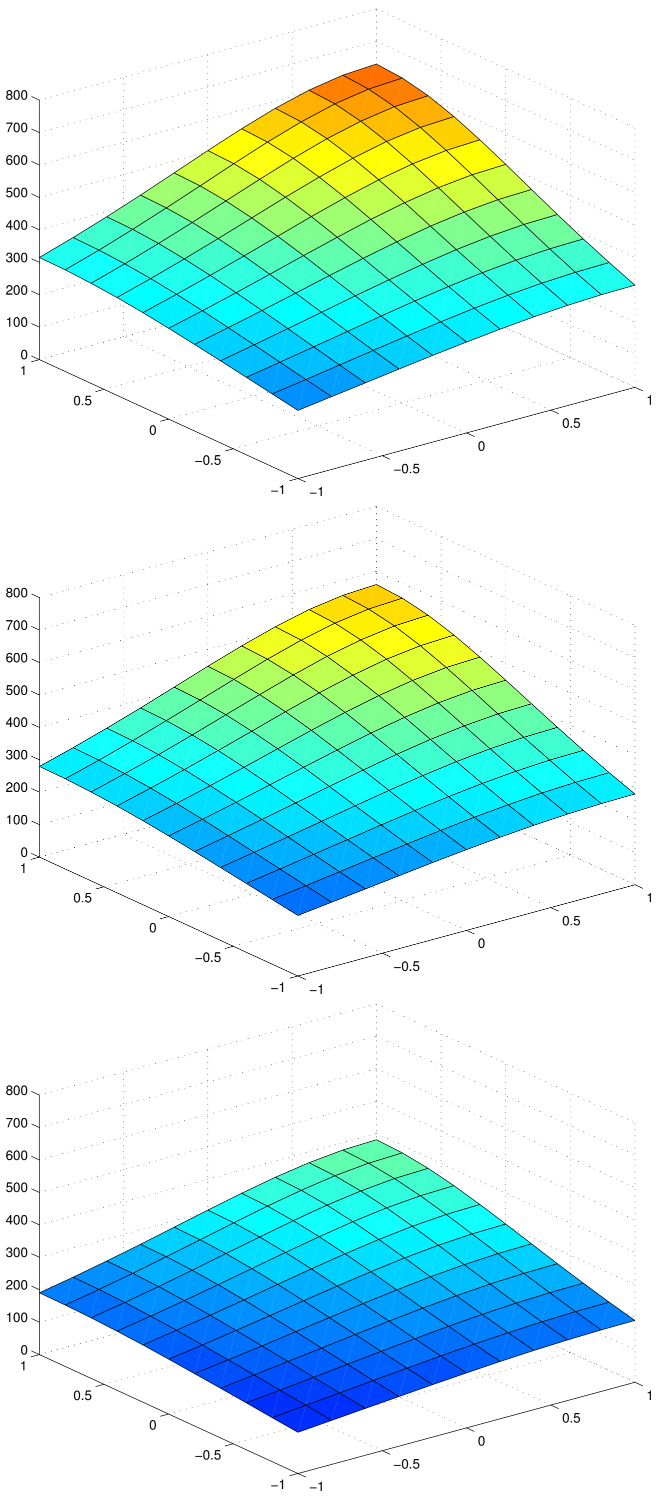


Figure 2: The sensor value function 2 at time $t = 0.0, 0.7$, and 1.4. Note the location of the maximum (at the corner) does not change, but its value does.

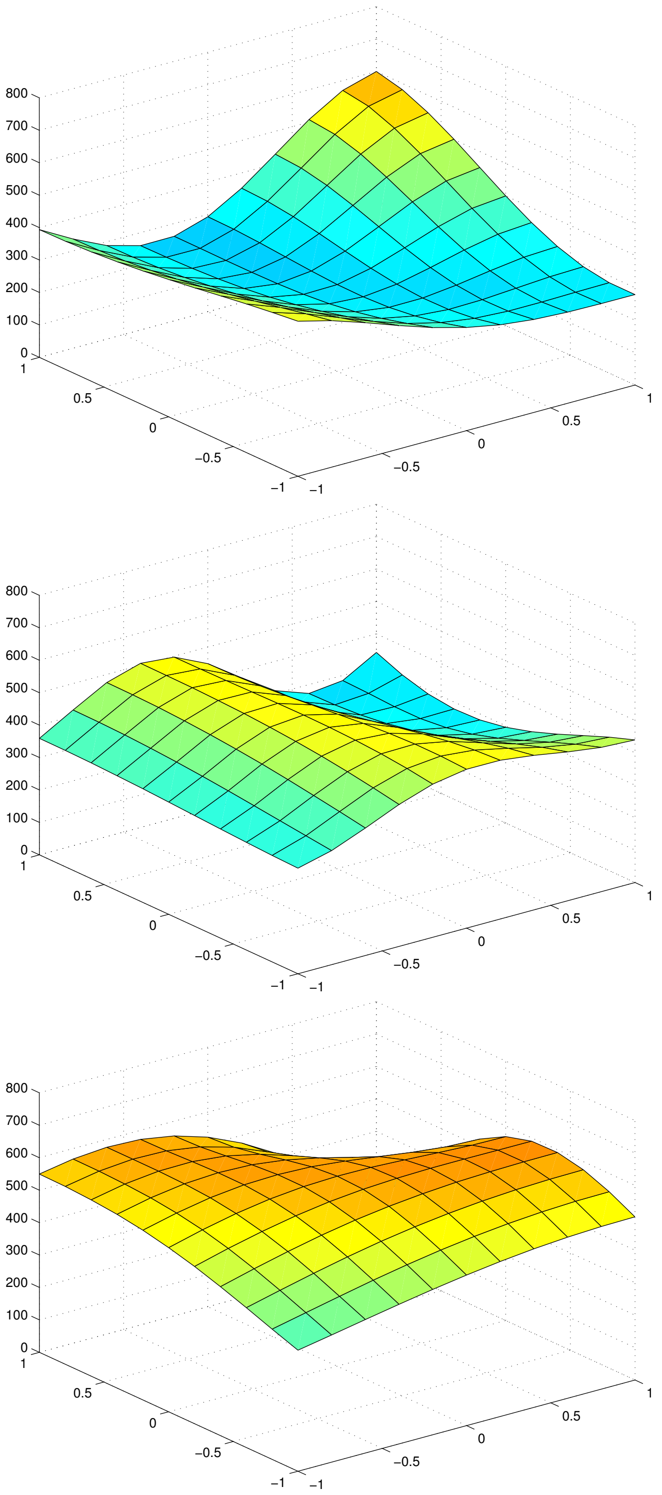


Figure 3: The sensor value function 2 at time $t = 0, 2$, and 4. Note that the location of the maximum changes over time.

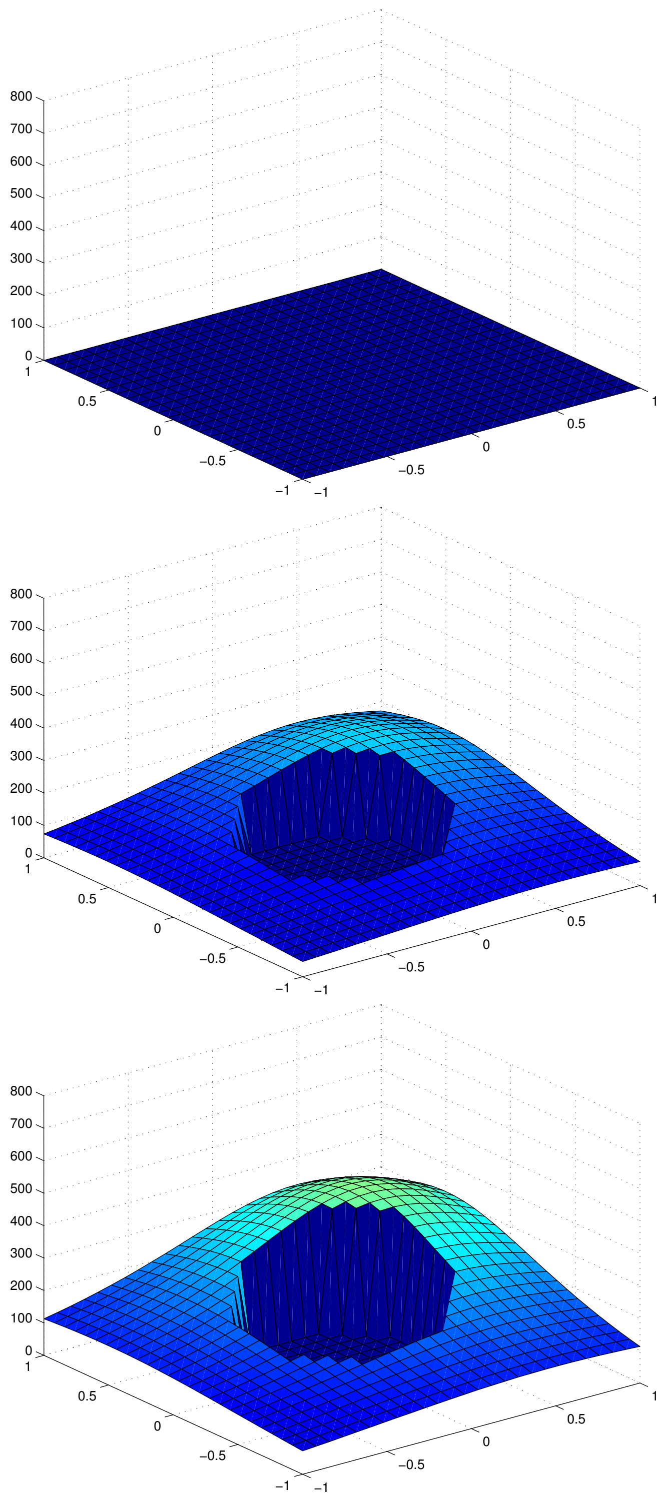


Figure 4: Discontinuous sensor value function at time $t = 0, 0.7$, and 1.4. Note that boundary where the sensor value abruptly changes.

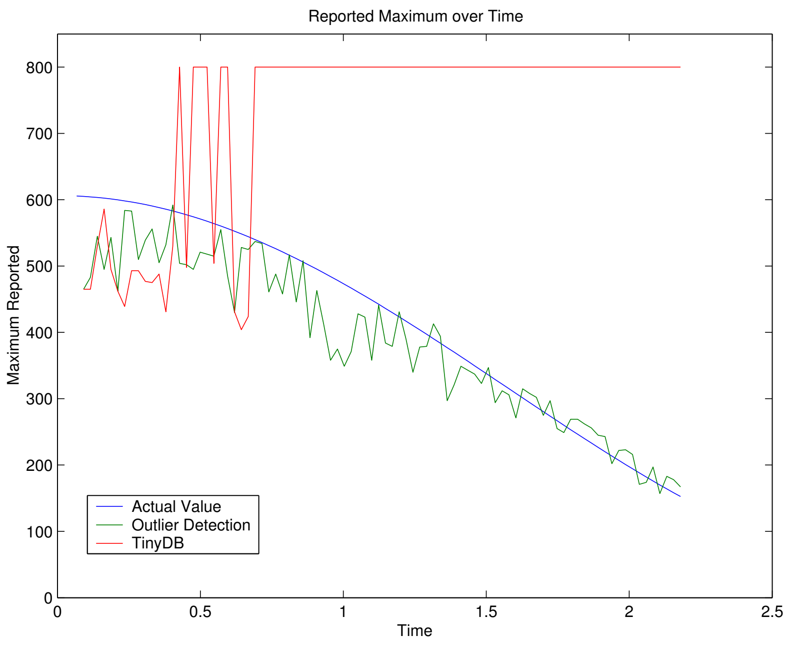


Figure 5: Reported Max vs Time. Node failures occur in the time $0.3 - 0.5$. Note that the original TinyDB reports totally useless values after node failure. With our modifications the output follow the true value relatively closely.

5 Experimental Results

Figure 5 shows the typical behavior when the maximum sensor value is requested from the sensor network. After a few node fails, the original TinyDB application[2] reports totally useless values. With the outlier detection turned on, the aggregate values follows the true value relatively closely.

The reported values does not follow the true value exactly because of three reasons:

1. some nodes will fail to report on time,
2. some packets are lost due to contention, and
3. there is a delay between sensor read and when the value is actually reported.

In figure 6, we plotted the same plot as figure 5, except the aggregation function is average instead of maximum. Here the aggregate result is less affected by the outliers since averages are less affected by faulty sensor readings. Still, we can see that the outlier detection helps being the reported average closer to the true value.

Figure 7 shows the reported maximum on a discontinuous sensor readings (function 4). Since the around the root node all the sensor values are zero, we see that the maximum is correctly reported through a discontinuous boundary.

To test our outlier detection, we setup tests that varies a couple of parameters. One parameter is the percentage of failed nodes. By failed nodes, we mean motes that report back erroneous sensor

[2]With multipath routing implemented but without outlier detection.

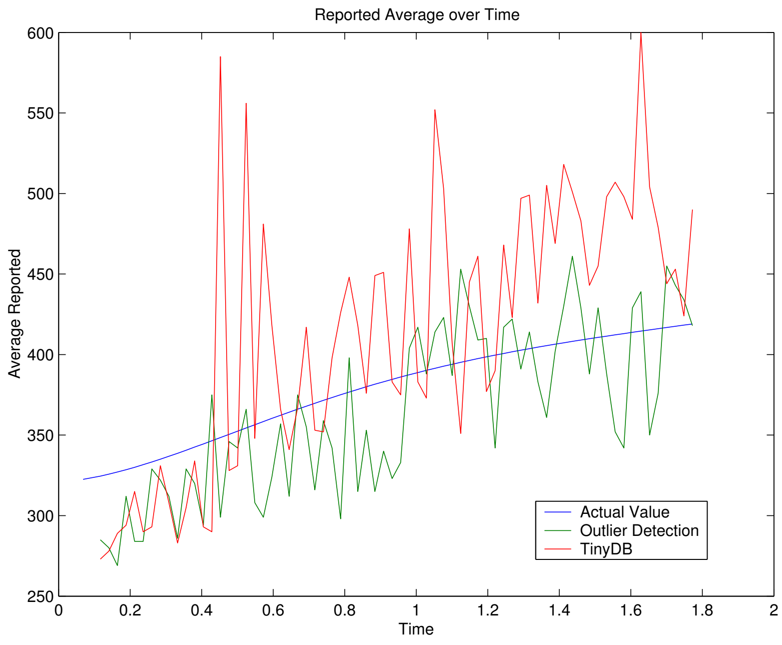


Figure 6: Reported Average vs Time. Here the effect of the outliers is less pronounced than in the case of max, but the effect is still evident.

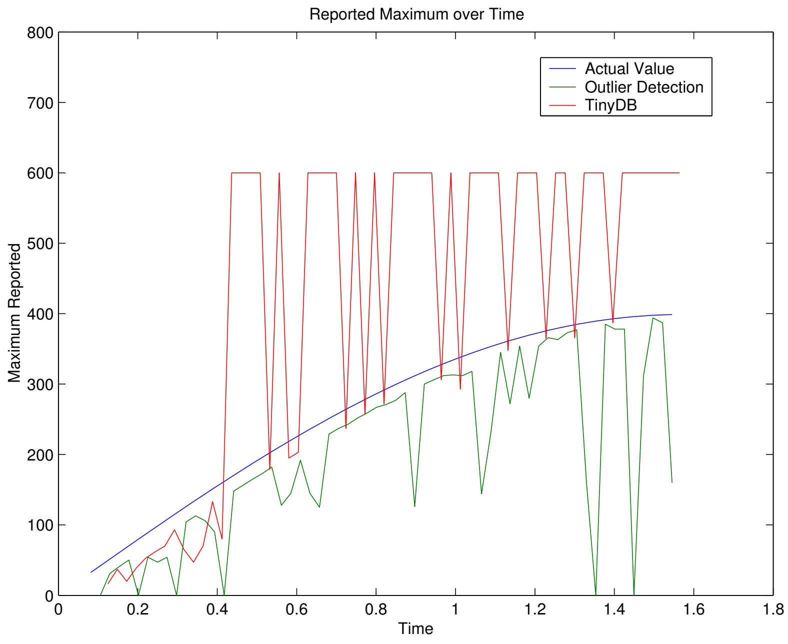


Figure 7: Reported Maximum vs Time. The correct maximum is reported even though there is a sharp discontinuity in the sensor value on the path to the root.

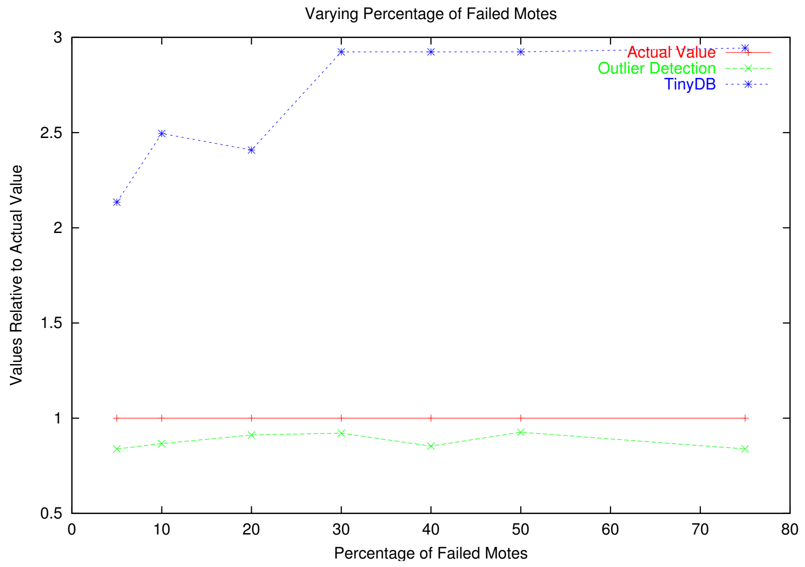


Figure 8: Varying the Percentage of Failed Motes

reading values. Other parameters that we vary are the size of the outlier, node density, aggregation function, and the sensor data model over the sampled region.

The default values for the parameters are as follows:

Percentage of Failed Nodes	10%
Size of Outlier	1500
Node Density	81 Motes
Aggregation Function	Max
Data Model Function	Function 3

For each of the three graphs with varying parameter, we have three data plots. The actual value is known by the function we use to generate the data to supply to the sensor readings. The maximum of the actual values range from 600 to 50, so we take a select portion of epochs whose maximum remains relatively stable, and normalize the actual value to 1.

The TinyDB line is running the model on the TinyDB code without outlier detection. The outlier detection line is the TinyDB code with our outlier detection modifications. In figure 8, the TinyDB code will report back the outlier as is. So when larger percentage of the motes are reporting 1500, the TinyDB aggregation will report 1500, which is about 3 times the actual value. The outlier detection code works well in this setting and marks 1500 as an outlier. The outlier detection line stays just below the actual value because in every epoch, not all the sensor readings make it back to the root. Since we are using model 3, the maximum value moves to different motes, so the root node will not hear from the maximum value mote all of the time.

Figure 9 shows the result of varying outlier size. The original TinyDB just reports the outlier. When the outlier size is only slightly larger than the true maximum (600 vs. 550), our outlier detection cannot distinguish it from the true data and hence the maximum reported becomes slightly higher. Once the size of outlier is large enough, it will be pruned, and the reported result closely follows the true maximum.

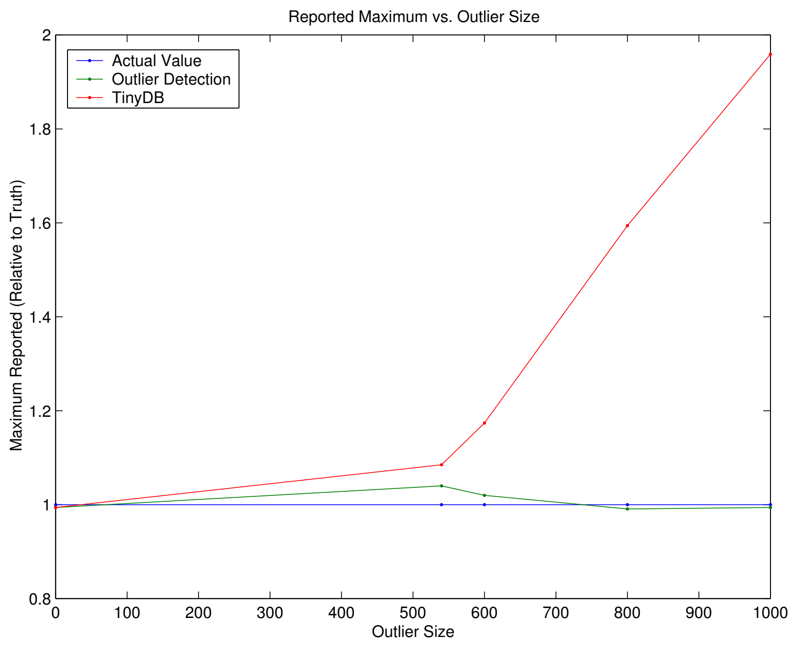


Figure 9: Varying the Outlier Size. Notice when the outlier is at size 600, it is not always detected as an outlier and hence it raises the reported maximum by a small amount.

In the figure 10 we vary, the TinyDB aggregation consistently report the outlying value, while the outlier detection aggregation just reports slightly below the actual value.

During the experiment we noticed several shortcomings of our outlier detection scheme. First, when the nodes are faulty from the very start, it may not be detected as an outlier since at the point we cannot tell (since we have not heard from all the sensors). Figure 8 shows that when the sensor values abruptly changes, it will flag it as an outlier and never allows it to report them even if surrounding nodes eventually reach similar value.

6 Related Work

Recently, there has been some research on security in the sensor networks and behaviors of malicious motes. [18] gives a good overview of various security problems in wireless sensor network, and [1] gives secure routing protocol resilient against malicious nodes. These works do not look at faulty (but non-malicious) sensor nodes affecting aggregation over these sensor networks. Other works on directed diffusion [8, 6] do look at aggregations in these sensor networks, but do not consider malicious motes nor faulty sensor readings. However, they do mention installing a filter to modify the data as it is routed; our work can be interpreted as a case study of this filter in case of outlier detection. In [3], the authors give a very simple outlier detection through selection operators, but do not consider in-network aggregation, and does not consider local information or requires predetermined position of the motes.

There are also several works concerning the detection of spatial outliers in the context of net-

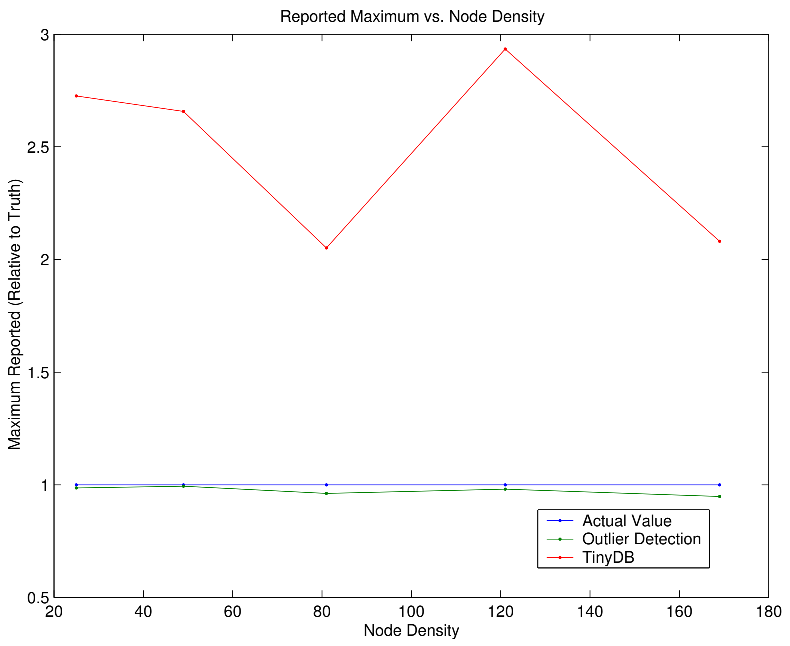


Figure 10: Varying the Density of the Motes

work. We based our algorithm on that of [16]. The algorithm was taken from a data mining application where outliers in graphs had to be found. The method used to detect outliers is the same, but the implementation is different since a central node ran the outlier detection for the data mining case while our algorithm was run inside the network itself. Another algorithm described in [13] works based on identifying a neighborhood from a certain node, and then how far the nodes that lie outside that neighborhood deviate from the nodes at the edge of the neighborhood. The part of this algorithm that was implemented in our algorithm was the user defined deviation. This value allows the user to tell the motes how much variation to expect in the sensor values.

7 Future Work

Currently, the outlier detection only uses a level 1 sphere of influence. A node bases its detection on all the nodes within its 1-hop communication distance. An interesting modification would be to include a second level detection, in which the node would check not only the node it could hear, but the nodes that its children can hear as well. This would enlarge the neighborhood of a particular node and give a much better understanding of how the data should behave.

We can also consider modifying the outlier detection algorithms to look at temporal and spatial trends of the sensor values. In temporal case, this can be accomplished by approximating one or more derivative, and using this to predict the "correct" value for that mote. The statistical test can be then done using these predicted values. Thus if the sensor values are rising, it will likely pass data that is on the projected path.

Similar technique can be used in the spacial dimension if the relative location of the nodes are known. We can compute an approximation to the gradient (and perhaps curvature) at each motes, and use this information to predict what each neighbor should report. These more advanced outlier detection techniques can be found in [2]. However, both of these techniques require further smoothness assumption in the underlying data, and whether this will improve outlier detection remains to be seen.

Finally, there are user interface issues that can be improved. Currently, the user defined threshold standard deviation value are hard coded. Modification to the SQL query is necessary so the user is can specify the expected variation of the data and whether or not to use outlier detection. On a similar note, there may be some application where the user is interested in the outlier itself. In this case, the outlier detection can be used in the reverse sense: if the outlier is detected, report it back to the root so the user is notified and can take action. This may be of use in sensor network used in fire alarms; we are interested in abnormal or unresponsive nodes at the cost of some false alarms. Even in general sensor network, the user may want to know which nodes have failed and in need or repair.

8 Conclusion

We have presented a simple method of dealing with two types of faulty nodes: unresponsive nodes and those reporting faulty sensor values. The unresponsive nodes were bypassed through multipath routing, where the nodes changes the parent when it does not hear from the current parent. Faulty sensor values were dealt with in-network outlier detection, performs a pair of statistical tests of each incoming values with all the nodes within the sphere of influence.

With the assumption of smooth underlying data (in terms of both time and space), we have demonstrated that our outlier detection algorithm filters out outliers and reports the correct aggregated maximum and average. Some limitation were pointed out: presence of outliers at the very beginning and abrupt changes in data are currently not handled well. Finally we pointed out numerous ways to which this outlier detection scheme can be expanded for more reliable outlier detection.

Appendix: Implementation Details

Our implementation can be downloaded from our project page `http://www.cs.berkeley.edu/~yozo/cs252/project.html`. There are two downloads, one for the modified TinyOS tree and another for auxiliary utilities to generate the sensor input files. The implementation consists of three main parts: modification to the TinyOS Nido simulator itself (found in `tinyos/tos/platform/pc` directory) and modification to the TinyDB itself (found in `tinyos/tos/lib/TinyDB` directory), and auxiliary utilities to produce the sensor input files. The modification is based on TinyOS CVS source pulled around April 27, 2003.

Modification to the simulator consists of modifying the radio model to reading a text file to set up sensor placement, connectivity, sensor value function, and which nodes to fail when. To handle this, extra command line switch is added:

```
-i <input>    use the sensor input file <input>
```

To run the simulation, one uses something like,

```
./main.exe -r static -i grid81.txt
```

and start the TinyDB application.

The sensor input file contains information about node placement, node failure time, type of failure, sensor value function, and connectivity between nodes. Main modification here is in the files `rfm_model.c` and `adc_model.c`.

The modification to the TinyDB consists of outlier detection and multipath routing. Outlier detection is implemented as nesC module (`DetectOutlier.nc` and `DetectOutlierIntf.nc`) and called from appropriate places (`TupleRouterM.nc` and `AggOperator.nc`). Multipath routing is implemented in `NetworkC.nc`.

Finally, the auxiliary utilities is a Java GUI program that can display and manipulate the sensor input files. They can be compiled by running `make` in the `util` directory, and run with `java SensorDisplay`. It can produce random, rectangular grid, and hexagonal grid node layout. Connectivity is determined by the radius which can be adjusted. Number of nodes to fail and the function to be used can be specified as well. The format of the sensor file is described in `doc.txt`.

References

[1] Baruch Awerbuch, David Holmer, Cristina Nita-Rotaru, and Herbert Rubens, *An on-demand secure routing protocol resilient to byzantine failures*, ACM Workshop on Wireless Security (WiSe) (Atlanta, Georgia), September 2002.

[2] Vic Barnett and Toby Lewis, *Outliers in statistical data*, Wiley, New York, 1984.

[3] Phillipe Bonnet, J. E. Gehrke, and Praveen Seshadri, *Towards sensor database systems.*, Proceedings of the Second International Conference on Mobile Data Management (Hong Kong), January 2001.

[4] Stefan Dulman, Tim Nieberg, Paul Havinga, and Pieter Hartel, *Multipath routing for data dissemination in energy efficient sensor networks*, Tech. Report TR-CTIT-02-20, Center for Telematics and Information Technology (CTIT), July 2002.

[5] Deepak Ganesan, Ramesh Govindan, Scott Shenker, and Deborah Estrin, *Highly-resilient, energy-efficient multipath routing in wireless sensor networks*, Mobile Computing and Communications Review **1** (2002), no. 2.

[6] John S. Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan, *Building efficient wireless sensor networks with low-level naming*, Symposium on Operating Systems Principles, 2001, pp. 146–159.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister, *System architecture directions for networked sensors*, Architectural Support for Programming Languages and Operating Systems, 2000, pp. 93–104.

[8] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann, *Impact of network density on data aggregation in wireless sensor networks*, Tech. Report 01-750, University of Southern California, November 2001.

[9] David B Johnson and David A Maltz, *Dynamic source routing in ad hoc wireless networks*, Mobile Computing (Imielinski and Korth, eds.), vol. 353, Kluwer Academic Publishers, 1996.

[10] Sam Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, *Tag: A tiny aggregation service for ad hoc sensor networks*, OSDI, December 2002.

[11] Samuel Madden, Robert Szewczyk, Michael J. Franklin, and David Culler, *Supporting aggregate queries over ad-hoc wireless sensor networks*, Workshop on Mobile Computing and Systems Applications, 2002.

[12] A. Nasipuri and S. Das, *On-demand multipath routing for mobile ad hoc networks*, October 1999, pp. 64–70.

[13] Spiros Papadimitriou, Hiro Kitawaga, Phillip B. Gibbons, and Christos Faloutsos, *Loci: Fast outlier detection using the local correlation integral*, Tech. Report IRP-TR-02-09, Intel Research Laboratory, Pittsburgh, July 2002.

[14] Vincent D. Park and M. Scott Corson, *A highly adaptive distributed routing algorithm for mobile wireless networks*, INFOCOM (3), 1997, pp. 1405–1413.

[15] Kay Römer, *Temporal message ordering in wireless sensor networks*, Submitted for publication, December 2002. Available at `http://www.inf.ethz.ch/vs/publ/papers/temporder.ps`.

[16] Shashi Shekhar, Chang-Tien Lu, and Pusheng Zhang, *Detecting graph-based spatial outliers*, Intelligent Data Analysis **6** (2002), no. 5, 451–468.

[17] *TinyOS*, `http://webs.cs.berkeley.edu/tos/`.

[18] Anthony D. Wood and John A. Stankovic, *Denial of service in sensor networks*, IEEE Computer **35** (2002), no. 10, 54–62.