Work in Progress: Resource-Aware Fault Localization in Large Sensor Networks

Richard Mietz, Kay Römer
Institute of Computer Engineering, University of Lübeck, 23562 Lübeck, Germany
Email: {mietz,roemer}@iti.uni-luebeck.de

***Abstract*—Sensor networks are exposed to hostile environments that may cause failures of single nodes and communication links which affect the whole network. Localizing the cause of the problem in space and time requires to collect diagnostic data from the network. Due to resource and energy constraints, however, it is not possible to continuously collect detailed diagnostic data from all nodes. We therefore propose an incremental approach where first data is logged to flash memory and later the user can pose a sequence of diagnostic queries with decreasing scope and increasing level of detail to pinpoint the cause of the problem.**

I. Introduction

Sensor networks are often exposed to unpredictable and hostile environments that lead to performance problems or even (partial) failures of individual nodes or communication links. Such localized problems, however, often have an impact on the whole network and it is typically very difficult to localize the cause (i.e., nodes and/or links) of the problem. For example, packet drops or delays due to a *single* bad link in a collection tree may lead to lost or delayed packets from *all* nodes in the subtree of the bad link. As the topology of the collection tree is dynamically changing, it is difficult to identify the bad link. Often it is even difficult to identify the point in time when the cause of the problem occurred. For example, in a network that detects events, the lack of event notification messages might be caused by the absence of events or by a failure in the network leading to loss of the notification message.

Localizing the cause of the problem in space (i.e., which nodes?) and in time (i.e., when did it occur?) requires *visibility* into the state of the network by collecting diagnostic information in addition to the actual sensor data. However, given the limited resources and energy, it is often not feasible to continuously collect detailed diagnostic data from all nodes in the network.

We therefore propose an incremental approach to fault localization, where a user poses a sequence of diagnostic queries with decreasing scope and increasing level of detail as illustrated in Fig. 1. Here, scope refers to the set of nodes that might be the cause of the problem and a time frame during which that cause appeared. Initially, the user might start with a scope that includes all nodes in the network and a timeframe of the week during which the problem was first noticed. However, only highly aggregated information can be collected from each node (e.g., the maximum forwarding delay at each node during the whole week) as otherwise a huge amount of data would have to be extracted from the network due to the large scope. From the aggregated data the user can conclude which nodes have a large maximum delay and pose another query to these nodes (i.e., reduced spatial scope) to obtain maximum delays for each day in the week (i.e., increased level of detail). From the results the user may conclude on which days the problem occurred and request hourly maximums (i.e., increased level of detail) only for those days (i.e., reduced temporal scope) and so on until the problem has been tracked down to a scope that is small enough.

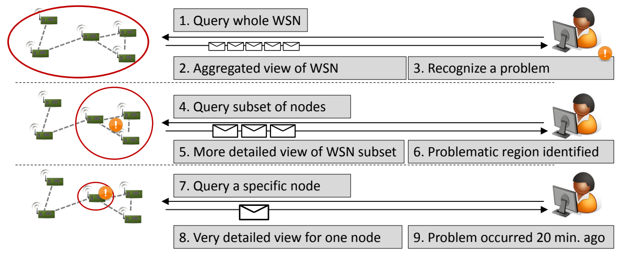
1. Query whole WSN
2. Aggregated view of WSN
3. Recognize a problem
4. Query subset of nodes
5. More detailed view of WSN subset
6. Problematic region identified
7. Query a specific node
8. Very detailed view for one node
9. Problem occurred 20 min. ago

Fig. 1: The user poses a sequence of diagnostic queries.

In this paper we report about our ongoing work to design and implement a system to support such incremental diagnostic queries. The use of the system can be divided into three phases. In the *initialization phase*, the user sends a logging specification to the network which specifies data (e.g., the forwarding latency of certain types of messages or battery voltage) that should be logged into flash using a ring buffer of a user-defined size. During the *collection phase*, all nodes continuously log the requested data locally. Note that recent flash technologies are very energy efficient. Finally, during the *diagnosis phase* the user poses incremental diagnostic queries as described above to extract aggregated versions of the data collected in flash from a given set of node. Each query contains a scope and a resource budget (e.g., amount of data, amount of energy, lifetime reduction) the user is willing to spend for the execution of this query. To implement the actual data reduction and aggregation on the nodes, the user also specifies a set of LODs (level of detail), where each LOD is a user-defined data aggregation/reduction function that compresses the previously collected data. The system then automatically selects the appropriate LOD for a query to meet the user-defined budget.

II. Resource-Aware Diagnostic Queries

We briefly describe each of the three phases outlined above along with the different specifications used during each phase.

A. Initialization and Collection Phases

Initially, the users specifies what to log, on which nodes, when to log, as well as the size of a ring buffer in flash memory to use for logging. Multiple logging jobs can be defined for different sets of nodes. The logging specification is then compiled into a compact byte code and sent to all affected nodes in the network, where an interpreter executes the specification in order to log the requested data.

Listing 1: Logging specification for aspect *Energy* and *Latency* written in YAML

```
 1  ---
 2  #Log energy of every node every 60 seconds with a budget of 10 kb of memory
 3  all:
 4    Energy:
 5        budget amount: 10
 6        interval: 60
 7  #Log lantency of nodes 99f8 and 98dd with their neighbors with 20 kb of memory
 8  99f8,98dd:
 9    Latency:
10        budget amount: 20
11  ...
```

Listing 1 shows a *logging specification* written in YAML (YAML Ain't Markup Language). The user defines several logging jobs by selecting either all nodes or a list of nodes (cf. line 3 and 8), the aspects to log (cf. line 4 and 9), and a budget in terms of number of kilobytes of flash memory (cf. line 5 and 10) a node is allowed to use for logging. At the moment our system supports logging remaining energy, message forwarding latency, as well as message drops. When logging the remaining energy, the user has to additionally define a logging interval (cf. line 6). Data for latency is logged whenever the ”normal” application running on the node sends respectively receives a message.

In the *collection phase* the nodes log a timestamp and the remaining energy every 60 seconds. For each transmitted message sender and receiver log the timestamp, address and a hash of the message. Hence, the given log budget has to be divided among the specified node and all of its neighbors which he communicates with. Later, when needed, the logs of the neighbors are transferred to the main node, are compared, and the latency is calculated by subtracting the timestamps of messages with the same hash.

B. Diagnosis Phase

In the diagnosis phase, the user poses a sequence of diagnostic queries with decreasing scopes and increasing detail. In addition, each query specifies a resource budget that the system can use to execute the query. In order to meet the specified budget, the system aggregates and reduces that data collected in flash memory. For this, the user first has to specify a set of LODs, each of which consists of a set of functions to filter or aggregate the collected data.

The data collected in flash has the structure of a table with one column for each collected aspect (e.g., timestamp, remaining energy) and one row for each logging entry. Table I (a) shows an example for energy logs with columns for the timestamp and the energy value. The functions defined by a LOD transform this table by deletion of columns, sorting columns, filtering rows by different criteria, different aggregations over columns (e.g., count, min, max, average, histograms), mapping a set of values to a single value, and reducing the precision of values. A LOD is an atomic operation on the table, i.e., all functions of a LOD are executed on the table, producing a new output table. The LODs have to be arranged in such a way that the output table of LOD x is the input table for LOD $x-1$. Thus, the logging node can reduce the LOD of the table and hence, the number of bits needed to encode the table LOD by LOD. Furthermore, every node on the route to the sink is able to further reduce the LOD of the table received from another node if necessary.

Listing 2: LOD description with five levels

```
 1  ---
 2  Energy:
 3    #Data or structure is not modified
 4    4:
 5      - Original
 6    #Reduce the precision of the timestamp containing column to 24 bits
 7    3:
 8      - Precision: [time,24]
 9    #Keep only low energy values and remove the time column
10    2:
11      - Filter: [value,'<',100000]
12      - Delete: [time]
13    #Map energy values and reduce precision of that column to 2 bit per value
14    1:
15      - Map: [[1:[0,1000], 2:[1000,10000], 3:[10000,.inf]], value]
16      - Precision: [value,2]
17    #Only count number of rows (i.e. number of energy values lower than 100000]
18    0:
19      - Count
20  ...
```

A sample *LOD specification* is given in Listing 2. Five LODs are defined (lines 4, 7, 10, 14, and 18). On the highest LOD (lines 4 and 5) the data is kept as stored in flash, i.e., without any function applied to it. The next LOD reduces the precision of the timestamps from 32 to 24 bit by setting the least significant bits to 0 such that they do not have to be transmitted. LOD 2 has two functions. First, only energy values smaller than the given threshold are kept and secondly, the timestamp column is deleted. The map function in LOD 3 maps energy values to three different categories: values in the range 0-1000 are mapped to category 1, values in range 1001-10000 are mapped to 2 and so on. The precision is then reduced to 2 bits to encode the three categories. The final LOD computes the number of remaining rows in the table, i.e., the output table consists of a single column with one row holding the count. Again, the specification is compiled to a compact byte code and sent to the network, where an interpreter uses the specification to execute queries.

Listing 3: Query for node 99f8 to deliver energy with at LOD 1

```
1  ---
2  #Get energy monitoring data of node 997f on level 1
3  99f8:
4    Energy:
5        level: 1
6  ...
```

Timestamp (32)	Energy (32)
1317889870	11000000
1317889930	11000000
1317889990	11000000
⋮	⋮
1317975990	99000
⋮	⋮
1317983190	8000
1317983250	7600

(a) LOD 4: 1702 bits

→

Timestamp (24)	Energy (32)
1317889792	11000000
1317889920	11000000
1317889920	11000000
⋮	⋮
1317975936	99000
⋮	⋮
1317983104	8000
1317983232	7600

(b) LOD 3: 1494 bits

→

Energy (32)
99000
⋮
8000
7600

(c) LOD 2: 862 bits

→

Energy (2)
3
⋮
2
2

(d) LOD 1: 82 bits

→

Energy (2)
13

(e) LOD 0: 32 bits

TABLE I: Size reduction of monitoring data by applying different LODs.

Each such query is encoded in a specification as shown in Listing 3. Besides the involved nodes (line 3) and the aspect (line 4), the LOD (line 5) needs to be given to issue a query. The addressed nodes read their data from their flash and execute LOD after LOD until the desired one is reached. The query specification is also compiled to byte code and sent to the query interpreter executing on each node.

III. Next steps

In the final version of the system the user would not directly specify the LOD for a query, but the budget he is willing to spent to execute the query. The network will translate the given budget to the highest LOD that does not exceed that budget. In a first step the user should be able to define the budget in terms of network bandwidth, but later on it should be possible to define it in terms of lifetime reduction of the network.

Apart form the implementation of the WSN components, we are working on a graphical user interface (GUI) to support the user by providing wizards to create and edit the necessary specifications. The sensor network, query scopes, and the received data will be visualized by the GUI to easily inspect the state of nodes and the complete network, also in the field.

IV. Preliminary Evaluation

As our focus is on resource-awareness, we illustrate the data reduction that can be achieved using the example specifications given in this paper. Firstly, we show by how much byte code compilation can reduce the size of specifications that need to be sent to the network. Secondly, we show how different LODs reduce the encoded size of logging data collected by a node.

Table II shows the original (comments removed) and compiled size of the logging, LOD, and query specifications from Sect. II in bits as well as the savings. As one can see, we save around 90 % of bits for each specification.

Specification	Original size (UTF-8)	Compiled size	Savings in %
Logging	976	78	92.0
Level	2056	207	89.9
Monitoring	376	28	82.5

TABLE II: Size reduction of specifications by compilation.

To evaluate the reduction of monitoring data by a LOD specification, we use the raw data collected by a sensor node shown in Table I (a), which contains 26 rows (each vertical dot represent 10 rows of data) of logged energy data. The numbers in the column header depict the number of bits used to encode a value in that column. We use the *LOD specification* from Listing 2 to reduce the number of bits needed to encode the table. In section II we described already how the tables are altered on each LOD. The number of bits needed to encode these five tables are shown in the subcaptions of Table I. It can be seen that a lower LOD needs less bits to encode. The table for LOD 0 compared to LOD 5 saves 92.1% of bits. However, expressiveness of the table is also drastically reduced. Nevertheless, a table on a low LOD can still give useful hints on a potential problem. In the example, the user can see that there are already some nodes with energy values below a certain threshold indicating that these nodes may die soon.

V. Related Work

In TinyDB [1] the WSN acts as a database which can be queried by the user using an SQL-like query language to retrieve sensor readings or data from previously installed logging queries. Queries are optimized before dissemination to the network to save resources. However, there is no support for specifying a resource budget for a query.

The work described in this paper builds on our previous work on Visibility levels [2], a monitoring system that allows to manage the trade-off between the visibility of node state and resource consumption. By annotating the source code, program variables can be logged at execution time. Additionally, the user defines compression schemes similar to our *level specification* and a resource budget for storing the logged data. The annotated code is compiled to executable code which automatically compresses data if needed to not exceed the given budgets. In contrast, our current work addresses performance debugging at the network level. Also, we avoid the need to recompile and upload application source code when a specification is changed.

VI. Conclusion

We presented an approach for resource-aware performance debugging of resource-constrained WSNs, where the user can pose diagnostic queries with decreasing scopes and increasing detail, offering the user full control over the resources used to execute these queries.

Acknowledgment

This work was funded by the Federal Ministry of Education and Research of the Federal Republic of Germany (Förderkennzeichen 01BK0905, GLab). The authors alone are responsible for the content of the paper.

References

[1] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “Tinydb: an acquisitional query processing system for sensor networks,” *ACM Trans. Database Syst.*, 2005.

[2] J. Ma and K. Römer, “Visibility levels: managing the trade off between visibility and resource consumption,” in *Proceedings of the 4th international conference on Real-world wireless sensor networks*, 2010.