Detecting Reporting Anomalies in Streaming Sensing Systems

Shiree Hughes[1], Yuheng Du[2], and Jason O. Hallstrom[1]
[1]I-SENSE, Florida Atlantic University, {shughes2015, jhallstrom}@fau.edu
[2]School of Computing, Clemson University, yuhengd@clemson.edu

Abstract

Sensor networks must be monitored to identify and correct problems as they occur. We present a comparison of two approaches to monitoring deployed sensors. The first relies on configuration parameters to define expected reporting behavior. The second automatically identifies normal reporting patterns based on a combination of configuration parameters and an analysis of reporting times. Using these patterns, the system notifies personnel of possible malfunctions. We present empirical evaluations in the context of the Intelligent River® system [11].

1. Introduction

Wireless sensor networks are often used to collect data over large geographic areas. Environmental factors, ranging from changes in cellular or satellite signal strength to equipment damage can disrupt data collection. It is important to identify when sensors are malfunctioning to avoid prolonged data loss. Equally important is the ability to ignore routine or minor fluctuations in reporting behaviors to avoid spurious maintenance notifications.

We present two approaches to monitoring the behavior of deployed sensors. The *static* approach is based on user configuration data that captures the expected reporting period of each sensor. The *adaptive* approach relies on an online time-series analysis of reporting times. The ability to automatically identify reporting patterns enables the determination of whether a sensor is more likely to be damaged, versus exhibiting normal variation in reporting behavior.

We evaluate the performance of both approaches using historical data from 122 devices reporting at various rates over a six-month period within the Intelligent River® network. Many of these devices are enclosed in bouys designed to keep them afloat and protect them from the water. We explore the effects of various parameters on the sensitivity of the monitoring system.

2. Related Work

Many authors have considered methods to identify sensor malfunction and proposed various solutions [5, 6, 7, 13, 17], most concerned with the validation of received data.

DOI reference number: 10.18293/SEKE2015-181

Ramanathan et al. [10] present Sympathy, a tool to detect and debug sensor failures. Similar to our static approach, Sympathy assumes an expected time period in which a sensor should report, and then flags the sensor if it does not report within the expected time. Sympathy requires each sensor to transmit a set of metrics, including information on network connectivity, packet reception rate, and packet corruption rate. When a failure is detected, the metrics are analyzed to determine cause and location. Sympathy does not adapt to periodic network delays and will repeatedly send notifications during periods of abnormal network behavior.

Shebaro et al. [12] focus on determining the cause of packet loss. Their method employs comparisons of RSSI and LQI values. While link quality can be a good indicator of malfunction, information detailing when sensors are expected to report is also needed to diagnose sensor failure.

Duche and Sarwade [2] use round-trip delay (RTD) to detect malfunction in multi-hop scenarios. When a sensor fails to transmit, the RTD of the sensor approaches infinity. By defining an initial threshold for comparison, a malfunctioning sensor can be identified if its RTD is greater than the threshold. Each sensor has several paths it may transmit over, and each path may have a different RTD. Their approach assumes a circular topology.

Guo et al. [4] concentrate on the detection of faulty data. Their approach considers both sensor data and location and assumes that all sensors observe and report data on the same event. Sensors are designated as faulty if they do not adhere to distance monotonicity. The goal is to create a list of sensors reporting faulty data ranked by probability, not to notify personnel of sensors' failure to report.

Warriach et al. [14] propose a hybrid fault detection approach that combines rule-based, linear least-squares estimation and hidden Markov methods to identify data faults. The approach does not detect the absence of data.

Bartzoudis and McDonald-Maier [1] describe a method to validate sensor readings. Their approach compares prior readings to current readings to identify improbable discrepancies, such as drastic temperature changes in a short time period. Reported values are also verified to be within the operating limits specified by the manufacturer. The purpose is to identify data faults, not to identify reporting failures.

Zang et al. [16] use a combination of principal component analysis and wavelet analysis to detect sensor failure

in small and medium-scale networks. Detection of a malfunctioning sensor is dependent upon data collected from neighboring sensors. Our goal is to identify malfunctioning sensors independent of other sensors within a network.

Friend et al. [3] are concerned with commercial credit-card fraud. We take inspiration from methods of fraud detection in this context to identify malfunctioning sensors. Like people, each sensor has an identifiable reporting pattern which can be used to detect changes in behavior. Commercial fraud detection is achieved by comparing current observations with expected values derived from previous observations. Friend et al. find that a customer's transactions can be represented by a standard bell curve. The average value of a customer's transactions defines the peak of the curve. Each standard deviation away from the average contains transactions that are more likely to be fraudulent. The authors find that only 1% of transactions are more than 3 times the standard deviation. We use this principle and apply it to sensor reporting periods in the adaptive approach to determine sensor malfunction.

3. The Static Approach

We now detail our implementation of the static approach.

3.1. Configuration

Configuration data is specified using JSON data stored in MongoDB. Sensors are separated into groups based on deployment area, and each group is represented as a JSON object, such as the example shown in Listing 1. Six properties are defined for each group: `groupID` is a unique group identifier. `members` specifies a list of sensors included in the group, each identified by ID. `expectedInterval` specifies the maximum number of seconds that may pass between reports from sensors within the group. `notificationTime` specifies the number of seconds to wait between personnel notifications while a sensor is malfunctioning. `maxNotifications` specifies the maximum number of notifications that should be sent to network personnel while a sensor is malfunctioning. The configuration data also stores *notification* group objects, which define personnel contact information. A sample notification group object is shown in Listing 2. The `notificationGroupID` in Listing 1 specifies the identifier corresponding to the notification group defined in Listing 2, declaring the list of personnel to be contacted if a malfunction is identified within the sensor group.

```
1 {"GroupID":"Sensor-Group-1",
2  "members":["sensor-1",
3             "sensor-2",
4             "sensor-3"],
5  "notificationGroupID":"Notification-Group-1",
6  "expectedInterval":900,
7  "notificationTime":1500,
8  "maxNotifications":5 }
```

Listing 1. Sample Sensor Group

```
1 {"notificationGroupID":"Notification-Group-1",
2  "addresses":["user1@domain.com",
3               "user2@domain.com"] }
```

Listing 2. Sample Notification Group

3.2. Implementation

The monitoring service was developed as part of the Intelligent River® system [15], a network of sensing devices deployed throughout the Savannah River Basin to support water management and agricultural applications [11]. Each sensor transmits environmental data through the network using RabbitMQ [9], an open source implementation of the AMQP standard [8]. In the Intelligent River® system, RabbitMQ receives data reports routed through dedicated queues. The sensor monitoring service uses one of these queues to obtain device readings, and to monitor time-stamps. The implementation is developed in Java and consists of three main classes: `MessageHandler`, `StatusChecker`, and `Notifier`.

3.2.1. MessageHandler

The `MessageHandler` class consumes messages from a dedicated message queue. Each message is parsed to determine the identity of the reporting device and the time-stamp of the report. Instances of the class maintain a local hashmap, mapping from sensor ID to a data object containing the time-stamp of the most recent report, the time-stamp of the most recent notification, the number of notifications sent, and the current status of the sensor (i.e. alive or dead).

3.2.2. StatusChecker

The `StatusChecker` class determines if a sensor's status should be updated. At startup, a `StatusChecker` thread is created for each sensor group identified in the input configuration. Listing 3 summarizes the core logic used in each thread. `StatusChecker` wakes every `expectedInterval` seconds (line 2) and queries `MessageHandler` for the latest time-stamp associated with each sensor within the group (lines 3–4). The system then calculates the most recent reporting interval, `timeDifference` (line 5). If the reporting interval is not within the `expectedInterval`, the sensor's status is set to dead (lines 6–7). After marking a sensor as dead, `StatusChecker` determines the interval between the last time of notification and the current system time (line 8). If the interval is more than `notificationTime` and the number of notifications sent is less than `maxNotifications`, a `Notifier` instance is created to handle notification (lines 9–10).

`StatusChecker` also detects revivals; i.e. sensors marked as dead which begin to report. When

```
1  while(true) {
2    wait(expectedInterval);
3    for each sensorID in members {
4      sensor = hashmap.get(sensorID);
5      timeDifference = currentTime-sensor.lastReportTime;
6      if(timeDifference>expectedInterval) {
7        sensor.status(dead);
8        notifyInterval = currentTime-sensor.notifyTime;
9        if(notifyInterval < notificationTime &&
             sensor.notificationsSent<maxNotifications) {
10         new Notifier(sensor);
11         sensor.notificationsSent++;
12         sensor.notifyTime = currentTime;       }
13     } else if (sensor.status == dead) {
14       sensor.reset();
15       new Notifier(sensor);     }    }     }
```

Listing 3. `StatusChecker` Algorithm

`StatusChecker` identifies such a sensor, the sensor's status, notification count, and time of last notification are reset, and a notification is generated (lines 13–15).

3.2.3. Notifier

A `Notifier` instance is created to notify personnel of node malfunctions and revivals. The sensor's data object is passed to the object at construction. The `Notifier` object then retrieves the `addresses` within the notification group identified by `notificationGroupID` and composes a message containing the sensors' status, identifier, and time of last report. Malfunction notifications also contain the notification count to remind network personnel of how many messages they have already received.

4. The Adaptive Approach

Although each sensor is programmed to transmit at a specific interval, environmental and other factors introduce regular fluctuations in observed reporting behavior. In Intelligent River® deployments, wind, rain, dam discharge events, and other factors can cause bouys to drop below the water surface, preventing the enclosed sensors from transmitting via cellular or satellite. We have observed that in most locations, changes in transmission intervals due to temporary environmental factors follow regular patterns.

Since each sensor deviates from its programmed reporting interval differently, it is difficult to set a uniform maximum time interval that encompasses the behavior of all sensors. It would also be tedious to manually update the expected behavior of every sensor within the network. To avoid excessive notifications without requiring significant effort from network administrators, we developed an approach that detects normal variance and estimates a best-fit interval for each sensor within a given group.

4.1. Configuration

A sample configuration showing the definition of a sensor group in the adaptive approach is shown in Listing 4. We continue to use the parameters from the static approach, with three additions. `numberOfStdDevs` represents the number of standard deviations from the average reporting interval considered to be an acceptable deviation. `windowSize` represents the number of previous reporting intervals to be considered when calculating metrics such as standard deviation and mean. `decayConstant` represents how quickly a sensor is expected to return to normal reporting behavior after experiencing failure.

```
1  {"GroupID":"Sensor-Group-1",
2   "members":["sensor-1",
3              "sensor-2",
4              "sensor-3"],
5   "notificationGroupID":"Notification-Group-1",
6   "expectedInterval":900,
7   "notificationTime":1500,
8   "maxNotifications":5,
9   "numberOfStdDevs":3,
10  "windowSize":25,
11  "decayConstant":12 }
```

Listing 4. Sample Sensor Group (adaptive)

4.2. Implementation

In the *adaptive* implementation, `StatusChecker` is modified, and an `Analyzer` class is added.

4.2.1. Analyzer

Each time a sensor reports, an instance of the `Analyzer` class calculates the interval between the two most recent reports and stores the interval in a circular buffer. The size of the buffer is determined by `windowSize`. Each sensor has a dedicated `Analyzer` object stored in the hashmap discussed in Section 3.2.1. The object maintains the buffer and provides mean and standard deviation methods.

4.2.2. StatusChecker

In the adaptive approach, we define three sensor states: (i) *initial*, indicating that sufficient data to accurately predict the sensor's behavior has not yet been collected; (ii) *normal*, indicating that the sensor is reporting within the expected range; and (iii) *abnormal*, indicating that the sensor is reporting outside its expected range.

As in the static approach, each `StatusChecker` thread wakes every `expectedInterval` seconds to check the status of the sensors within its associated group. A new `check()` method is invoked at wake-up to determine the status of the sensor, as discussed in the next paragraphs. Each time the `MessageHandler` object receives a report from a sensor, the associated `StatusChecker` updates its timer, its `Analyzer` object, and its last timestamp. This is handled in the `update()` function of `MessageHandler`. As in the static approach, if a sensor is identified as dead or revived, the system takes the necessary notification steps. Listing 5 shows the `update()` method for the adaptive version of `StatusChecker`. `timeDifference` represents the observed interval between consecutive sensor reports (line 1). `average` represents the average of the intervals stored in the sensor object's buffer (line 3). `stddev` represents the standard devi-

```
1  timeDifference = currentReport - sensor.lastReport;
2  sensor.addToBuffer(timeDifference);
3  average = Analyzer.getAverage(sensor);
4  stddev = Analyzer.getStdDev(sensor);
5  allowable = average + stddev*numberOfStdDevs;
6  switch(sensor.state) {
7    case(INITIAL):
8      sensor.resetTimer(configExpectedInterval);
9      break;
10   case(NORMAL):
11     if(timeDifference>allowable) {
12       sensor.state = ABNORMAL;
13       sensor.expectedInterval = timeDifference;
14       sensor.resetTimer(timeDifference);
15     } else { sensor.resetTimer(allowable); }
16     break;
17   case(ABNORMAL):
18     if(timeDifference<sensor.expectedInterval){
19       sensor.expectedInterval -= (
              (sensor.expectedInterval-allowable) /
             decayConstant);
20       if(sensor.expectedInterval < allowable) {
21         sensor.state = NORMAL;
22       }
23     } else { sensor.expectedInterval = timeDifference;}
24     sensor.resetTimer(sensor.expectedInterval);
25     break;
26 }
27 sensor.lastReport = currentReport;
28 if(sensor.status == dead) {
29   sensor.status == alive;
30   new Notifier(sensor); }
```

Listing 5. `StatusChecker update()` **Method**

ation of these intervals (line 4). `allowable` represents the allowable maximum interval between normal sensor reports (line 5).

***Initial* State**. Initially, the `Analyzer`'s buffer is empty. Before the buffer contains the required number of intervals, the average and standard deviation may not reflect the longer-term values and may fluctuate significantly. In this state, the system expects to receive the next report within the period specified in the configuration (line 8). Regardless of state, the time-stamp of the most recent report is updated to reflect the new time-stamp (line 27). Finally, `update()` checks if the reporting sensor is flagged as dead and notifies network personnel of revivals (lines 28–30).

Listing 6 shows the logic for `check()` in the *adaptive* version of `StatusChecker`. On wake-up in the *initial* state, the system compares the time difference between the current system time and the last known reporting time to the `expectedInterval` specified in the configuration file (lines 6–11). If the sensor has not reported within the `expectedInterval`, a notification is sent (lines 7–8). Next, the system checks if `Analyzer` contains enough data to move the senor to the `normal` state (lines 9–10).

***Normal* State**. Consider the `update()` logic for a sensor in the *normal* state, shown in Listsing 5 (lines 10–16). The method queries the corresponding `Analyzer` for the average and standard deviation of the sensor's reporting intervals (lines 3–4). The maximum reporting interval is calculated based on the current average and the `numberOfStdDevs` parameter, which controls the sensitivity of acceptability. The system places the sensor in the *abnormal* state if the observed reporting interval is outside of `allowable`, and the timer corresponding to that sensor is then reset (lines 11–14). When the sensor is moved to the *abnormal* state, the sensor's `expectedInterval` is set to the observed reporting interval (line 13). This outlying interval serves as an estimate of the delay the sensor will experience while in the *abnormal* state.

Once a sensor enters the *normal* state, the `check()` method follows the actions in Listing 6 on wake-up (lines 12–16). `allowable` is calculated as before, in the `update()` method. The system then checks if the time difference between the current system time and the last known reporting time is outside `allowable` (line 13). If so, the sensor object is updated to reflect its change in status, and a notification is sent (lines 14–15).

***Abnormal* State**. When a sensor reports outside of its `expectedInterval`, it is placed in the *abnormal* state. In this state, the system assumes the sensor is malfunctioning and does not expect it to report within the calculated `expectedInterval`. The `update()` logic for the *abnormal* state is shown in Listing 5 (lines 17–25). `allowable` is calculated as in the *normal* state. If the observed reporting interval is less than the `expectedInterval` stored in the sensor object, it is assumed that the sensor is recovering, and the stored value is adjusted to reflect this (lines 18–19). The rate at which the adjusted interval is decreased is dependent on `expectedInterval`, `allowable`, and `decayConstant`. The rate is calculated as the difference between the sensor object's `expectedInterval` and `allowable`, divided by `decayConstant` (line 19). `decayConstant` enables network administrators to control how quickly abnormal behavior is expected to converge to normal behavior. If a sensor reports outside of its `expectedInterval`, `expectedInterval` is raised to the outlying value (line 23). Once `expectedInterval` is within `allowable`, the sensor is returned to the *normal* state (lines 20–22). This process ensures that a sensor is consistently demonstrating normal behavior before being returned to the *normal* state.

The `check()` method for the *abnormal* state follows the actions shown in Listing 6. The system checks if the difference between the current system time and the last known reporting time is within `expectedInterval`, and notifies network personnel accordingly (lines 17–20).

5. Evaluation

We collected data over a six-month period for 122 unique devices in various locations with various reporting intervals. These devices provided a total of 2,648,267 transmissions to analyze. First, we perform a baseline evaluation of the noti-

```
1  timeDifference = currentTime-sensorID.lastReportTime;
2
3  ...  average, stddev, and allowable are calculated as in
       Listing 5 on lines 3, 4, and 5 respectively ...
4
5  switch(sensorID.state) {
6    case(INITIAL):
7      if(timeDifference > configExpectedInterval) {
8        new Notifier(sensor);    }
9      if(sensorID.isFull()) {
10       sensorID.state = NORMAL;    }
11     break;
12   case(NORMAL):
13     if(timeDifference>allowable) {
14       sensor.status = dead;
15       new Notifier(sensor);     }
16     break;
17   case(ABNORMAL):
18     if(timeDifference > sensor.expectedInterval) {
19       new Notifier(sensor);     }
20     break; }
```

Listing 6. `StatusChecker check()` **Method**

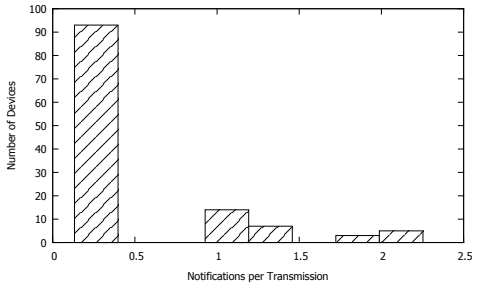


Figure 1. Static Approach Performance

fication tool's performance using the static approach. Next, the effects of the 3 configuration parameters introduced in the adaptive approach are evaluated. We then compare the performance of the adaptive and static approaches in terms of the number of notifications generated and the ability to customize performance based on user needs.

5.1. Static Approach

For the static method, `expectedInterval` was set for each device by personnel familiar with the reporting behavior of the device. The number of notifications generated was then compared to the number of transmissions sent by each device. The findings are shown in Figure 1. Figure 1 shows the distribution of notification rates across devices. Devices were placed in bins of width .265 (notifications per transmission) based on notification rate. The x-axis represents the number of notifications sent per transmission. The y-axis represents the number of devices with a notification rate within the given range. 93 out of 122 devices (76%) exhibited a notification rate of less than .265 notifications per transmission. This means that for the majority of devices, out of every 100 transmissions, approximately 26 or less of the transmissions were flagged as irregular, generating a notification. The other 29 devices exhibited notification rates greater than .795 notifications per transmission. Devices with a notification rate greater than .795 notifications per transmission generated at least 79 notifications for every 100 transmissions. Due to the configuration of `notificationTime`, which enables multiple notifications to be generated during a single interval between transmissions, some devices had a rate greater than 1 notification per transmission. The maximum rate observed was 2.12 notifications per transmission.

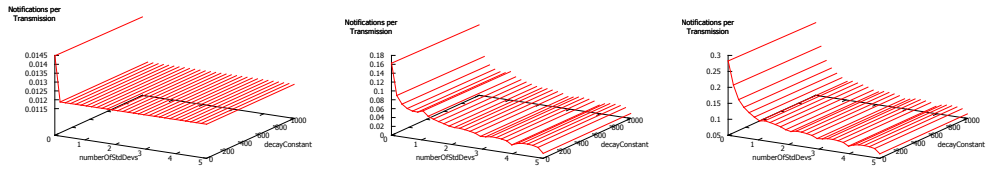
5.2. Adaptive Approach

To evaluate changes in notification rate as a function of changes to the three configuration parameters used in the adaptive monitoring approach, we first determine the value of `windowSize` for each device corresponding to a 24-hour period. Each device is placed in one of seven groups based on its programmed reporting interval: 6-second, 5-minute, 10-minute, 15-minute, 20-minute, 30-minute, and 1-hour. For each group, we run the adaptive approach using various combinations of `numStdDevs` and `decayConstant`. We varied `numStdDevs` between 0 and 5, and `decayConstant` between 0 and $1.4e6$. This range was chosen to explore expected time to recovery ranging from instantaneous (`decayConstant` equal to 0) up to 6 months (`decayConstant` equal to $1.4e6$).

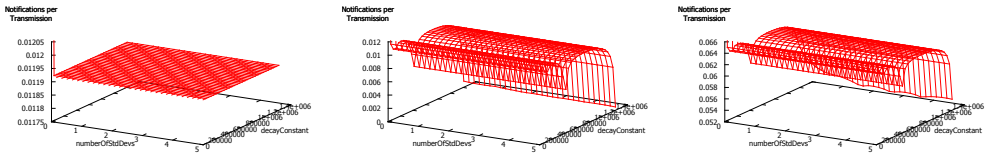
Figure 2 summarizes the effects of these two parameters on notification rate for 3 of the groups. The graphs were subdivided to provide a more detailed understanding of performance. Figures 2(a), 2(b), and 2(c) show the effects of `decayConstant` in the range of 0 to 1000, while Figures 2(d), 2(e), and 2(f) show its effects in the range of 1000 to $1.4e6$. For each group, the worst performance (i.e., the highest notification rate) occurs when both `numStdDevs` and `decayConstant` are equal to 0. This case is equivalent to the static approach. With the exception of the 6-second group shown in Figures 2(a) and 2(d), the notification rate generally decreases as `decayConstant` increases. Again, with the exception of the 6-second group, each group has a trough around `decayConstant` equal to 400,000 where it reaches a local minimum and then increases slightly before permanently decreasing toward 0. Devices reporting at 6-second intervals are virtual machines experiencing very little disruption. This is why devices in this group experience the lowest notification rate, as well as, the least variance in rate with respect to the parameters.

5.3. Static Versus Adaptive

Figure 3 depicts the behavior of both methods on a single device over time. The x-axis represents time, and the y-axis represents the interval between consecutive reports, in seconds. The static approach is considered in Figure 3(a), and the adaptive approach is considered in Figure 3(b). We observe a more tailored fit to the behavior of the device in the adaptive approach. Using the static approach, the majority of devices exhibit a notification rate below 30%, while some generate notification rates of 100%, and a few generate rates of 200%. The highest notification rate for the

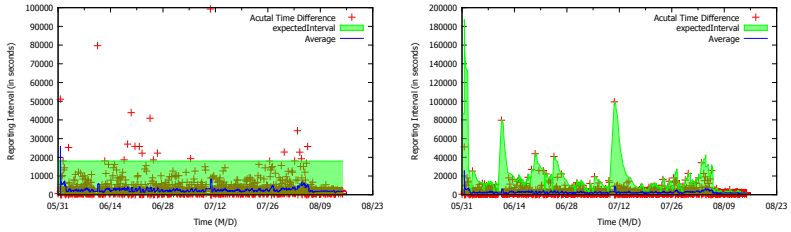


(a) 6-Second Group (low range) (b) 15-Minute Group (low range) (c) 30-Minute Group (low range)



(d) 6-Second Group (high range) (e) 15-Minute Group (high range) (f) 30-Minute (high range)

Figure 2. Effect of `decayConstant` and `numOfStdDevs` (`windowSize`=24-hours)



(a) Static Approach (b) Adaptive Approach

Figure 3. Static vs. Adaptive Approach

adaptive approach is 45%. This shows that in terms of notification rate, the adaptive approach can be used to generate fewer notifications. In some cases, variation may be intolerable, and users may wish to define malfunction using a constant threshold.

6. Conclusion

We presented two approaches to identifying malfunctioning sensors in a streaming monitoring network: a *static* approach and an *adaptive* approach. Trade-offs exist between the two approaches. The static approach may generate too few or too many notifications based on network managers' estimates of reporting behavior. The adaptive approach may generate too many notifications in sensors with sporadic variation. However, we found that both approaches are successful in detecting variations in a sensor's behavior and notifying personnel of sensor failure in real-time. With this method, a sensor cannot be flagged as malfunctioning unless it is actually exhibiting abnormal behavior. Due to `decayConstant`, it is possible to neglect repeated malfunctions within a close time frame, but this is by design. This work was supported by the NSF (CNS-0745846).

References

[1] N. Bartzoudis and K. McDonald-Maier. An adaptive processing node architecture for validating sensors reliability in a wind farm. In *BLISS 2007.*, pages 83–86, Aug 2007.

[2] R.N. Duche and N.P. Sarwade. Sensor node failure detection based on round trip delay and paths in wsns. *Sensors Journal, IEEE*, 14(2):455–464, Feb 2014.

[3] Stephen O. Friend and others. Standard deviaion: The new standard for out-of-pattern transaction analysis. *ACAMS Today*, January/February 2009.

[4] Shuo Guo et al. Find: Faulty node detection for wireless sensor networks. SenSys '09, pages 253–266, New York, NY, USA, 2009. ACM.

[5] MichaelA Hayes and MiriamAM Capretz. Contextual anomaly detection framework for big sensor data. *Journal of Big Data*, 2(1), 2015.

[6] Chun Lo et al. Pair-wise reference-free fault detection in wireless sensor networks. IPSN, pages 117–118. ACM, 2012.

[7] M.R. Napolitano et al. Kalman filters and neural-network schemes for sensor validation in flight control systems. *Control Systems Technology, IEEE Transactions on*, 6(5):596–611, Sep 1998.

[8] OASIS. AMQP:advanced message queuing protocol. `www.amqp.org/about/what`, 2015.

[9] Pivotal. RabbitMQ: messaging that just works. `www.rabbitmq.com`, 2014.

[10] Nithya others Ramanathan. SenSys '05, pages 255–267. ACM, 2005.

[11] Intelligent River. Intelligentriver®: from observational to operational. `intelligentriver.org`, 2015.

[12] Bilal Shebaro et al. Fine-grained analysis of packet loss symptoms in wireless sensor networks. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 38:1–38:2, New York, NY, USA, 2013. ACM.

[13] S. Taniguchi and Y. Dote. Sensor fault detection for uninterruptible power supply (ups) control system using fast fuzzy-neural network and immune network. In *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, volume 1, pages 99–104 vol.1, 2001.

[14] Ehsan Ullah Warriach et al. Fault detection in wireless sensor networks: A hybrid approach. IPSN, pages 87–88, New York, NY, USA, 2012. ACM.

[15] D.L. White et al. The Intelligent River©: Implementation of sensor web enablement technologies across three tiers of system architecture: Fabric, middleware, and application. pages 340–348, May 2010.

[16] Xi-Liang Zhang et al. Sensor fault diagnosis and location for small and medium-scale wireless sensor networks. In *Natural Computation 2010*, volume 7, pages 3628–3632, Aug 2010.

[17] J. Zhao et al. Computing aggregates for monitoring wireless sensor networks. In *The IEEE International Workshop on Sensor Network Protocols and Applications, 2003.*, pages 139–148, May 2003.