Managing application-level QoS for IoT stream queries in hazardous outdoor environments

Holger Ziekow[1], Annika Hinze[2] and Judy Bowen[2]
[1]*Business Information Systems, Furtwangen University, Germany*
[2]*Computer Science Department, Waikato University, New Zealand*
zie@hs-furtwangen.de, hinze@waikato.co.nz, jbowen@waikato.ac.nz

Keywords: Internet of Things, Quality of Service, RIoT, Stream Queries, Complex Event Processing, Hazard Prevention

Abstract: While most IoT projects focus on well-controlled environments, this paper focuses on IoT applications in the wild, i.e., rugged outdoor environments. Hazard warnings in outdoor monitoring solutions require reliable pattern detection mechanisms, while data may be streamed from a variety of sensors with intermittent communication. This paper introduces the Morepork system for managing application-level Quality of Service in stream queries for rugged IoT environments. It conceptually treats errors as first class citizens and quantifies the impact on application level. We present a proof of concept implementation, which uses real-world data from New Zealand forestry workers.

1 INTRODUCTION

It is the now well-established aim of the Internet-of-Things (IoT) to merge physical and virtual worlds with the goal of creating so-called ‘smart environments’. However, most IoT initiatives emphasise versions of smart cityscapes. Our project context targets the harder problem of a smart landscape, which will then be equally applicable to an urban setting.

In rugged, hard-to-reach and rough environments, such as hazardous outdoor workplaces (e.g., forestry, mining, and fishing) or recreational spaces (e.g., mountaineering and wild-water rafting), the Internet of Things is challenged by scarcity of resources and issues of robustness. Data may have to be obtained from sensor-equipped safety clothing (or other body-area-network contexts) as well as from machine-based sensors.

Our project embraces the notion of a *Rugged Internet of Things*– RIoT (Bowen et al., 2017b), supporting robust communication in remote outdoor environments that involve factors not typically considered in city-focussed IoT (weather, heavy machinery, safety clothing, environmental hazards and potentially rough handling). Communication may be intermittent and collaboration/data transfer between different types of sensors may be hard to establish and maintain.

Actions may need to be taken based on aggregated data and patterns detected in the RIoT. The required pattern detection mechanisms therefore have to consider possible errors and delays in available data streams.

This paper introduces the Morepork system, an uncertainty quantifier system which manages the application-level errors introduced by data stream distortions in a rugged IoT environment, through deliberate Quality-of-Service management at application level. It consists of an architectural framework that manages quality of service for RIoT applications. In Morepork, data stream errors are considered as first class citizens. To demonstrate we use the real life application of the Hakituri project (`isdb.cms.waikato.ac.nz/research-projects/hakituri/`) which develops a wearable monitoring approach for New Zealand (NZ) workers in hazardous work environments, particularly a forestry context.

The remainder of this paper is structured as follows: Section 2 provides a RIoT use case description from the Hakituri project on monitoring forestry workers in hazardous work environments and introduces our concept of application-level QoS management for RIoT applications, while Section 3 describes the architecture of Morepork, which implements these QoS management concepts. A proof of concept application of the Morepork system with real-world data from the Hakituri project is described in Section 4. Related approaches are discussed and compared in Section 5, and the paper finishes with a summary.

2 Use case and Morepork concept

This section describes the use case defined by the Hakituri project, the data collections undertaken, and the core concept of our Morepork approach.

2.1 Use case: hazard detection in NZ forestry outdoor work environments

Because New Zealand forestry has a high rate of workplace fatalities, our project targets this industry's specific settings as a test use-case. Forestry is labour-intensive (most work more than 40-60 h/week), with the tree felling and breaking out being main activities contributing to serious accidents. The work is both physically and mentally demanding, with operations being performed irrespective of the weather.

An independent review identified a number of factors contributing to the high accident rate such as fatigue, lack of training, poor health and safety cultures (Adams et al., 2014). We focus on identification of fatigue as it results in slowing of reaction times and decision making which are crucial to safety in a hazardous environment. Fatigue is a subjective physiological state experienced by individuals as a result of either physical or mental exertion (Hockey, 2013). Physiological changes occurring as one enters a fatigued state can be used as indicators of reduced performance, for example, changes in heart rate and heart rate variability (physical fatigue) and reduced reaction time (mental fatigue).

2.2 Data collection: personal monitoring of forestry workers

We are using data obtained in the wild through the Hakituri project, which aims to predict hazardous situations in forestry by using wearable technology and environmental sensors. This data constitutes the results of the first in-situ data collection on fatigue in the forestry industry in NZ. Three studies were conducted with forestry workers in their outdoor working environment. Participants were sourced from three forestry industry subcontractors performing harvesting operations (Bowen et al., 2017a). The studies collected streaming data on heart rate (HR) and heart rate variability (HRV) as potential measures of fatigue. HRV consists of changes in the time intervals between consecutive heartbeats. They also measured aggregated step counter data as a measure of physical work. Additional point data were collected as feedback on mental and physical fatigue (reaction times and NASA-TLX questionnaire) and contextual data such as temperature and humidity.

2.3 Morepork concept: Data Errors as first class citizens

The fundamental concept of the Morepork system is to treat data errors as first class citizens. This means that instead of considering data errors as the exception in otherwise well-formed and complete data streams, data errors are treated as an ordinary part of the application data streams. Dealing with errors as exceptions in the data stream means that typically efforts are made to reduce their impact through preprocessing (i.e., cleaning), while the data analysis itself proceeds without acknowledging errors. In the case of Morepork, instead of implementing primitive QoS metrics that capture properties of the input, we shift to metrics that are relevant for the application level as part of the data analysis. That is, we present a generic approach to model the impact of data errors on application level. We use such models to make the application-level error explicit as a QoS parameter and monitor it at run time. Morepork thus creates awareness for errors and their magnitude, while any reduction of errors is subject to the error handling mechanisms in the application logic.

This paper showcases how these application-level QoS measures are implemented for IoT stream queries in Morepork, and uses real-world data from forestry studies to explore the implications for data stream analysis. To eliminate communication errors, the studies described above collected the data at the point of origin. These data streams can therefore be used in our work as best case data, without external system interference. Interference, e.g., through poor communication, is understood to cause incomplete data streams. In our model, these data errors are treated as first class citizens and application-level QoS is used as a quality measure for the analytics model depending on the current data quality (i.e. completeness of the data stream).

3 Morepork System Architecture

In this section we introduce our architectural framework for managing quality of service for the harsh conditions of a rugged internet of things (RIoT). Figure 1 provides an overview of the key components and their relations. The architecture follows a common IoT middleware structure, see e.g. (Bandyopadhyay et al., 2011). The core of the system is built on top of a data capturing layer (Fig. 1 bottom) that acts as adapter to different sensor sources. Sensor data arrives at the core (Fig. 1 middle) in an event-based manner via a message bus. The core wraps the

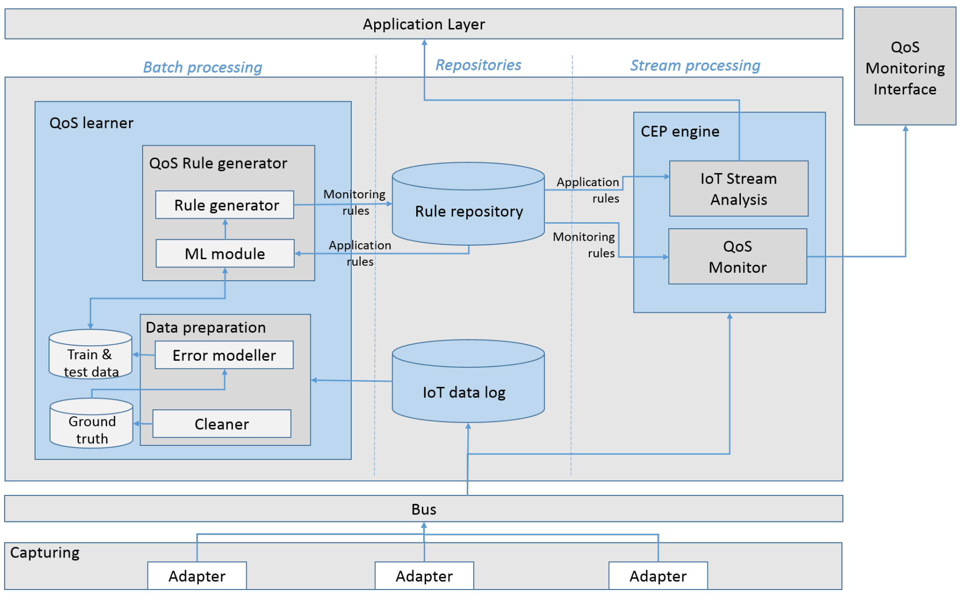

Figure 1: Morepork Architecture and Components

application-specific stream analytics and augments it with the functionality to continuously monitor the application-level QoS. The core outputs two main data streams: (1) the result stream of application-specific stream analysis, toward the application layer, and (2) the application-level QoS information, accessible via QoS monitoring interface.

Overall we distinguish three groups of components. These are (1) repositories, (2) batch processing components for QoS learning, and (3) stream processing components for live analysis. We describe these components in detail throughout the next subsections.

3.1 Storage components

Morepork includes two types of databases. The first database is the *IoT data log*. It stores all incoming raw data from IoT sensors. Conceptually it is equivalent to the master data set in the lambda architecture (Marz and Warren, 2015). The second database is the *Rule repository*. It holds the processing logic for the components *IoT Stream Analysis* and *QoS Monitor*. It thereby serves as the basis for rule managers as discussed in (Cugola and Margara, 2012). Note, that the specific encoding of these rules depend on the execution environment (i.e. the specific choice of CEP engine).

3.2 Batch processing: QoS learning

The batch processing components for QoS learning automatically derive rules for continuously evaluating the application-level QoS. They comprise two main components: the *QoS Rule Generator* and a component for *Data Preparation*. Each of these include two sub-components.

Data Preparation. The *Data preparation* component takes raw data from the IoT log as input and produces data that is subsequently used for training and testing QoS monitoring rules. In a first step, the data is *cleaned* such that a data set can be obtained that is free of errors and can serve as ground truth for further processing. A default approach is to only keep data streams of a defined minimum length and without gaps.

The second step of the data preparation is done by the *Error modeller*, generating training and test data for the subsequent learning of QoS monitoring rules. The error modeller analyses data in the IoT log to build a model that describes the occurrence of errors. An alternative is that domain experts create a model of what errors to expect. The error modeller then uses the model on the error-free data streams (ground truth) to create a copy that contains realistic errors. The result is a repository of erroneous data streams where the correct version of the stream is known as well.

QoS Rule Generator. The *QoS Rule Generator* uses machine learning to create monitoring rules for continuously predicting the quality of service for the application specific stream analytics (Application Rules). It comprises the two sub-components *ML module* and *Rule Generator*.

Figure 2 shows details of the operations within the *ML module*. The module runs in batch mode to create monitoring rules for each application rule in the rule repository. It first creates training and test sets for learning the relation between errors in the input stream and the QoS of each rule. This is done by replaying the corresponding stream from the *Train & Test Data Storage* to create rule specific test and training sets. These test and training sets have error characteristics of the input stream as features and corresponding errors of the analytics results (application rule) as labels. The system can determine the application error by leveraging the previously extracted ground truth (i.e. error free streams) and corresponding streams with introduced errors in the *train & test data storage*. Formally, the computation of the error is

$$\varepsilon(s,s') = err(appRule(s), appRule(s')).$$

Here *err* is an error metric (e.g. absolute error) and *appRule* is the output of running the application rule over an error free stream s and a version s' of the stream with introduced errors. The feature set is formally defined as $f(S')$ where f is a function that outputs a set of aggregates over S'. An example for instantiating f is to compute the percentage of missing messages and the variance of the data values over a time window.

With the training set and test set of the structure $\langle f(S'), err(appRule(S), appRule(S'))\rangle$, the ML module builds a prediction model for estimating application level QoS. Note that with the given structure, the prediction model may not only base predictions on the error characteristics (i.e. missing values) but also on the currently observable data characteristics. For instance, it may learn that missing values affect the error more when the data stream currently shows high variance than when values are currently stable.

Figure 2 also illustrates how the ML module may use a multitude of stream observers to create candidate feature sets and a multitude of ML mechanisms to build candidate prediction models.

The subsequent processing step after the ML module is the *Rule generator*. The rule generator takes the learned models for application level QoS prediction as input and transforms them into executable rules. Here we use the term *rules* to denote a format that is executable in the CEP engine. Note, that this format depends on the chosen embodiment of this engine. For instance, one may chose Esper for the CEP engine and create the rules in the declarative event processing language EPL[1]. Alternatively, one may wrap the model in a bolt for executing it in a Storm topology[2].

[1]`http://www.espertech.com/`

[2]`http://storm.apache.org`

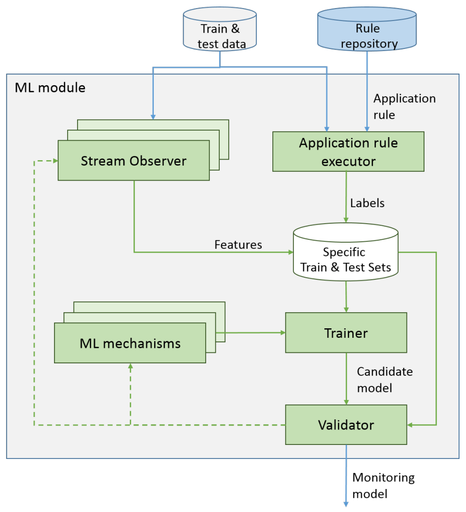

Figure 2: Morepork ML module

3.3 Stream Processing: real-time data analysis

The stream processing components realise the application logic (IoT Stream Analysis) and corresponding QoS monitoring (QoS Monitor) at runtime. That is, these components continuously run over live input data from IoT devices. Within the Morepork we propose to use a complex event processing engine (CEP engine) or stream processing engine as execution environment for the continuous analysis. These are dedicated systems that leverage in-memory technology for efficient event-driven processing. (Cugola and Margara, 2012). The specific choice of engine determines how to define the processing logic, i.e. application rules and monitoring rules in the rule repository.

The Morepork rule repository contains monitoring rules *monRules* for the QoS Monitor and application rules for the IoT Stream Analysis over the erroneous live data S'. Executing a monitoring rule $monRules(S')$ consists of two main parts. The first part is to observe the current input stream and to derive the features for QoS prediction in real-time. The second part is to feed the extracted features in the model for QoS prediction and to derive the current application level QoS. The application rules are of arbitrary structure and we deliberately do not pose any constraints on them in the Morepork system.

4 Proof of concept

In this section we describe an instantiation of Morepork components as well as tests of this instantiation with data from the application domain. We thereby provide proof of concept as well as an illustrative example of how to build an embodiment of the Morepork concept. First we describe the data and sample application logic that we use in the tests. Second, we describe our sample implementation and third, we discuss the results of our tests.

4.1 Sample application and data

For our tests we choose a simple logic for stress and fatigue monitoring as an example, based on the use case data collected, see Section 2.1.

We used *data measures* of heart rate variability (HRV), taken every minute for each worker, provided as stream data. The data was collected in the context of the experiments described in (Bowen et al., 2017b). HRV data are personalised measures that require baseline determination, with generally higher HRV indicating greater physical fitness. Complex HRV signals provide data in high-frequency (HF), low-frequency (LF), and very low-frequency (VLF) bands, of which the LF/HF ratio is typically used as an indicator for occupational stress and fatigue (Järvelin-Pasanen et al., 2018). Heightened occupational stress is associated with lowered HRV, specifically with an increase in LF/HF ratio, with the threshold being dependent on person-related baselines. Specifically, for our tests we use HRV recordings of one of the study participants, taken on five consecutive days, focussing on the LF/HF ratio. The measurements were taken over 8 hours each day, resulting in 3151 measurements overall.

The sample *application logic* is to monitor the sliding average of the LF/HF ratio and to detect if a threshold is crossed. Crossing a threshold indicates a high stress level and one may initiate a break or refrain from dangerous work in response. For our test we use a sliding average over 10 values to indicate the stress level.

However, if measurements get lost in the communication, the application shows an average value that differs from the true value. With the Morepork concept, the system can learn to estimate the magnitude of this error for each data point and make this transparent to the user. It can factor in, for instance, the number of lost messages and the current context (e.g. if the LF/HF ratio is rapidly changing or rather constant) to produce an estimate. It is then possible on application level to factor in the current certainty of the data, that is, one can make an educated decision if the displayed stress level is alarming or not, given the current certainty of the value.

4.2 Proof-of-Concept Instantiation

We here describe a proof-of-concept instantiation of the Morepork concept for the sample application. We discuss implementation of the components and dataflow that was described in Section 3.

The starting point for the analysis is the IoT data log with a history of captured raw measurements (referred to as S). In our sample, we base this log on the five days recording of LF/HF data. For simplicity we concatenate the values of the five days into one continuous value stream.[3]

4.2.1 Data Preparation

The first step in the processing pipeline is data preparation, to create ground truth as well as the training and testing data. For the ground truth we need to extract undistorted parts of the IoT data log.

In our sample application we obtained that data from an experiment that recorded locally and without distributed transmission. Hence we are in a special situation where we get training data that is free of communication errors and can omit the cleaning step.

The second step is modelling the communication errors that we expect in the application setup in the wild, where measurements are transmitted via wireless communication from remote locations and one must expect gaps in the stream. In the Morepork framework this is the responsibility of the error modeller. For our test we choose a simple error model that drops messages with a probability of 50%. The error modeller stores the undistorted data S as well as a version with gaps in the repository for test and training data (named S'). We then continue the processing with half of the values (S_{train} and S'_{train}) and set aside the rest to later use in simulation test (S_{sim} and S'_{sim}).

4.2.2 QoS Rule Generator

The next component in the processing pipeline is the QoS Rule Generator with the ML module.

Specific test and training sets are created using the (a) addressed application logic from the *rule repository* and (b) data from the *repository for test and training data*. That is, the ML module creates features and labels for the subsequent model training. In our sample application, we compute a rolling average $avg(t, S_{train}, n)$ over a data window of size n from data

[3]This means that at the beginning of a new day, some data from the previous evening is part of the data window. However, the very first values of each day typically carry distortions because of how the sensors are mounted, and any effects of the data concatenation are minimal.

stream S_{train} for each time slice t. Hence, our objective is to learn a model that predicts the application level error, given data losses.

We compute the corresponding label by running the application log (sliding average) over the undistorted data (ground truth) S_{train} as well as over the copy S'_{train} with introduced communication errors. Both data sets are outputs of the preceding *Data preparation* component. The absolute value of the difference between the two computations $|avg(t, S_{train}, n) - avg(t, S'_{train}, n)|$ is the label for each time slice t in our specific test and training set.

Similarly, we compute the features as function $f(t, S'_{train}, n)$ over a data window of size n from data stream S'_{train} for each time slice t. Again S'_{train} is the copy of the ground truth with induced errors. In our sample implementation, we use two features: One is the number of data gaps in the current data window. This captures the current data quality. The other feature is the variance over the window. This feature captures the current context with regards to how stable/unstable the measurements are. (Note that the variance serves as example for context data in our proof-of-concept implementation and additional context information–such as time of the day–may be included as well). With this limited set of features and for the sake of simplicity in this example, we omit a search through the feature space and decide on only one candidate model type (i.e. decision tree).

Subsequent to the creation of the specific training and testing sets, the model building is performed. In our implementation, we train a model that predicts the application level error distribution for each data point. For this we use a regression tree from the R package *party* (`http://party.r-forge.r-project.org`), which is based on the algorithm described in (Hothorn et al., 2006). The resulting tree predicts the expected error and not an error distribution. However, it associates each leaf node of the tree with the corresponding training samples. For each node we use these training samples to fit a Gaussian distribution of the error size in each node.

The Morepork concept uses a *Rule Generator* to transform the model into a form that can run over a data stream. However, in our Proof of Concept implementation, we simulate a live data stream by iterating through a pre-recorded log and therefore proceed directly with the R code instead of CEP rules.

4.2.3 CEP Engine

The components for IoT stream analysis and QoS Monitor run in parallel in the CEP engine. In our instantiation, we simulate the *IoT stream analysis* through the replay of S'_{sim} to generate the application level output. For the evaluation we additionally run the application logic over a replay of S_{sim} to obtain the application level ground truth.

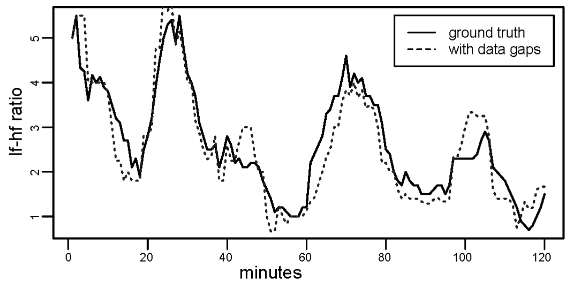

Figure 3: Data on application level

The *QoS Monitor* provides continuous feedback about the application level quality of service in terms of the accuracy of the query results. In our sample application we realize this through providing a confidence interval around the query results. We obtain this interval though applying the previously learned tree for error prediction on the current data window and looking up the leaf-specific error distribution. We then tested the QoS monitor through a replay of the set-aside data S'_{sim} and comparison with a replay of S_{sim}. Thereby we can observe the effects of having a dynamic (i.e. context dependent) confidence interval and can test if the predicted interval is valid. The subsequent section shows the results.

4.3 Experimental Results

The aim of the experiments with the proof-of-concept implementation is to test the viability of the Morepork approach. Specifically, we aim to validate the applicability on real data and to analyze how the Morepork concepts play out in a sample scenario. We tested the implementation with the above described measurements of LF/HF ratio and an assumed loss of 50% of the values. The application-level QoS metric is here defined as the absolute difference between the computed sliding average in the LF/HF ratio measurements with complete data (ground truth) and incomplete measurements (application output with errors). Figure 3 displays a snap shot of the application level data (i.e. smoothed LF/HF ratio). It shows at application level the difference between the ground truth – without communication errors – and the results based on an input stream with gaps. The QoS monitor in Morepork dynamically quantifies the expected magnitude of the errors for each data point through a confidence interval.

In our experiments we used both S_{train} and S'_{train} for training a regression tree that predicts the error. Figure 4 shows the resulting tree. According to the

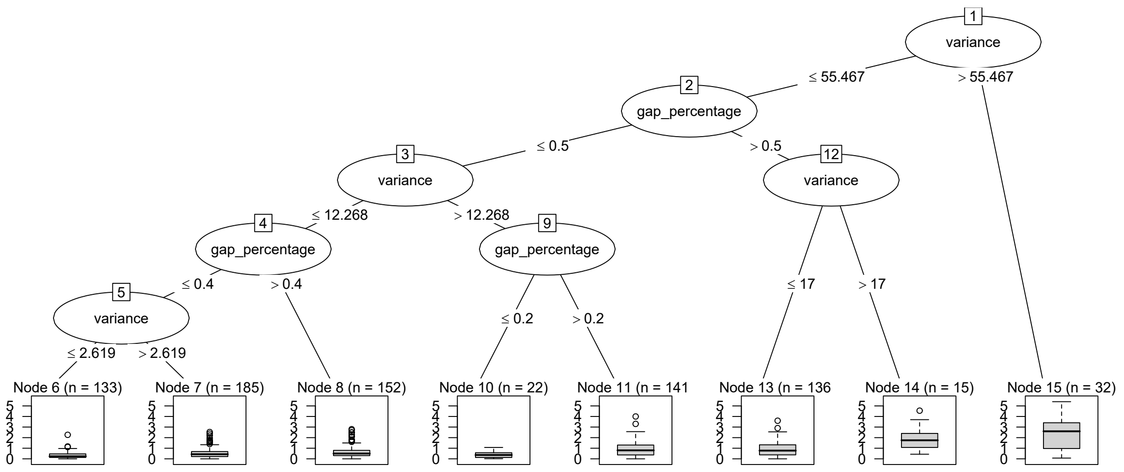

Figure 4: Learned model

model structure, the application error depends on the current data quality (`gap_percentage`) as well as on the current data context (`variance`). This manifests in the corresponding model nodes. The statistics for the leaf nodes show clearly distinctive distributions for the application level errors dependent on the current `gap_percentage` and variance. This suggests the feasibility of providing a confidence interval for each data value.

We set aside the second half of the 3151 LF/HF ratio measurements (S_{sim} and S'_{sim}) for testing the applicability of an adaptive confidence interval in the QoS Monitor. Figure 5 shows a snapshot of the results. The figure depicts the application-level data (sliding average of the LF/HF ratio) with data gaps (white squares) and without data gaps (black squares) in the input data. We selected here a situation that illustrates the crossing of a threshold (which is marked by a horizontal line). The QoS monitor adds for each data point a 95% confidence interval that quantifies the expected application level data quality. The confidence interval is inferred individually for each point based on our learned model, depending on the gaps and variance in the respective data window. The snapshot shows how the confidence interval changes over time and adapts for each value. We highlighted a potential area of interest in which the (erroneous) application data appear to be well away from the threshold, however, the wide confidence interval reaches beyond the threshold and indicates that another level of fatigue may have been reached. Inspecting the real data value shows that this has indeed been the case.

Overall we found that 92% of the data were within the 95% confidence interval. 87% of the data were in a 90% confidence interval and 79% in an 80% confidence interval. These observations with the proof-of-

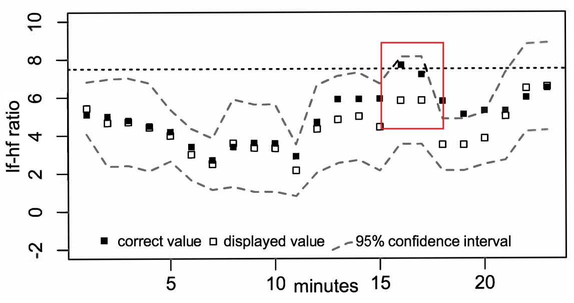

Figure 5: Data with confidence level in QoS monitor

concept implementation confirm the applicability of the Morepork concepts on real data.

5 Related Work

The closest related work to Morepork are IoT middleware and complex event processing (CEP) systems. Such systems have been proposed for a range of application domains (e.g., (Merlino et al., 2014; Strohbach et al., 2015; Tönjes et al., 2014; Adeleke et al., 2017)). These works are related in the sense that they present architectures and system for handling IoT data streams. However, such systems have little or no explicit support to deal with quality of service in the presence of errors. A common approach is to address errors in a best effort manner in a cleaning step. Often, it is left to the application developers to deal with the errors. In contrast, Morepork treats errors as first class citizens and offers dedicated support for estimating their impact on application level. A CEP system that addresses QoS is Aurora (Abadi et al., 2003). It adapts processing under consideration of QoS parameters. It considers response time, tuple drops and produced values as QoS parameters. Unlike Morepork, it offers no support to estimate and monitor the QoS on application level.

Another example of related work is the MILTON measure for event detection (Efros et al., 2017). The MILTON measure aims to quantify the effect of lossy transformation on event detection processes. The general concept of MILTON is similar to our approach of quantifying the effect of errors on application level QoS; our work draws on the principles behind the MILTON measure. However, MILTON does not explicitly consider communication errors and provides no architecture for dealing with QoS in an IoT system. In that sense it is only loosely related to our work. Similar, the machine learning approaches suggested by (Shrestha and Solomatine, 2006) are related to parts of the Morepork system. Specifically, our approach in training a model for estimating the error of another model is inspired by the work of Shrestha et al. However, these works only address a small part of the Morepork concept and do not aim at providing a system of similar scope.

6 Conclusion

This paper introduced the Morepork system for managing application-level Quality of Service in stream queries for rugged IoT environments. To the best of our knowledge the system is unique in its approach for treating errors as first class citizens and providing a generic solution for making application-level QoS explicit in an IoT system. Morepork thus acknowledges the error-prone nature of data streams from real-world IoT applications in rugged outdoor environments. It provides a system for generic support of IoT applications with application-level QoS. In Morepork, machine learning components are used as wrappers around the application-specific data analytics logic. To explore the Morepork concept for a real-world setting, we used data from the Hakituri project in a Proof of Concept implementation.

Acknowledgements. We acknowledge the work of Chris Griffiths in collecting the forestry data, and the support of the forestry contracting companies and workers.

REFERENCES

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139.

Adams, G., Armstrong, H., and Cosman, M. (2014). Independent forestry safety review – an agenda for change in the forestry industry. report published by the NZ Ministry for Business and Innovation, Wellington.

Adeleke, J. A., Moodley, D., Rens, G., and Adewumi, A. O. (2017). Integrating statistical machine learning in a semantic sensor web for proactive monitoring and control. *Sensors*, 17(4):807.

Bandyopadhyay, S., Sengupta, M., Maiti, S., and Dutta, S. (2011). Role of middleware for internet of things: A study. *International Journal of Computer Science and Engineering Survey*, 2(3):94–105.

Bowen, J., Hinze, A., and Griffiths, C. (2017a). Investigating real-time monitoring of fatigue indicators of new zealand forestry workers. *Accident Analysis and Prevention*. in press.

Bowen, J., Hinze, A., Griffiths, C., Kumar, V., and Bainbridge, D. (2017b). Personal data collection in the workplace: Ethical and technical challenges. In *British Computer Society Human Computer Interaction Conference*, pages 57:1–57:11.

Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15.

Efros, P., Buchmann, E., Englhardt, A., and Böhm, K. (2017). How to quantify the impact of lossy transformations on event detection. *Big Data Research*, 9:84–97.

Hockey, R. (2013). *The Psychology of Fatigue, Work, Effort and Control*. University of Sheffield.

Hothorn, T., Hornik, K., and Zeileis, A. (2006). Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical statistics*, 15(3):651–674.

Járvelin-Pasanen, S., Sinikallio, S., and Tarvainen, M. (2018). Heart rate variability and occupational stress-systematic review. *Industrial Health*.

Marz, N. and Warren, J. (2015). *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co.

Merlino, G., Bruneo, D., Distefano, S., Longo, F., and Puliafito, A. (2014). Stack4things: integrating iot with openstack in a smart city context. In *Smart Computing Workshops*, pages 21–28.

Shrestha, D. L. and Solomatine, D. P. (2006). Machine learning approaches for estimation of prediction interval for the model output. *Neural Networks*, 19(2):225–235.

Strohbach, M., Ziekow, H., Gazis, V., and Akiva, N. (2015). Towards a big data analytics framework for iot and smart city applications. In *Modeling and processing for next-generation big-data technologies*, pages 257–282. Springer.

Tönjes, R., Barnaghi, P., Ali, M., et al. (2014). Real time iot stream processing and large-scale data analytics for smart city applications. In *European Conference on Networks and Communications*.