Providing Fault Tolerance via Complex Event Processing and Machine Learning for IoT Systems

Alexander Power
Gerald Kotonya
{a.power3,g.kotonya}@lancaster.ac.uk
School of Computing and Communications
Lancaster University, United Kingdom

ABSTRACT

Fault-tolerance (FT) support is a key challenge for ensuring dependable Internet of Things (IoT) systems. Many existing FT-support mechanisms in IoT are static, tightly coupled, inflexible implementations that struggle to adapt in dynamic IoT environments. This paper proposes Complex Patterns of Failure (CPoF), an approach to providing reactive and proactive FT using Complex Event Processing (CEP) and Machine Learning (ML). Error-detection strategies are defined as nondeterministic finite automata (NFA) and implemented via CEP systems. Reactive-FT support is monitored and learned from to train ML models that proactively handle imminent future occurrences of known errors. We evaluated CPoF on an indoor agriculture system with experiments that used time and error correlations to preempt battery-depletion failures. We trained predictive models to learn from reactive-FT support and provide preemptive error recovery.

CCS CONCEPTS

• **Computer systems organization** → **Embedded and cyber-physical systems**; **Dependable and fault-tolerant systems and networks**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

internet of things, fault tolerance, complex event processing, machine learning, dependability, automata

ACM Reference Format:
Alexander Power and Gerald Kotonya. 2019. Providing Fault Tolerance via Complex Event Processing and Machine Learning for IoT Systems. In *9th International Conference on the Internet of Things (IoT 2019), October 22–25, 2019, Bilbao, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3365871.3365872

1 INTRODUCTION

The *Internet of Things* (IoT) provides an infrastructure for a vast network of real-world "things" to be discoverable and interconnected in order to achieve greater value and services in domains such as logistics, healthcare, and agriculture [4, 20]. An important challenge to realize IoT is how to provide a *dependable* infrastructure for billions of devices and deliver their intended services without failing in unexpected and catastrophic ways [9]. This problem can be addressed by designing IoT systems to support *fault tolerance* (FT) so that they can detect errors caused by faults and recover from them to mitigate and prevent service failures.

The dynamic and emergent nature of IoT systems makes it difficult to specify adequate error detection and recovery mechanisms *a priori*. Most IoT systems operate in dynamic contexts, where new services, devices, and features may be added, removed, and changed over time. Therefore, IoT systems should be able to harness *context awareness* for dynamic and intelligent decision making [32]. For FT, this means that error-detection logic must constantly evolve with system changes.

Current FT-support implementations in IoT are inadequate because they are designed for bespoke architectures [31] and specific applications (e.g. healthcare [36]), and only designed to handle specific faults (e.g. component failures [16] and communication link failures [18]). IoT systems present three key challenges for ensuring effective FT support, namely: (1) how do we infer erroneous system behaviors given system context? (2) How do we mask the failure of many constrained and ephemeral smart objects? (3) How do we ensure FT support remains relevant?

Sezer et al. [29] identify that both rule-based and supervised learning approaches are commonly used for context-based reasoning in IoT systems. *Complex Event Processing* (CEP) is used in research and industry to identifying complex situations (*composite events*) by defining rule-based patterns in stream data (*primitive events*). It is considered the paradigm of choice for monitoring and reactive applications [6, 13], making it ideal for *reactive*, context-aware FT support. *Machine Learning* (ML) has been widely proposed in literature to address many IoT use cases, such as smart traffic/cities, healthcare, and agriculture [23]. In IoT, ML has been used for making predictions, finding insights hidden in data, and making intelligent decisions from the big data generated in IoT [24]. These attributes are useful for providing data-centric FT support that uses data to identify and anticipate erroneous system behaviors for proactive FT support.

Our proposed solution is *Complex Patterns of Failure* (CPoF), an approach that uses CEP and ML to provide reactive- and proactive-FT support for IoT systems, where error-detection events are defined as *nondeterministic finite automata* (NFA) which act as input to CEP systems. Events are recursively fed back into the CEP system for use in more complex detection events, a process we call *Complexity via Recursion* (CvR). Error events are used to train ML

IoT 2019, October 22–25, 2019, Bilbao, Spain

ACM ISBN 978-1-4503-7207-7/19/10…$15.00
https://doi.org/10.1145/3365871.3365872

models to preemptively detect and recover from errors in the future. The rest of the paper is as follows. Section 2 discusses related work. Section 3 explores NFAs for error detection. Section 4 covers CvR. Section 5 evaluates CPoF. Section 6 concludes our work.

2 RELATED WORK

Reactive FT refers to a situation where system recovery is initiated *after* an error has occurred and been detected. Reactive FT techniques fall within five categories outlined by Egwutuoha et al. [14], namely: redundancy, migration, failure semantics, failure masking, and recovery. Choubey et al. [8] presented a smart home architecture where sensors were analyzed for correlations so that, if some data could be predicted using other correlating sensor data, a neural network was trained to predict the data, which provided *redundancy* and *job migration* for devices. Hu et al. [16] presented a framework that enabled developers to employ backward error recovery via *checkpointing* implemented via user-defined exception handling to handle sensor component failures.

Past literature has also explored *proactive FT*, where system recovery is initiated *before* an error has occurred. As software "ages" during its life-cycle, there is an increased failure rate and performance degradation that is attributed to software changes and 'elusive' bugs that accumulate and lead to an eventual software failure [10]. *Software rejuvenation* stops aging software, cleans its internal state, and resumes it, where the downtime is masked using FT strategies [5]. Umesh et al. [33] used rejuvenation to examine the behavior and utilization of *virtual machines* (VMs) such that, when a VM was predicted to fail imminently, services were migrated to new VMs to avoid downtime. *Preemptive migration* is designed to prevent failures by preemptively migrating parts of a system away from hardware or software that will soon fail [26]. However, it is not capable of addressing all types of failures, and thus a mixture of reactive and proactive FT provides coverage for unpredictable faults as well.

A core challenge when analyzing stream data is how to infer the occurrence of interesting and *complex* situations in the environment. We focus on NFA-based CEP systems because NFA is the established mechanism upon which most CEP systems are based [15]. The language model of existing CEP systems share many common operators, such as [12, 37]: (1) **logic operators** that define rules by combining several items (e.g. conjunction, disjunction, negation); (2) **sequences** that are similar to logic operators but items are order dependent i.e. detected in a specified order; **iterations** are a special case, where the sequence length is not *a priori* known, enabling unbounded sequences; (3) **windows** involve limiting portions of input flow to those only within a given timeframe, also ensuring the termination of unbounded iterations; and (4) **event selection**: events can be dispersed over many input streams and, thus, are not always contiguous; we focus on the **skip till next match** (STNM) selection policy, where irrelevant events are skipped until the next relevant event occurs.

Moreno-Cano et al. [25] applied CEP to 'trip-chaining', whereby a transit system, which could only record the *origin* of a trip made by a user, attempted to recover both the origin and the destination of a trip. Combining CEP with a fuzzy clustering album enabled a large volume of profiles to be identified in semi-real-time and produced many meaningful patterns of transit usage. Wang et al. [34] proposed the *multilayered adaptive dynamic Bayesian network* model for large-scale transportation IoT that used proactive CEP to mitigate or eliminate undesirable future events (e.g. traffic congestion) using predictions and automated decision making methods. Wang et al. [35] later proposed a predictive CEP method based on evolving Bayesian networks that used a Gaussian mixture model and the expectation-maximization algorithm for approximate inference of traffic prediction. Their evaluation included simulated and real data from a traffic-monitoring network, where vehicles traveled to and from home, office, and supermarket locations. Akbar et al. [1] proposed the *adaptive moving window regression* algorithm that combined CEP and ML to provide a proactive solution to cope with dynamic environments and IoT data. The model utilized a moving window for training the model and updated as new data arrived. Their evaluation used traffic data to predict traffic speed and intensity throughout the day, and CEP was used to infer traffic state and avoid congestion.

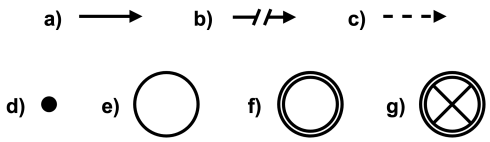

Figure 1: NFA diagram symbols: (a) transition between states using STNM; (b) same as (a), omitting some intermediary states; (c) arrow pointing to composite event(s) produced after NFA acceptance; (d) a starting point; (e) a state; (f) an accepting final state; (g) a non-accepting final state.

3 ERROR DETECTION

An *error* is a deviation of a program operation from its exact requirements due to the presence of bugs that only appear when a program is running or being tested [19]. An error is *detected* if its presence is indicated by an error message/signal, and *latent* if undetected [3]. We want to consider how service failures can be detected using error checks expressed as NFAs that can be implemented in CEP systems.

3.1 Automata Model

Our automata model is similar to that in [28] and we represent our NFA diagrams using the symbols in Figure 1. For each state transition there is an event e_i that causes a transition to some state S_i, starting at state S_1. An event has a value e^v, i.e. its data; an origin e^o, i.e. where it was generated; and a timestamp e^t, i.e. when it was generated. We assume the STNM selection policy (Section 2) for transitions because strict contiguity is unsuitable for analyzing the high-volume, heterogeneous data in IoT. S_A is the final *accepting* state of an NFA. A dashed arrow (Figure 1c) points from S_A to a composite error-detection event d and an error-recovery event r; recovery is not mandatory. The events that caused a composite event to be generated are called the *pattern* of the event. S_F is the *non-accepting* state that causes an NFA to halt when transitioned to it. An NFA might transition to S_F regardless of its current state for

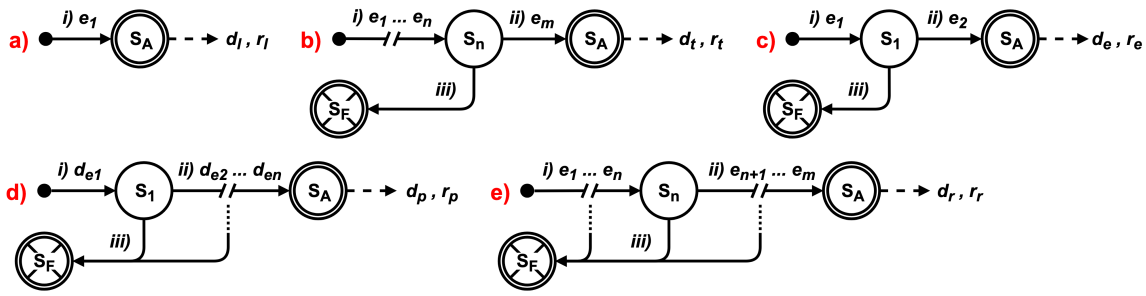

Figure 2: NFAs for (a) limit, (b) trend, (c) performance, (d) persistence, and (e) correlation checking.

state clearance. It can be implemented using: (1) a *time window*, that halts on a time elapse; and (2) an *until* predicate, that halts if true.

3.2 Error-Detection Checks

Lee et al. [21] define seven error-detection checks: replication, timing, reversal, coding, reasonableness, structural, and diagnostic checks. Due to space constraints, we will explore three of these, discussed next.

3.2.1 Reasonableness. Event reasonableness refers to whether an event is acceptable based on criteria envisaged by the system designer and implemented via the internal design and construction of the system [21]. We consider three types of data unreasonableness explored by Liu et al. [22], namely: (1) *outliers*, where e^v exceeds some threshold ϵ; (2) *stuck-at faults*, where the last n event values are all equal; and (3) *spikes*, where some values in the last n events are drastically higher or lower than others, resulting in high variance.

Limit Checking. Detecting outliers involves checking if e^v is 'within its limits'. In a NFA, this would simply require a predicate that checks if e_1 is *not* within some defined limits (Figure 2a), e.g. $\neg(\epsilon_{min} \leq e^v \leq \epsilon_{max})$. If true, the NFA transitions to S_A and an error-detection event d_l is produced, optionally followed by error-recovery event r_l. The pattern for d_l and r_l is $\{e_1\}$.

Trend Checking. Isermann [17] proposed calculating trend checking by taking the first derivative of the event value $f'(e^v)$, and then limit check as before, e.g. $\neg(\epsilon_{min} \leq f'(e^v) \leq \epsilon_{max})$. If true, a trend has *not* been smooth, and thus can be considered unreasonable. We propose the NFA in (Figure 2b) for trend checking, which calculates the slope between all relevant events that occur within time window t. If the slope between two events e_1, e_m, or an aggregate of n prior events $f(e_1, ..., e_n), e_m$, surpasses a slope threshold, error events d_t, r_t are generated. Otherwise, e_m is ignored and the NFA reattempts with some future e_m event, or halts on state clearance. This design enables spike detection by checking for exceptionally large trend changes between events. Stuck-at detection occurs if the trend is persistently 0. The pattern for d_t and r_t is $\{e_1, ..., e_n, e_m\}$.

3.2.2 Timing. A timing check is a simple implementation that detects when an operation fails to satisfy a specified time bound, and typically uses absolute or interval timers to invoke the detection mechanism. These checks address three scenarios: (1) where there exist two events e, e' and an 'unacceptable' time interval ϵ between them; (2) where one event e exists and an unacceptable time elapse ϵ that occurs *without* the next event e'; and (3) where n events occur within some time bound ϵ. We explore these next.

Performance Checking. We want to identify timeliness errors such that an event beyond time threshold ϵ would produce error event d_e, representing a *performance failure* i.e. a late timing failure (Figure 2c). Event e_1 is first accepted, and a transition to S_A occurs if: $e_2^o = e_1^o \wedge (e_2^t - e_1^t) > \epsilon$. If $\leq \epsilon$, the NFA halts. To detect when a second event does not arrive at all, the CEP system still needs an e_2 event to reach S_A. A limitation with NFAs is that negation (Section 2) cannot be the final state transition i.e. it cannot reach S_A by waiting for something to *not* happen. In literature, pruning NFAs is accomplished using a periodically generated *null event*, $e^{\emptyset}$, that helps when reasoning about intervals between events [2]. Thus, the time between e_1 and $e_2 = e^{\emptyset}$ is calculated instead.

Persistence Checking. Persistence is classified as [3, 19]: (1) *transient*: arbitrary faults that cause erroneous behavior for a short time before going away; (2) *intermittent*: faults that oscillate between being active and dormant; and (3) *permanent*: faults assumed to be continuous in time. If the pattern of an error event d does reference some other error event d', then we consider d to be *transient*, because it is independent from any other detected system errors. In Figure 2d, we consider how d_e from Figure 2c can be checked for persistence. For intermittent and permanent persistence, the NFA accepts $n > 1$ events of type d_e to reach S_A. Halting occurs on state clearance. We propose that intermittent persistence be implemented as having $n > 1$ occurrences of d_e in time t, and permanent persistence as having $n' \geq n$ occurrences in time $t' \geq t$. The intuition behind this is that permanent persistence would have more error occurrences over more time than intermittent faults.

3.2.3 Reversal. A reversal check takes the output from a system and calculates what the input(s) should have been in order to produce that output, where the calculated inputs are used to compare with the actual inputs to check for an error [21]. This check has predominantly deterministic applications (e.g. reading back what was just written to disk). However, we consider how this check can be used in scenarios to check for a relationship between two (sets of) events.

Correlation Checking. We want to identify whether, given $n \geq 1$ system events $e_1, ..., e_n$, there were $n \geq 1$ erroneous events $e_{n+1}, ..., e_m$ that occurred afterwards within a given time frame, or

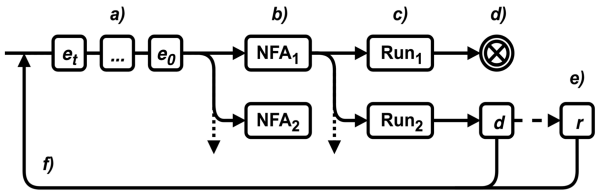

Figure 3: The process of CvR.

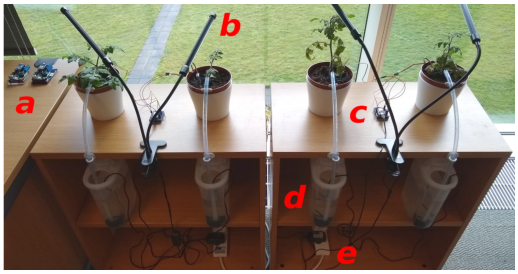

Figure 4: Our indoor agricultural system.

halt otherwise (Figure 2e). If they do occur, we can react as though the latter event(s) were *caused by* the former. This check helps to handle scenarios where an error *propagates* through a system and causes more errors [3].

4 COMPLEXITY VIA RECURSION

Instead of defining complex, monolithic NFAs to handle application-specific error scenarios, we want to define simple, modular, and reusable NFAs, where error-detection and recovery events produced by them are *recursively* fed back into the CEP system for use in other NFAs to express more complex scenarios. The process (Figure 3) is as follows:

(a) A stream of events enter the CEP system over time.
(b) Each event is passed to the NFAs. When an event fulfills the predicate to transition to the first state of an NFA, a copy of the NFA, called a *run*, is created.
(c) Events are passed to each incomplete and unhalted run and might cause a state transition.
(d) A run might eventually transition to S_A, producing an error event d, or halt if it transitions to S_F.
(e) Event d may be passed to an error-recovery handler that will attempt to recover from d, producing an error-recovery event r detailing the actions taken to handle the error and whether they were successful or not.
(f) Events d, r are fed back into the CEP data stream to potentially be used in other runs.

This approach to error definition means that common errors which are universal to any IoT system have standard error-check NFAs that can be used to detect them. This makes FT-support setup easier for system designers because they simply need to understand what failures can occur in their system and pair them with appropriate NFAs.

We consider limit, trend, and performance checks (Figures 2a-c) to be the most *simple*, resuable, and generic checks of the checks we have defined. For example, the three data faults from Section 3 (i.e. outliers, stuck-at faults, spikes) can be detected using limit and trend checking, and data loss can be detected using performance checks. These errors are generic and highly applicable to any IoT device on any IoT system, making them easy to implement as they are.

We consider persistence and correlation checks (Figures 2d-e) to be *complex* because they are more appropriate for detecting errors based upon prior composite events generated by other checks that have re-entered the system. For example, in our evaluation (Section 5), we detected data loss using a performance check and checked for device battery depletion by whether the data-loss events had been persistent using a persistence check. An even more complex scenario might be whether n devices all generated persistent-data-loss events within a short space of time. Correlating all of these via a correlation check might infer that the gateway to the n devices had failed and was causing a data blackout across the n devices, instead of individual hardware failures as previously inferred. This approach enables intermediary errors to be built upon to gain greater insight into complex system failures that are difficult to define as a single NFA. System designers have fine-grained control of how errors are correlated and handled because the FT-support system is able to handle errors separately and as a whole.

We propose that CvR is used as the basis for providing proactive-FT support via supervised learning techniques to predict imminent system error events using correlation check NFAs, as follows. The first NFA receives event e_1 and is followed by the second event $e_2 \in d$ within some time t, which reaches S_A and produces error event d_r, where d_r acts as a 'positive' predictive label 1 for the dataset of a supervised learning model. The second NFA generates event d'_r when e_2 does *not* occur after e_1 in time t (i.e. when $e_2 = e^{\emptyset}$), where d'_r acts as a 'negative' predictive label 0. With these, we wish to predict $\|P(e_2|e_1)\| = 1$ when e_2 is likely to follow e_1 in time t and proactively performing recovery as though $e_2 \in d$ had just been detected. Conversely, we wish to predict $\|P(\neg e_2|e_1)\| = 0$ when e_2 is unlikely to occur in time t. We apply this approach in our evaluation when we check for correlations between data-spike and persistent-data-loss errors, discussed next.

5 EVALUATION

We evaluated CPoF on an indoor, automated agriculture system. This was motivated by the growing trend of *vertical farms* that grow produce indoors, where environmental factors can be controlled to ensure a correct and efficient amount of light and water for produce [4]. Our objective was to see whether the combination of CEP and ML was effective in providing reactive and proactive FT support.

5.1 Case Study

Our system had two shelves, each with two plants (Figure 4). Beneath the plants were water containers (d) that pumped water to each plant. Grow lights (b) turned on when the room was dark, and off when bright. Microcontrollers (c) each had moisture sensors for each plant. Two multi-sensors, *DevicePrimary* and *DeviceBackup* (a), sent infrared-light data every 5 seconds to a Raspberry Pi 3 at

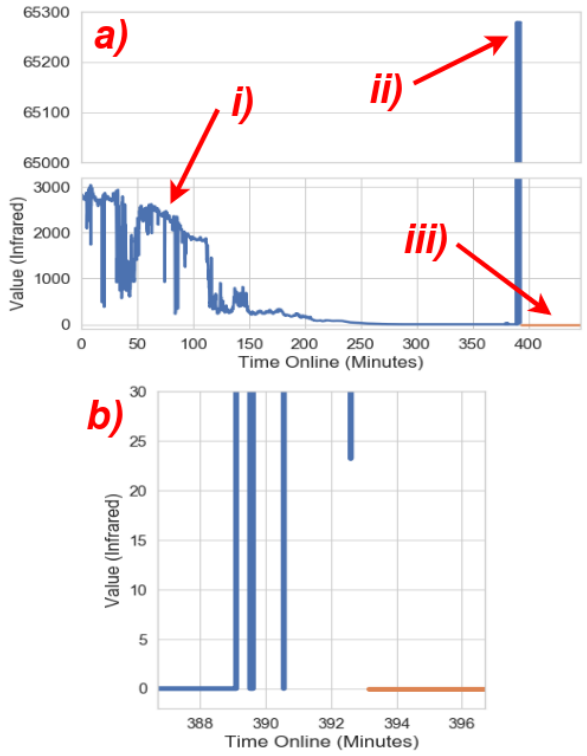

Figure 5: Infrared light data from DevicePrimary (blue) and DeviceBackup (orange): (a) data during the lifetime of DevicePrimary; and (b) a close-up of battery depletion and switchover to DeviceBackup.

the network edge. Water pumps and grow lights were controlled using smart plugs (e). If a moisture value were < 0.5, its associated water pump activated. If an infrared-light value were < 0.2, the grow lights switched on, or off if higher. We used *FlinkCEP* v1.4.2[1] for CEP and *scikit-learn* v0.2[2] for ML.

We focused on the failure scenario of sensor battery depletion and how we can infer its occurrence via CEP and predict its imminent occurrence via ML. Low battery voltage has been identified as a prevalent cause of data spikes in IoT sensors [7]; our multi-sensors exhibited this phenomenon in previous work [27]. We considered how temporal correlations, as well as correlations with other errors (i.e. data spikes) could be exploited to reactively and proactively handle power failures.

5.2 Experiment: Reactive FT

5.2.1 CEP Setup. To handle our failure scenario, we defined error automata for CEP. A **data spike** error was defined using the trend check (Figure 2b). As multi-sensor data was sent every 5 seconds, we considered the last 12 infrared-light events from DevicePrimary to get roughly the last minute's worth of data from the device. The first 11 events $e_1, \ldots e_{11}$ were consumed. On the twelfth event e_{12}, the values of the first 11 events were averaged, $avg = (\sum_{i=1}^{11} e_i^v)/11$, and compared with e_{12} as $(e_{12}^v - avg)/avg$ to compute the percentage by which e_{12}^v had increased beyond the average. When the increase was $\geq 150\%$, error d_t was generated.

A **data loss** error was defined using the performance check (Figure 2c). It first checked for event e_1 from DevicePrimary, followed by a null event $e_2 = e^{\emptyset}$ that occurred 15 seconds after e_1. If another DevicePrimary event were received in this time, the NFA would halt; otherwise error d_e was generated. The recovery r_e for d_e was to ping DevicePrimary. For every 15 seconds that elapsed without DevicePrimary data, another d_e, r_e was generated. If 3 unsuccessful r_e were generated without any successful pings or DevicePrimary data within 60 seconds, the **persistent data loss** error d_p was generated using a persistence check (Figure 2d), and recovery r_p would cause a switchover to DeviceBackup.

We also checked for **error correlations** using the correlation check (Figure 2e). Specifically, we checked whether a data spike $e_1 = d_t$ was followed by persistent data loss $e_2 = d_p$ on the same device within 10 minutes, which produced error d_r. Conversely, we defined another NFA to check when d_p did *not* follow d_t within 10 minutes (i.e. if $e_2 = e^{\emptyset}$ instead), which produced event d_r'.

Demonstration. We ran DevicePrimary on full charge until depletion, and monitored the infrared-light values generated by it. Throughout the day, the values fluctuated (Figure 5a-i) which produced several data-spike d_t errors. Minutes before battery depletion, the values would spike several times to 65279 (Figure 5a-ii). It also produced three unsuccessful data-loss recovery r_e events, which led to a persistent-data-loss d_p error and caused a switchover to DeviceBackup (Figures 5a-iii & 5b). The wait for d_p meant that failure recovery took approx. **45 seconds** to complete.

5.3 Experiment: Proactive FT

5.3.1 Time Correlation. Our first approach to predicting battery depletion correlated data-loss events with DevicePrimary's duration online. The intuition was that a data-loss event that occurred when DevicePrimary often failed was more likely to be persistent. Thus, the system could avoid a persistent-data-loss error by preempting the switchover instead.

We used a *support-vector machine* (SVM) for C-support vector classification. SVMs are linear classifiers that attempt to construct a maximum-margin hyperplane to optimally separate data into two categories to provide the best generalization capacity, and are suitable for binary classification problems such as ours [30]. We used the minutes online and elapse seconds as two features for this model (Figure 6a). Both features had been normalized before training the model because, with SVMs, normalization of feature vectors had been shown to lead to superior generalization performance [11]. The C hyperparameter was set to 1000.0 and was trained with 1000 iterations. The benefit of a SVM was that it provided the widest margin between data points in both classes and prevented the decision boundary from being too close to any particular data point, which might have provided unrealistic predictions.

Demonstration. DevicePrimary was activated on a full charge with the model running. It died at approx. 18:29:37 (Figure 6b). The device had been online for approx. 495 minutes (approx. 0.9 when

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/libs/cep.html
[2] https://scikit-learn.org/0.20/

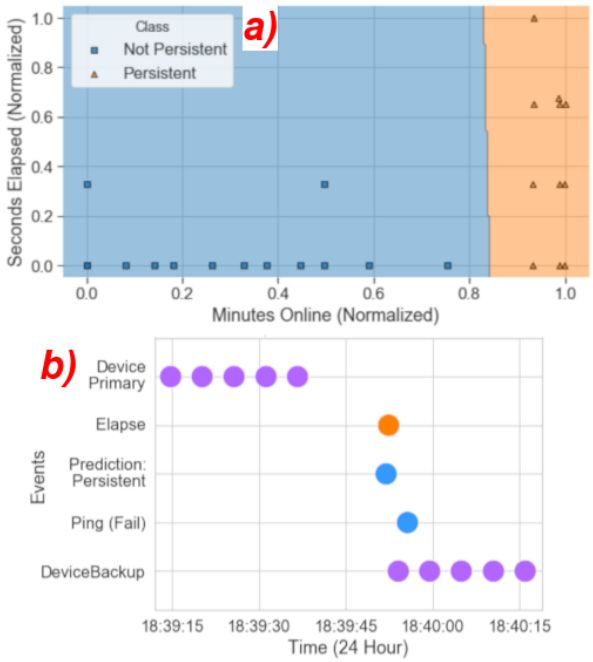

Figure 6: (a) Linear SVM model to predict battery depletion; and (b) a data loss error correctly identified as persistent.

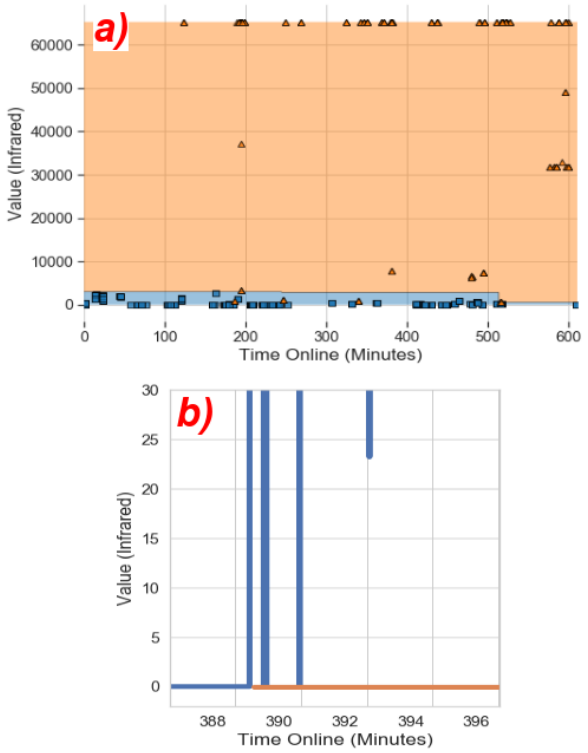

Figure 7: (a) RF model to predict battery depletion; and (b) a close-up of the switchover to DeviceBackup (orange) before battery depletion had occurred on DevicePrimary (blue).

normalized in Figure 6a). When a data-loss error (Figure 6b, orange circle) occurred 15 seconds after the last infrared-light value from DevicePrimary, the model predicted that this error would persist and preemptively switched over to DeviceBackup. This solution provided preemptive error detection that was approx. **30 seconds faster** than the reactive-FT solution. However, it was only effective at predicting failure when DevicePrimary was on a full charge. Next, we considered a complementary model that identified the same failure based on error correlations.

5.3.2 Error Correlation. Our second approach to predicting battery depletion correlated data-spike events with data-loss events. We wanted to exploit the data spikes caused by low battery voltage (Figure 5a-ii) by detecting them using error-correlation event d_r and executing a preemptive switchover to DeviceBackup before complete depletion of DevicePrimary's battery.

To build the model's dataset, we activated DevicePrimary 60 times with varying levels of initial battery charge to build a dataset of 473 total data spikes: 187 caused by malfunctions (class 1) and 286 *not* caused by malfunction i.e. instead caused by natural light fluctuations (class 0). We trialed several classification algorithms and decided on *Random Forest* (RF)[3] with 10 trees/estimators and a maximum tree depth of 2 (Figure 7a). Class 1 is shown in orange with its data instances as triangles and class 0 in blue with squares. We used RF because the decision boundary was not too curved and kept the threshold close to class 0, which was appropriate for this task. The model was trained and tested with an 80%-20/% randomized dataset split and achieved an accuracy of 97.89% and F1 score of 96.43%.

During data collection, we identified the following phenomena. As the battery neared depletion, the infrared values would often spike to unusually high values, predominantly 65279. It sometimes spiked within a 'reasonable' value range (i.e. 0-3500), which influenced the predictions in the model at around 510 minutes online (Figure 7a) because it caused the model to predict class 1 for spikes in the normal range if the device had been online for a long time.

Demonstration. We re-executed the DevicePrimary data from the reactive-FT experiment (Figure 5a) into the system in the same manner as the original, to be able to compare how much faster proactive-FT support was to the reactive-FT solution. During execution, many of the early data spikes (Figure 5a-i) were correctly predicted as being in class 0 because they did not pass the decision boundary that sits approx. at 3500 (Figure 7a). However, on the first spike to 65279 (Figure 5a-ii), the model predicted class 1, and triggered a switchover to DeviceBackup, leading to the early introduction of its data before DevicePrimary had fully depleted (Figure 7b).

This solution provided preemptive migration that was approx. **45 seconds** faster than the reactive-FT solution and **30 seconds** faster compared to the previous proactive-FT solution. However, it was only effective at predicting failure if a prior data spike error

[3]https://scikit-learn.org/0.20/modules/generated/sklearn.ensemble.RandomForestClassifier.html

occurred, which was not always guaranteed to happen near battery depletion.

6 CONCLUSION AND FUTURE WORK

FT support is a key challenge for ensuring dependable IoT systems, with many existing implementations being static, tightly coupled, and inflexible. We proposed CPoF, an approach to provide reactive and proactive FT support in IoT, where error detection was defined using NFA that were implemented in CEP systems, and errors could be learned from and predicted over time using ML. We evaluated CPoF on an indoor agriculture system and considered how it could be used to provide reactive- and proactive-FT support, where time and error correlations were used to train predictive models to preempt battery-depletion failures and provide preemptive error recovery. In future work, we will expand our taxonomy of error-detection NFAs to establish a generic framework for the easy and adaptive FT implementation in IoT systems.

REFERENCES

[1] Adnan Akbar, Francois Carrez, Klaus Moessner, and Ahmed Zoha. 2015. Predicting complex events for pro-active IoT applications. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. IEEE, 327–332.

[2] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. 2012. Stream reasoning and complex event processing in ETALIS. *Semantic Web* 3, 4 (2012), 397–407.

[3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

[4] Jan Bauer and Nils Aschenbruck. 2018. Design and implementation of an agricultural monitoring system for smart farming. In *IoT Vertical and Topical Summit on Agriculture-Tuscany (IOT Tuscany), 2018*. IEEE, 1–6.

[5] Dario Bruneo and Salvatore Distefano. 2015. *Quantitative assessments of distributed systems: Methodologies and techniques*. John Wiley & Sons.

[6] Alejandro Buchmann and Boris Koldehofe. 2009. Complex event processing. *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* 51, 5 (2009), 241–242.

[7] Tusher Chakraborty, Akshay Uttama Nambi, Ranveer Chandra, Rahul Sharma, Manohar Swaminathan, Zerina Kapetanovic, and Jonathan Appavoo. 2018. Fallcurve: A novel primitive for IoT Fault Detection and Isolation. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 95–107.

[8] Prafulla Kumar Choubey, Shubham Pateria, Aseem Saxena, Vaisakh Punnekkattu Chirayil SB, Krishna Kishor Jha, and Sharana Basaiah PM. 2015. Power efficient, bandwidth optimized and fault tolerant sensor management for IOT in Smart Home. In *2015 IEEE International Advance Computing Conference (IACC)*. IEEE, Bangalore, 366–370. https://doi.org/10.1109/IADCC.2015.7154732

[9] Gaurav Choudhary and A.K. Jain. 2016. Internet of Things: A survey on architecture, technologies, protocols and challenges. In *2016 International Conference on Recent Advances and Innovations in Engineering (ICRAIE)*. IEEE, Jaipur, 1–8. https://doi.org/10.1109/ICRAIE.2016.7939537

[10] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. 2014. A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 10, 1 (2014), 8.

[11] Sven F Crone, Jose Guajardo, and Richard Weber. 2006. The impact of preprocessing on support vector regression and neural networks in time series prediction.. In *DMIN*. 37–44.

[12] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 15. https://doi.org/10.1145/2187671.2187677

[13] Gianpaolo Cugola and Alessandro Margara. 2015. *The Complex Event Processing Paradigm*. Springer International Publishing, Cham, 113–133. https://doi.org/10.1007/978-3-319-20062-0_6

[14] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing* 65, 3 (2013), 1302–1326.

[15] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Kamp, and Michael Mock. 2017. Issues in complex event processing: Status and prospects in the big data era. *Journal of Systems and Software* 127 (2017), 217–236.

[16] Yung-Li Hu, Yuo-Yu Cho, Wei-Bing Su, David S.L. Wei, Yennun Huang, Jiann-Liang Chen, Ing-Yi Chen, and Sy-Yen Kuo. 2015. A Programming Framework for Implementing Fault-Tolerant Mechanism in IoT Applications. In *Algorithms and Architectures for Parallel Processing*, Guojun Wang, Albert Zomaya, Gregorio Martinez, and Kenli Li (Eds.). Springer, Zhangjiajie, 771–784.

[17] Rolf Isermann. 2006. *Fault-Diagnosis Systems: An Introduction from Fault Detection to Fault Tolerance*. Springer Berlin Heidelberg.

[18] Surabhi Abhimithra Karthikeya, J. K. Vijeth, and C. Siva Ram Murthy. 2016. Leveraging Solution-Specific Gateways for cost-effective and fault-tolerant IoT networking. In *2016 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, Doha. https://doi.org/10.1109/WCNC.2016.7564811

[19] Israel Koren and C. Mani Krishna. 2010. Preliminaries. In *Fault-Tolerant Systems*. Morgan Kaufmann, Chapter 1, 400.

[20] In Lee and Kyoochun Lee. 2015. The Internet of Things (IoT): Applications, investments, and challenges for enterprises. *Business Horizons* 58, 4 (2015), 431–440.

[21] Peter A. Lee and Thomas Anderson. 2012. *Fault Tolerance: Principles and Practice*. Springer Vienna.

[22] Yu Liu, Yang Yang, Xiaopeng Lv, and Lifeng Wang. 2013. A self-learning sensor fault detection framework for industry monitoring IoT. *Mathematical problems in engineering* 2013 (2013).

[23] Mohammad Saeid Mahdavinejad, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi, and Amit P Sheth. 2018. Machine learning for Internet of Things data analysis: A survey. *Digital Communications and Networks* 4, 3 (2018), 161–175.

[24] Mohsen Marjani, Fariza Nasaruddin, Abdullah Gani, Ahmad Karim, Ibrahim Abaker Targio Hashem, Aisha Siddiqa, and Ibrar Yaqoob. 2017. Big IoT data analytics: architecture, opportunities, and open research challenges. *IEEE Access* 5 (2017), 5247–5261.

[25] Victoria Moreno-Cano, Fernando Terroso-Saenz, and Antonio F Skarmeta-Gómez. 2015. Big data for IoT services in smart cities. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 418–423.

[26] Mukosi Abraham Mukwevho and Turgay Celik. 2018. Toward a smart cloud: A review of fault-tolerance methods in cloud systems. *IEEE Transactions on Services Computing* (2018).

[27] Alexander Power and Gerald Kotonya. 2018. A Microservices Architecture for Reactive and Proactive Fault Tolerance in IoT Systems. In *2018 IEEE 19th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. IEEE, 588–599.

[28] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM, 4.

[29] Omer Berat Sezer, Erdogan Dogdu, and Ahmet Murat Ozbayoglu. 2018. Context-aware computing, learning, and big data in internet of things: a survey. *IEEE Internet of Things Journal* 5, 1 (2018), 1–27.

[30] Uday Shankar Shanthamallu, Andreas Spanias, Cihan Tepedelenlioglu, and Mike Stanley. 2017. A brief survey of machine learning methods and their sensor and IoT applications. In *2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA)*. IEEE, 1–8.

[31] Penn H. Su, Chi-Sheng Shih, Jane Yung-Jen Hsu, Kwei-Jay Lin, and Yu-Chung Wang. 2014. Decentralized fault tolerance mechanism for intelligent IoT/M2M middleware. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE, Seoul, 45–50. https://doi.org/10.1109/WF-IoT.2014.6803115

[32] Itorobong S Udoh and Gerald Kotonya. 2018. Developing IoT applications: challenges and frameworks. *IET Cyber-Physical Systems: Theory & Applications* 3, 2 (2018), 65–72.

[33] IM Umesh and G N Srinivasan. 2017. Dynamic software aging detection-based fault tolerant software rejuvenation model for virtualized environment. In *Proceedings of the International Conference on Data Engineering and Communication Technology*. Springer, 779–787.

[34] Yongheng Wang and Kening Cao. 2014. A proactive complex event processing method for large-scale transportation internet of things. *International Journal of Distributed Sensor Networks* 10, 3 (2014), 159052.

[35] Yongheng Wang, Hui Gao, and Guidan Chen. 2018. Predictive complex event processing based on evolving bayesian networks. *Pattern Recognition Letters* 105 (2018), 207–216.

[36] Min Woo Woo, JongWhi Lee, and KeeHyun Park. 2018. A reliable IoT system for personal healthcare devices. *Future Generation Computer Systems* 78 (2018), 626–640.

[37] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 217–228.