A Predictive Fault-Tolerance Framework for IoT Systems

Alexander Power

School of Computing and Communications

Lancaster University

Submitted in partial fulfilment of the requirements for the
Degree of Doctor of Philosophy
in Computer Science

Supervisor Dr Gerald Kotonya

August 2020

For A.

Acknowledgements

I would like to thank my supervisor for his invaluable support throughout my many years at Lancaster University. His sense of humour and unabating enthusiasm for software engineering have provided great joy and motivation. I would also like to thank Andrew Cumming for generously contributing key data used in the evaluation of my research. Finally, I would like to thank the *Engineering and Physical Sciences Research Council* (EPSRC) for the funding that I received to complete my research.

Declaration

This thesis is my own work and has not been submitted in any form for the award of a higher degree elsewhere. The work has been carried out under the supervision of Dr Gerald Kotonya of the School of Computing and Communications at Lancaster University.

Alexander Power

11th August 2020

Related Publications

A. Power and G. Kotonya, '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, 2018, pp. 588–599

A. Power and G. Kotonya, 'Complex patterns of failure: Fault tolerance via complex event processing for iot systems,' in *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, IEEE, 2019, pp. 986–993

A. Power and G. Kotonya, 'Providing fault tolerance via complex event processing and machine learning for iot systems,' in *Proceedings of the 9th International Conference on the Internet of Things*, Bilbao, Spain: ACM, 2019, 1:1–1:7

Abstract

As *Internet of Things* (IoT) systems scale, attributes such as availability, reliability, safety, maintainability, security, and performance become increasingly more important. A key challenge to realise IoT is how to provide a dependable infrastructure for the billions of expected IoT devices. A dependable IoT system is one that can defensibly be trusted to deliver its intended service within a given time period.

To define a FT-support solution that is applicable to all IoT systems, it is important that error definition is a generic, language-agnostic process, so that FT can be applied as a software pattern. It must also be interoperable, so that FT support can be easily 'plugged into' any existing IoT system, which is facilitated by an adherence to standards and protocols. Lastly, it is important that FT support is, itself, fault tolerant, so that it can be depended on to provide correct support for IoT systems.

The work in this thesis considers how real-time and historical data analysis techniques can be combined to monitor an IoT environment and analyse its short- and long-term data to make the system as resilient to failure as possible. Specifically, *complex event processing* (CEP) is proposed for real-time error detection based on the analysis of stream data in an IoT system, where errors are defined as *nondeterministic finite automata* (NFA). For long-term error analysis, *machine learning* (ML) is proposed to predict when an error is likely to occur and mitigate imminent system faults based on previous experience of erroneous system behaviour in the IoT system.

The contribution is threefold: (1) a language-agnostic approach to error definition using NFAs, designed to provide 'FT as a service' for easy deployment and integration into existing IoT systems; (2) an implementation of NFAs on a bespoke CEP system, BoboCEP, that provides distributed, resilient event processing at the network edge via active replication; and (3) a ML approach to intelligent FT that can learn from

system errors over time to ensure correct long-term FT support. The proposed solution was evaluated using two vertical-farming testbeds and a dataset from a real-world vertical farm. Results showed that the proposed solution could detect and predict the successful detection and recovery of erroneous system behaviours. A performance analysis of BoboCEP was conducted with favourable results.

Table of Contents

1 Introduction **1**

1.1 Problem Statement 1

1.2 Key Issues and Challenges 4

1.2.1 Research Questions 5

1.3 Objectives 5

1.4 Significant Contribution to Thesis 7

1.5 Thesis Structure 10

2 Fault Tolerance **12**

2.1 Dependability 12

2.1.1 Attributes 13

2.1.2 Threats 15

2.1.3 Means 20

2.2 Stages 21

2.2.1 Detection 21

2.2.2 Assessment 30

2.2.3 Recovery 31

2.2.4 Treatment 34

2.3 Summary 35

3 Fault Tolerance in IoT **36**

3.1 Review 36

3.1.1 Self-Learning Group-Based Fault Detection (SGFD) 36

3.1.2 Services Choreography (SerCho) 38

3.1.3 IoT/M2M Middleware (M2M-Mid) 39

3.1.4 Smart Control Algorithm (SCA) 40

3.1.5 6LoWPAN Health Monitoring (6LoW-HM) 41

3.1.6 FT Programming Framework (FTPF) 42

3.1.7 Cloud Application Placement Problem (CAPP) 43

3.1.8 UbiFlow 44

3.1.9 Modalities via Virtual Services (ModVS) 45

3.1.10 Z_{ij}-Routing . . . 46
3.1.11 Modalities for Graceful Degradation (ModGD) . . . 48
3.1.12 NewIoTGateway-Select (NIoTGS) . . . 49
3.1.13 Smart Cities Architecture (SmartCA) . . . 49
3.1.14 Privacy-Preserving Data Aggregation (PPDA) . . . 51
3.1.15 CEFIoT . . . 52
3.1.16 Personal Healthcare Devices (PerHD) . . . 53
3.2 Summary . . . 55
3.2.1 Failures . . . 55
3.2.2 Vulnerabilities . . . 56
3.2.3 Error Detection . . . 58
3.2.4 Error Recovery . . . 61
3.2.5 Evaluation . . . 61

4 Fault-Tolerance Framework Design 64
4.1 Fault-Tolerance Architecture . . . 64
4.1.1 Microservices . . . 68
4.1.2 Error Events . . . 72
4.1.3 Scenario . . . 74
4.2 Vulnerabilities, Faults, and Failures . . . 75
4.2.1 Attributes . . . 76
4.2.2 Applicability . . . 77
4.3 Summary . . . 79

5 Complex Patterns of Failure 81
5.1 Complex Event Processing . . . 81
5.1.1 Nondeterministic Finite Automata . . . 83
5.1.2 Language Specification . . . 83
5.1.3 Context Awareness . . . 85
5.2 Error-Detection Automata . . . 86
5.2.1 Automata Model . . . 86
5.2.2 Checks . . . 87
5.3 Complexity via Recursion . . . 92
5.3.1 Proactive Detection . . . 93
5.4 Summary . . . 95

6 Implementation 96
6.1 BoboCEP . . . 96

6.1.1 Event Processing . . . 97
6.1.2 Architecture . . . 99
6.1.3 Message Broker . . . 103
6.2 Vertical Farming Systems . . . 104
6.2.1 Small-Scale Testbed . . . 105
6.2.2 Medium-Scale Testbed . . . 107
6.2.3 Real-World Dataset . . . 109
6.3 Summary . . . 110

7 Evaluation **111**
7.1 Methodology . . . 111
7.1.1 Techniques . . . 111
7.1.2 Justification for Evaluations . . . 112
7.1.3 Predictive Models . . . 114
7.2 Experiments: Reactive FT . . . 118
7.2.1 Scenario: Empty Water Container . . . 118
7.2.2 Scenario: Data Transmission Degradation . . . 119
7.3 Experiments: Proactive FT via Correlations . . . 120
7.3.1 Scenario: Battery Depletion . . . 121
7.3.2 Time Correlation . . . 123
7.3.3 Error Correlation . . . 125
7.4 Experiments: FT Requirements . . . 126
7.4.1 Detect, Assess, Recover, Analyse . . . 127
7.4.2 Requirements . . . 129
7.4.3 W1: Reservoir Water Depletion . . . 130
7.4.4 L1: Natural Light Fluctuation . . . 135
7.4.5 P1: Pump Filter Blocked . . . 138
7.5 Experiments: BoboCEP . . . 141
7.5.1 FT Scenario . . . 143
7.5.2 Performance . . . 145
7.6 Summary . . . 146

8 Conclusion **148**
8.1 Objectives Revisited . . . 148
8.2 Reflection . . . 151
8.2.1 Limitations . . . 151
8.2.2 Lessons Learned . . . 152
8.3 Future Work . . . 153

8.4 Final Remarks 155

References **157**

A Code Listings **177**

A.1 Shared Versioned Match Buffer 177

A.2 Match Event 183

A.3 Run Version 185

B Testbed Design **189**

B.1 Smart Plugs 189

B.1.1 KanKun KK-SP3 189

B.1.2 TP-Link HS100 190

B.2 Actuators 192

B.3 Microcontrollers 193

List of Figures

1.1 The FT-support subunits are 'plugged into' an existing IoT system, to learn how the system fails, and execute actions on the system to provide reactive- and proactive- FT support. 7

2.1 The dependability tree from [17], with FT highlighted in red. 13
2.2 A visualisation of (a) transient, (b) intermittent, and (c) permanent fault persistence over time t. Orange circles represent an active fault, and blue circles represent a fault having transitioned to dormancy. . 17
2.3 An example scenario of error propagation between two components, adapted from Figure 10 in [16]. Fault activation is shown with a solid arrow, and error propagation with a dashed arrow. 20
2.4 The Recovery Block approach. 23
2.5 The N-Version Programming approach. 24
2.6 The N Self-Checking Programming approach. 26

3.1 An example scenario of the decentralised FT mechanism proposed by Su et al. [186]. 39
3.2 An example scenario of the smart home architecture proposed by Choubey et al. [51]. 41
3.3 An example scenario of the gateways in the IoT system proposed by Woo et al. [208]. 54

4.1 The generic 5-tier IoT architecture (left) and corresponding hardware components (middle) that define the infrastructure of IoT systems. Availability, resilience, and security are applicable across the IoT system infrastructure (right). 65
4.2 A reference microservices architecture. 66
4.3 The interfaces between IoT devices and the four proposed microservices. 68

4.4 The architecture of the Real-Time FT microservice. Purple arrows indicate the flow of properties data, yellow for detected errors, blue for error assessments, red for error recovery actions, and black for internal data. 70
4.5 The architecture of the Predictive FT microservice. Arrow colours are the same as in Figure 4.4. 71
4.6 Device1 and Device2 used in the example scenario. 74
4.7 Data showing four properties over a six minute period on Device1 (left) and Device2 (right). Device1 experiences a stuck-at fault from 21:46 onwards, whereas Device2 is running as normal at that time. . 74

5.1 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 run acceptance; (d) a starting point; (e) a state; (f) an accepting final state; (g) a non-accepting halt state. 86
5.2 Reasonableness NFAs: (a) limit checking; and (b) trend checking. . . 88
5.3 Timing NFAs: (a) performance checking; and (b) persistence checking. 89
5.4 Reversal NFA for correlation checking. 90
5.5 Replication NFA for inconsistency checking. 91
5.6 The process of CvR. 92
5.7 The value of generating a composite error-detection event over time when faults and errors occur in a system. 94

6.1 The relationship between automaton states (S_1, ..., S_A) and their labels (L_1, ..., L_6). 98
6.2 An example of run processing, adapted from Figure 2 in [60]. 99
6.3 The BoboCEP architecture, with the key subsystems from the IFP architecture [59] in blue. 100
6.4 Handlers (red) and runs (blue) in Decider. Handler h_1 currently contains runs $r_1, ..., r_n$ and can create more runs at runtime. Solid arrows show events passing through handlers and runs, and the dashed arrow indicates a run `CLONE` operation. 101
6.5 An example scenario of the SVMB, with run r_1 in red, r_2 in orange, r_3 in blue, r_4 in green, and r_5 in pink. 102
6.6 An example NFA with different transition types: (a,d) `CLONE`; (b) `TRANSITION`; (c) `HALT`; and (e) `FINAL`. 103

6.7 *Left*: The small-scale VFS testbed. *Right*: The testbed's infrastructure. . . . 106
6.8 *Left*: The medium-scale VFS testbed. *Right*: The testbed's infrastructure and its default configuration, where Shelf*n*'s microcontrollers (orange) send data to Edge*n*. . . . 107
6.9 Current data over a 9-day period. . . . 109

7.1 Moisture data from the four moisture sensors of the small-scale testbed, represented by different colours. . . . 119
7.2 LDR data from the two microcontrollers of the small-scale testbed (orange, green), and infrared data from Device1 (blue) and Device2 (red). . . . 120
7.3 Infrared light data from Device1 (blue) and Device2 (orange) during the lifetime of Device1. . . . 121
7.4 Infrared light data from Device1 (blue) and Device2 (orange), with a close-up of battery depletion and switchover from Device1 to Device2. 122
7.5 Time correlation: linear SVM model to predict persistent (orange) and non-persistent (blue) data loss. . . . 123
7.6 Time correlation: data loss error correctly identified as persistent. . . 124
7.7 Error correlation: RF model to predict persistent (orange) and non-persistent (blue) data loss. . . . 124
7.8 Error correlation: a close-up of the switchover to Device2 (orange) before battery depletion had occurred on Device1 (blue). . . . 126
7.9 The DARA process. System data are blue circles, along with error detection d, assessment a, and recovery r events. . . . 128
7.10 W1.DAR: water level data from a reservoir over a 7-day period, with a water-pump activation threshold at 500 (red line). . . . 130
7.11 W1.PRE: slope of water level data before pump activation, showing reasonable (blue) and erroneous (other) data trends. . . . 133
7.12 W1.PRE: KNN model to predict reasonable (blue) and erroneous (orange) data trends. . . . 134
7.13 W1.POST: slope of water level data after pump activation, showing reasonable (blue) and erroneous (orange) data trends. . . . 134
7.14 W1.POST: RFC model to predict reasonable (blue) and erroneous (orange) data trends. . . . 135
7.15 L1.DAR: light intensity data over a 24-hour period. . . . 135

7.16 L1.PRE: slope of reasonable (blue) and unreasonable (orange) light intensity data from 12pm to 12am (not including data from when lights were activated). 136
7.17 L1.PRE: RFR model to predict expected light intensity at a given hour of day (orange). 137
7.18 L1.POST: slope of light intensity data after light activation, showing reasonable (blue) and erroneous (other) data trends. 137
7.19 L1.POST: KNN model to predict reasonable (blue) and erroneous (orange) data trends. 138
7.20 P1.DAR: current data over a 9-day period. 138
7.21 P1.PRE: rolling average of current data since last filter clean, showing reasonable (blue) and erroneous (orange) trends, and their average slopes (red lines). 139
7.22 P1.PRE: RFC model to predict reasonable (blue) and erroneous (orange) data trends. 140
7.23 P1.POST: rolling average of current data since last filter clean, showing reasonable (blue) and erroneous (orange) data trends. 140
7.24 P1.POST: RFC model to predict reasonable (blue) and erroneous (orange) data trends. 141
7.25 Water level data from a reservoir over a 7-day period. 143
7.26 Light intensity data from Edge1 (green), Edge2 (orange), and Edge3 (blue). 144
7.27 Data from the run instantiation experiment. 145
7.28 Data from the rule throughput experiment. 146

B.1 The (a) KanKun KK-SP3 and (b) TP-Link HS100 smart plugs. . . . 189
B.2 *Left*: grow lights activated when natural light decreased in the evening. *Right*: plastic tubing connecting a water pump to its reservoir. . 193
B.3 A microcontroller connected to a reservoir, plant pot, and shelf. . . . 194
B.4 The configuration of the microcontrollers. 194

List of Tables

3.1 The failures addressed in the reviewed FT solutions (checkmarked). Little focus had been on data-centric i.e. response value failures. . . 56
3.2 The vulnerabilities identified in the reviewed FT solutions (checkmarked). Context-aware, data-centric FT was not prevalent in the reviewed solutions, and so environment vulnerabilities were scarcely addressed. 57
3.3 The error-detection checks used in the reviewed FT solutions (checkmarked). 59
3.4 The error-recovery mechanisms used in the reviewed FT solutions (checkmarked). 60
3.5 The technologies and evaluations used in the reviewed FT solutions (checkmarked). 62

4.1 The data transmitted when sending a property using the **/properties** interface. 69
4.2 The data transmitted when sending an error detection event using the **/events/errors/detect** interface. 72
4.3 The data transmitted when sending an error assessment event using the **/events/errors/assessment** interface. 73
4.4 Applicability between vulnerabilities, faults, and failures (checkmarked). 78

6.1 A sample of five rows from the IGS dataset. 108

7.1 Models trained in the experiments from Section 7.3, with the selected models in bold. 126
7.2 Failures and their error events to detect, assess, and recover from. . . 131
7.3 Pre-Detect and Post-Recover analyses of the failures in Table 7.2. . . 132
7.4 Models trained in the experiments from Section 7.4, with the selected models in bold. 142

Chapter 1

Introduction

1.1 Problem Statement

The term *Internet of Things* (IoT) has a broad definition that spans a wide range of applications. It extends the conventional concept of the Internet to a concept of interconnected objects forming pervasive computing environments, where a variety of heterogeneous sensor devices are connected with the intention of mining the data which they generate [64, 74]. The purpose of IoT is to connect all of these devices, such as *radio-frequency identification* (RFID) tags, infrared sensors, *Global Positioning System* (GPS) sensors and laser scanners, to the Internet. This allows systems to identify, locate, track, monitor items in real time, to achieve greater value and services in domains such as logistics, healthcare, and agriculture [113, 121, 23]. The surge in the number of Internet-connected devices is growing at an exponential rate, which is contributing to ever-increasing, massive data volumes that require real-time analytics in order to extract business value [201].

An important challenge to realise IoT is how to provide a *dependable* infrastructure for billions of devices and deliver their intended services without failing in unexpected and catastrophic ways [169, 52]. For IoT systems to deliver service on this scale, they must be able to provide long-term dependability despite: (1) environmental uncertainty e.g. ageing components and changes in system context; (2) functional changes e.g. dynamic changes to applications and requirements; and (3) technological changes e.g. new components, devices, interfaces, and protocols [181, 167]. It must deliver service that can justifiably be trusted, encompassing attributes such as

availability, reliability, safety, and maintainability, where *reliability* is a high-priority goal to address for IoT solutions concerned with *quality of service* (QoS) [207].

Resilience is central to effective and scalable IoT systems. System resilience is defined by its ability: (1) *to resist* external perturbations and internal failures; (2) *to recover* and enter stable state(s); and (3) *to adapt* its structure and behaviour to constant change [63]. One mechanism that can provide dependability and address resilience is *fault tolerance* (FT), which considers how to deliver correct service at runtime in the presence of system faults by implementing techniques during system runtime [163, 16]. Such techniques provide a means of improving system availability and reliability, which are key dependability attributes that can be improved using FT methods specifically. FT can be considered a subset of system *survivability*, which is a resilience concern that considers how hardware/software redundancy and diversity help to mitigate the effects of correlated failures, such as an attack by an intelligent adversary, or failures to large parts of the network infrastructure [184].

FT has four key responsibilities [122]: (1) to identify system errors caused by some underlying fault(s); (2) to assess the damage caused by the fault(s); (3) to perform actions that move the system into an error-free state; and (4) to provide system treatment to mitigate the probability of the fault(s) recurring. There are two key methods for providing FT [115, 144]:

- *Reactive FT*, where system recovery is initiated *after* an error has been detected, to decrease the influence of fault activation on the system.
- *Proactive FT*, where system recovery is initiated *before* an error has occurred, by continuously monitoring the system and conducting fault predictions to prevent their effects on the system.

Sezer et al. [181] identified that both rule-based and supervised learning approaches were commonly used for context-based reasoning in IoT systems. *Complex event processing* (CEP) is used in research and industry to identify complex situations (i.e. *composite events*) by defining rule-based patterns in stream data (i.e. *primitive events*). It is considered the paradigm of choice for monitoring and reactive applications [36, 60], 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 [135]. 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 [138]. These attributes are useful for providing data-centric FT support to identify and anticipate erroneous system behaviours for proactive FT support.

Despite the benefits of FT on improving system resilience, IoT presents several challenges that make it difficult to apply conventional FT support effectively. IoT systems suffer from failures akin to conventional distributed systems, namely [73]: (1) *crash* failures, where a server halts and requires a restart; (2) *omission* failures, where a server stops sending and receiving messages; (3) *timing* failures, where a server's response is too early or too late; (4) *response* failures, where a server's response value or state transition that takes place is incorrect; and (5) *arbitrary* failures, where the root cause is unclear. However, the nature of these failures is greatly influenced by factors unique to IoT systems, as follows.

- Failures are exacerbated because IoT devices are typically constrained (e.g. in terms of energy, computing power, and resources) and rely on wireless communication. This limits their ability to survive 'in the wild' or perform complex recovery strategies when faults manifest, meaning that FT is better delegated to external and more reliable entities, e.g. the fog or cloud [30].

- IoT systems are expected to continuously evolve to handle new services, features, and devices that had not been anticipated during initial system development, making it difficult to specify adequate error detection and recovery mechanisms *a priori*. The distribution of state and responsibility allows distributed systems to be robust and survive a variety of failures, however achieving such FT requires developers to reason through complex failure modes [26]. 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 to ensure that FT remains future proof [197].

- The scale of IoT devices is unprecedented. The number of devices that need to be managed and that communicate with each other is an order of magnitude greater than devices connected to the current Internet. Even more critical is data management and its interpretation for application purposes (i.e. *data semantics*) as well as efficient data handling [155].

1.2 Key Issues and Challenges

Existing FT solutions in literature suffer from many drawbacks that prevent them from providing effective FT support in IoT, namely:

- **Current FT implementations in IoT are static, tightly coupled, and inflexible**. For example: (1) they are designed for bespoke architectures [186, 86] and applications e.g. healthcare [208]; (2) they do not scale beyond small (decentralised) solutions [186, 116]; and (3) they only provide solutions for specific faults e.g. component failures [96] and communication link failures [103]. This limits the effectiveness of their proposed solutions in addressing emergent faults posed by the environmental uncertainty that is typical of most IoT systems.
- **Many FT solutions in IoT are already widely explored in distributed systems**. For example: (1) hardware redundancy, such as active-active (i.e. load-sharing) redundancy across two data sinks [86]; (2) checkpointing [96], where a snapshot of sensor data is stored for backward error recovery; and (3) data traffic rerouting when gateway failure occurs [103]. IoT must go beyond 'traditional' FT recovery mechanisms and further consider the importance of context awareness in FT, because IoT systems are highly associated with their physical environments. For example, context awareness can help to optimise waste management by installing level sensors on waste bins, so that waste trucks can optimise routes to reduce fuel consumption [140]. Moreover, it is important to infer context *accurately* because incorrect context can potentially lead to inappropriate resource usage, user annoyance, or system failure [47].

- **IoT systems rely on highly fallible, heterogeneous, and geographically dispersed low-level sensor networks**. With IoT emerges new data sources that had not existed before and, as IoT rises in popularity, the main challenge will be how many thousands, millions, or billions of devices can be managed despite their reliance on battery power, which make them vulnerable to frequent power outages. FT support must be able to scale in order to cope with this demand, so that *big data* is able to be reliably collected and rapidly processed to ensure fast decision making and achieve real-time business insights [93]. Furthermore, FT support will need to communicate and cooperate with these devices using a wide array of standards in order to accurately understand system state and provide effective FT.

1.2.1 Research Questions

These challenges lead to four key research questions:

1. How can fault events and patterns be identified and classified in IoT systems?
2. How can an effective, pluggable FT framework be developed that is able to mitigate and learn from fault events and fault patterns?
3. How can the FT framework be extended to incorporate dynamic strategies and mechanisms that facilitate effective fault mitigation?
4. How can the FT framework be designed to support scalability?

1.3 Objectives

The aim of the work in this thesis is to address the challenges described in Section 1.2 by proposing a predictive-FT framework that is able to provide reactive FT that can handle errors in real time, and also to learn from system errors in order to then proactively handle imminent errors and harness the value of streaming IoT data, which enables FT support to preempt and mitigate system errors before they are able to cause damage to the system. An experimental and heuristic approach will be adopted to test the objectives of this research, which are as follows:

1. **To identify and classify faults events and fault patterns in IoT systems**. For a system to be resilient to faults, they must be understood properly. The first objective is to have a thorough review of the different faults which can occur in IoT systems and classify them so that patterns can be identified on what are the root cause(s) and effect(s) that the faults may have on an IoT system. A review of existing and emerging FT solutions in IoT is the starting point for developing a comprehensive fault taxonomy.

2. **To develop a service-oriented fault-tolerance framework for IoT systems that combines both reactive- and proactive-FT support**. The aim of this objective is to investigate the provision of reactive- and proactive-FT *as services* to allow for changeable IoT system context and interactions. Microservices are used to develop and integrate a reactive-FT approach based on CEP to detect errors in real-time, and a proactive-FT approach via ML to predict whether known errors are likely to occur imminently.

3. **To incorporate strategies and mechanisms that facilitate effective fault mitigation**. The aim of this objective is to perform an extensive review of strategies to resolve system errors that are reactively and proactively detected in the second objective. The outcome of the review will inform the development and integration of strategies and mechanisms for mitigating faults.

4. **To ensure that the framework can scale to more complex IoT scenarios**. The aim of this objective is to ensure that the system design is one that is able to scale from small-scale to large-scale IoT systems. Microservices are able to be replicated and repositioned across the IoT infrastructure when necessary, which facilitates resilient FT support. The prior objectives would initially be tested within a controlled, small-scale environment. However, as large-scale IoT systems are equally viable in real-world solutions, it is important to assess whether the framework is able to cope at scale.

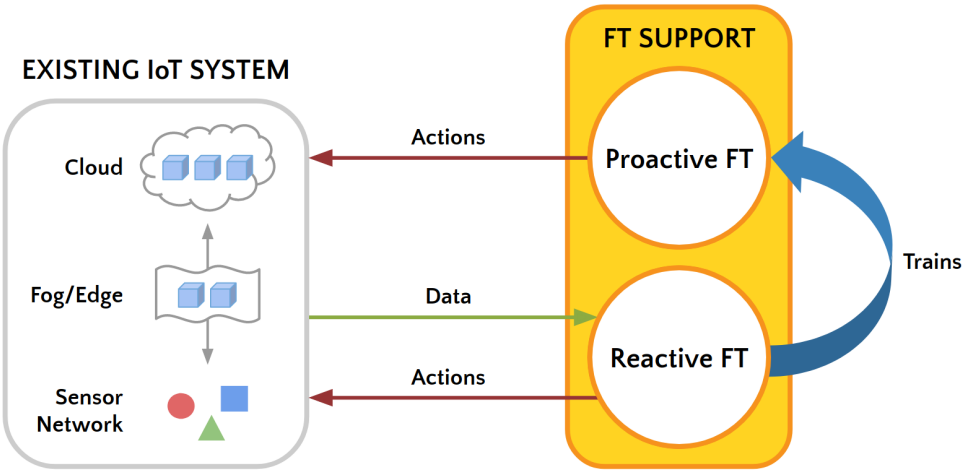

Figure 1.1: The FT-support subunits are 'plugged into' an existing IoT system, to learn how the system fails, and execute actions on the system to provide reactive- and proactive- FT support.

1.4 Significant Contribution to Thesis

The first contribution of this thesis is a literature review that provides a representative survey of state-of-the-art approaches to applying FT in IoT. It will focus on all aspects of the responsibilities of FT, namely: (1) the errors being detected and recovered from; (2) the techniques employed to provide this; (3) the overarching failures against which their FT-support solutions are protecting; and (4) how the systems are evaluated to demonstrate their efficacy. This contribution will inform the development of a *fault categorisation* framework that will provide a more structured understanding of how IoT systems can fail. Namely, system failure scenarios will be categorised by the system vulnerabilities, faults, and failures that enable the scenarios to occur [161].

The second contribution is to devise a novel *service-oriented predictive FT framework* that combines the real-time analytical power of CEP to react to errors in real time if they cannot be predicted, and the long-term predictive power of ML to anticipate and preempt errors that can be predicted [160, 162]. The FT framework consists of an interconnected set of subunits (Figure 1.1), whereby reactive- and proactive-FT support are designed to communicate information about the IoT system's state and errors via message passing. Specifically, they will be designed to be hosted as

two independent *microservices* with individual responsibilities, meaning that they can be deployed, scaled, and tested independently [192]. Adopting a microservices architecture makes the solution lightweight and easy to update in scenarios where future functionality requirements cannot be fully anticipated in advance, and multiple replicas of each microservice can exist at any given time for additional service redundancy and scalability [71].

The framework will learn about the ways in which the system can fail using the experience gained from the CEP system. ML models will identify correlations between contextual information and errors to anticipate the probability of a future error occurring imminently. The framework provides the following specific contributions:

- **Language-agnostic error definition**. Most CEP systems use *nondeterministic finite automata* (NFA) as the mechanism with which to define complex events [75]. NFA-based CEP systems can use NFAs to *define* errors, and composite events produced by NFAs can represent *detected* system errors. NFAs can be designed to be modular, reusable, and language-agnostic so they can be implemented in any NFA-based CEP system [161, 162].
- **Distributed CEP for resilient FT support at the network edge**. For CEP-based real-time FT to be effective, it needs to be pushed as close to the fallible sensor network as possible, and be fault tolerant itself. A bespoke CEP system, BoboCEP, will be designed to be distributed across the network edge on k instances of the software. The current state of partially completed NFAs can be maintained across CEP instances via *active replication*, so that $(k-1)$ instances can fail without complete FT-support service loss.

The third contribution is a rigorous evaluation of the FT framework against a *vertical farming system* (VFS) use case. It has been predicted that IoT device installations in the agriculture sector would increase from 30 million in 2015 to 75 million by 2020 [69]. VFS testbeds of varying scale and complexity will be designed to resemble those that exist currently in industry. This will help to assess whether the contributions of this thesis have the capability to scale to large-scale IoT systems [162].

The key qualities of the contributions in this thesis are as follows.

- **Interoperability**. The solution will be *pluggable* so that it can be introduced easily into existing IoT systems, which enables it to be applicable to all IoT systems. This requires the solution to adhere to communication standards that facilitate *semantic interoperability*, i.e. to understand system interfaces and data automatically for seamless connectivity and for FT support to fully understand the current system state [108].

- **Reusability**. FT support will be *generic* so that it is applicable to any IoT system. A solid understanding of common problems in IoT systems enables the development of common solutions to these problems. FT support solutions can then be reused in various contexts, e.g. for newly introduced hardware and software that fails similarly to existing system entities. Errors are represented as simple, modular, language-agnostic NFAs and, by being expressed mathematically and not programmatically, they are not tied to specific platforms or programming languages.

- **Trust**. For the FT-support solution to be effective, it must also be fault tolerant, so that it can be trusted to provide correct service despite faults in the FT support framework itself. The proposed solution avoids being a *single point of failure* (SPoF) by being capable of being redundantly distributed across an entire IoT system's infrastructure.

- **Scalability**. Evaluating the proposed FT framework on VFSs of varying scale ensures that the framework can cope with more sensors, more data, and a larger infrastructure that exhibits distributed event processing and load balancing. Having the FT support framework deployed as independent microservices means that support can be distributed across the IoT infrastructure e.g. reactive FT at the network edge for low-latency error detection, proactive FT in the cloud for more computing power.

1.5 Thesis Structure

Chapter 2 of the thesis provides a brief background on FT in terms of its relation to the wider topic of dependability, namely: (1) the attributes of dependability; (2) key FT concepts such as failure, error, and fault; and (3) other means of achieving dependability in literature, such as fault prevention, removal, and forecasting. It also explores the key stages of FT implementation: detection, assessment, recovery, and assessment.

Chapter 3 provides an in-depth analysis of FT specifically in the IoT domain. It provides an extensive review of sixteen proposed state-of-the-art FT solutions in IoT, and classifies them based upon: (1) the failures that FT is attempting to protect against; (2) the vulnerabilities the proposed solutions have; (3) the errors detection and recovery techniques they implement, and; (4) how the systems are evaluated.

Chapter 4 introduces an FT architecture based on microservices that underpins the key desired attributes of the FT-support solution in this thesis, namely: (1) to expose as much of the system state as possible; and (2) to adopt a standard method of acquiring the interfaces of system services. These qualities enable FT support to be 'plugged into' any existing IoT system and immediately begin to provide service. Furthermore, this chapter introduces the *Vulnerabilities, Faults, and Failures* (VFF) framework to classify system defects into tuples, which is used throughout the rest of the thesis for defect classification.

Chapter 5 introduces CEP in more depth, and its relation to being the mechanism to provide reactive-FT support. *Complex Patterns of Failure* (CPoF) is proposed, which is the overarching concept of defining errors as NFAs and detecting them by using CEP as a 'reactive-FT engine'. A variety of error-checking NFAs are proposed to help detect errors via data pattern analysis in CEP, namely: limit, trend, performance, persistence, correlation, and inconsistency checking. The correlation check is crucial to providing proactive FT with training data for supervised learning models, by learning from error correlations that occur within IoT systems.

Chapter 6 provides insight into the VFS use case, used to evaluate the efficacy of the

FT framework. A bespoke CEP system, *BoboCEP*, is proposed to provide resilient CEP at the network edge. Then, a series of VFSs are proposed. Firstly, a small-scale VFS testbed that contains a small number of plants, a limited infrastructure, and a lack of hardware or software redundancy. It uses the *FlinkCEP* NFA-based CEP system, which provides no CEP resilience. Secondly, a medium-scale VFS testbed is proposed that is an upgrade of the small-scale testbed, which contains more plants, more hardware and software redundancy, and CEP is distributed over three devices on the network edge using BoboCEP. Finally, a dataset from a real-world, large-scale VFS is used to validate that the FT framework can handle failure scenarios in a real-world VFS.

Chapter 7 evaluates the FT framework using the testbeds and dataset from Chapter 6. The approach taken is to first evaluate CPoF in isolation, to show how the proposed NFAs from Chapter 5 can provide reactive-FT support in isolation. Then, reactive FT and proactive FT are used together to provide predictive support to: (1) explore how correlations between with system context (e.g. time) and correlations with other errors can help to predict and anticipate future imminent errors; (2) ensure correct error detection despite erroneous data before detection; and (3) ensure that error recovery has actually moved the system into an error-free state. Finally, there is a demonstration of how BoboCEP facilitates long-term event processing and load balancing, as well as a performance analysis of the implementation used in the evaluation of this work.

Chapter 8 concludes the thesis by reflecting on how the work compares to the original research objectives from Section 1.3. The limitations of the work are considered, as well as future work that can address the limitations identified. Finally, the chapter concludes by summarising the findings of this research.

Chapter 2

Fault Tolerance

Fault tolerance (FT) is a means of achieving dependability that considers how to deliver correct service and avoid service failures in the presence of system faults [17, 16]. Its purpose is to mitigate the effects of system design faults by employing techniques during the development of the system, to enable it to tolerate any faults remaining in the system after its development [163].

This chapter provides a background on FT by first considering it in relation to the wider notion of dependability, followed by an overview of the stages employed by FT mechanisms to provide support.

2.1 Dependability

Many of the key fundamental dependability concepts and definitions that are widely accepted in FT literature arise from numerous publications by Avižienis et al. [15, 17, 16], which are outlined in this section. They define the dependability of a computing system as the ability for it to deliver service that can justifiably be trusted and can avoid service failures that are more frequent and more severe than is acceptable. The *trust* of a system is the acceptable level of dependence i.e. the extent to which a system is able to avoid service failures. The key concepts of dependability are outlined in their *dependability tree* (Figure 2.1), which considers the attributes of, the threats to, and the means by which to attain dependability, discussed next.

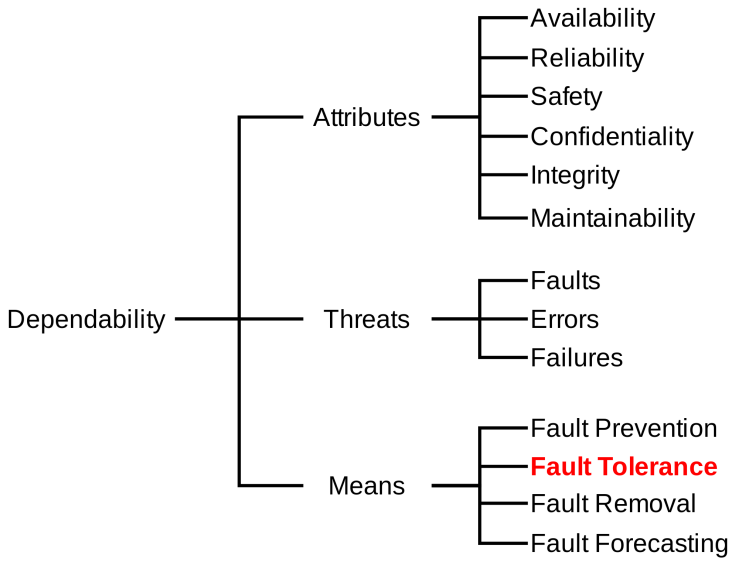

Figure 2.1: The dependability tree from [17], with FT highlighted in red.

2.1.1 Attributes

There are numerous attributes in literature that serve to measure the dependability of a system, and enable it to be understood from a variety of perspectives e.g. security, safety, and integrity. Some of the basic attributes are as follows [16]:

- **Availability**. Readiness for correct service. It is the degree to which a system or component is operational and accessible when required for use [40].
- **Reliability**. Continuity of correct service. Is it the ability of a system or component to perform its required functions under stated conditions for a specified period of time [40].
- **Safety**. Absence of catastrophic consequences on users and the environment.
- **Confidentiality**. Absence of unauthorised disclosure of information.
- **Integrity**. Absence of improper system state alterations.
- **Maintainability**. Ability to undergo repairs and modifications.

The applicability and prominence of these attributes can vary between system application. However, dependability is most often quantified in terms of *availability* and *reliability*, which are the two key attributes that can be improved using FT methods

specifically, i.e. a more fault-tolerant system is expected to be more available and reliable [150, 100]. They are both crucial for providing trustworthiness that a system is sufficiently *resilient* to failure, so that the system can withstand major disruptions within acceptable degradation parameters and recover in an acceptable and timely manner [49]. For example, they are important factors for cloud computing systems to ensure minimal disruption to offered services as the cloud scales, where key issues for scalability are avoiding, coping with, and recovering from failures [178, 175].

Sterbenz et al. [184] identified that the relative importance of availability and reliability (Figure 2.1) depends on the application service: for example, availability is of primary importance for transactional services such as HTTP-based web browsing, whereas reliability is of primary importance for session- and connection-oriented services, such as teleconferencing. Reliability is also a key requirement to protect the integrity of shared, distributed ledgers, such as *blockchain*, which support the execution of pieces of code called *smart contracts*, able to perform computations inside a blockchain. Putting multiple smart contracts into communication turns a blockchain into a proper distributed computing platform, which is appealing for applications that require code execution that is reliable, verifiable, and transactional [61]. Microservices are an important design pattern for providing more reliability and scalability because it breaks away from the monolithic *megaservice* antipattern that prevents the efficient upscaling of only the specific services that are in demand [5]. Microservices are better suited for applications that require high concurrency and high capacity, and test results have showed that they can help to reduce infrastructure costs in comparison with standard monolithic architectures, while guaranteeing the same performance and response times as the number of users increase [202, 170].

In this thesis, the desire to address availability and reliability, and therefore improve the dependability and trustworthiness of the system, is because the proposed FT framework needs to avoid the situation where its own ability to provide FT support fails, which would be catastrophic to the IoT system for which it provides support, and to the trust that the system designer would have in implementing the proposed framework. Key parameters to measure availability and reliability are [91, 114]:

- **Mean Time to Failure (MTTF)**. The average time for which a system will perform as specified, measured from the start of operation until the time that the first failure occurs. It can be used to compute the reliability of a system as $e^{-(1/MTTF)}$, where the failure rate is the inverse of MTTF. When hardware reliability assessment is coupled with software and firmware, the only way to ensure reliability is to set MTTF and repair objectives to qualify them through reliability testing [8].

- **Mean Time to Repair (MTTR)**. The average time required for a system to repair or replace a failed component in order to bring the system back into operation. It can be used, along with MTTF, as a common way to compute availability: $MTTF/(MTTF + MTTR)$. That is, the percentage of time that the system can perform its designed function.

- **Mean Time Between Failures (MTBF)**. The time from the start of operation until the a component is restored to operation after repair, i.e. $MTBF = MTTF + MTTR$. It is used to provide the amount of failure for a product with respect to time. MTBF is used for *repairable* components, whereas MTTF is used for those that cannot be repaired.

2.1.2 Threats

Failures

A *service failure*, or simply a *failure*, is an event that causes a deviation from the correct service, as defined by the functional specification of the system [16]. As discussed by Cristian [73], when designing a system to be fault tolerant, the system specification needs to consider not only the failure-free behaviours but also the likely failure behaviours, known as *failure semantics*. That is, if a specification prescribes that the failure behaviours likely to be observed in a system s are in some class F, then it can be said that: s has F failure semantics.

This is an important consideration because, when implementing FT mechanisms, it should be known which failure behaviours are likely to occur and which have a

probability of occurrence small enough such that they can be considered 'negligible'. Key failure semantics are as follows [73, 190]:

- **Omission**. When a server fails to respond to incoming messages. It can be classified into two sub-types: (1) **send**, when a server fails to send messages; and (2) **receive**, when a servers fails to receive messages.
- **Crash**. When a server halts and provides no further service. It is essentially an omission failure whereby *all* incoming messages receive no response.
- **Timing**. When a server's response is outside of a specified time interval. It can be classified into two sub-types: (1) **early**, when an action is performed too soon e.g. a timer runs too fast; and (2) **late** or **performance**, when a response occurs too late. If a response never arrives, it is an omission failure.
- **Response**. When a server's response is incorrect. It can be classified into two sub-types: (1) **value**, when the value of a response is incorrect; and (2) **state transition**, when the state transition of a server is incorrect.
- **Arbitrary**. When one of the aforementioned failure types occurs arbitrarily. In this thesis, arbitrary failures will be considered *unpredictable*, i.e. failures with no pattern of occurrence.

By treating failures as allowable system behaviours, they can be precisely defined and understood as functional requirements, enabling a systematic approach to applying error detection and recovery mechanisms that ensure failure mitigation.

Faults

A *fault* is the adjudged or hypothesised cause of an error and is the consequence of a failure in the system, where an *active* fault is one that produces an error, and a *dormant* fault is one that does not [163, 16]. A fault is a state in the system that can cause a reduction, or loss of, capability for the system to provide its correct functionality, which may cause a system to provide a lower standard of service to cope with untreatable faults (i.e. *graceful degradation*) [98, 122]. Faults can be placed into three major groupings [16]: (1) **development** faults, ones that occur

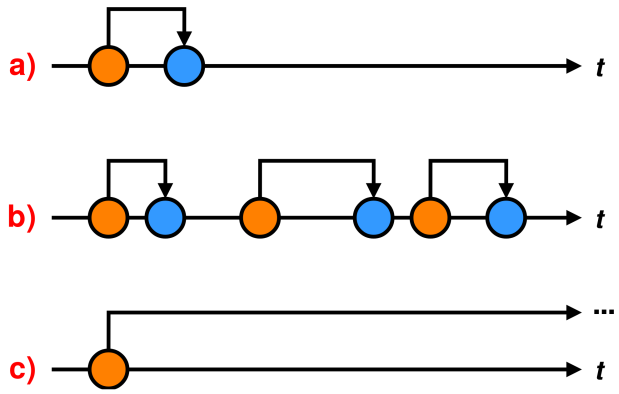

Figure 2.2: A visualisation of (a) transient, (b) intermittent, and (c) permanent fault persistence over time t. Orange circles represent an active fault, and blue circles represent a fault having transitioned to dormancy.

during system development; (2) **physical** faults, ones that affect hardware; and (3) **interaction** faults, ones introduced into the system from external sources.

Persistence The persistence of a fault is defined by three discrete categories (Figure 2.2), namely [150, 110]:

(a) **Transient**. Arbitrary faults, bounded in time, that cause erroneous behaviour for a short time before going away, with the potential to occur arbitrarily in the future. The errors usually do not have a clear root cause (e.g. network interference) and are resolved using simple recovery techniques (e.g. temporal redundancy) [91].

(b) **Intermittent**. A fault that does not go away entirely, but instead oscillates between being active and dormant (e.g. loose electrical connection).

(c) **Permanent**. A fault that is assumed to be continuous in time. For hardware, this reflects the permanent outgoing of a component (e.g. device failure). For software, this reflects the crash of some process.

Vulnerabilities Faults can be internal or external to a system. For example, a vulnerability in network security is a threat that can be exploited to cause faults in software e.g. malware [214]. An internal fault that enables an external fault to harm the system is a *vulnerability*, and is necessary for an external fault to cause an error and possibly subsequent failures [16].

Rauscher et al. [168] developed the *eight-ingredient* (8I) framework as a systematic way of conducting vulnerability analysis on both internal and external aspects of a given system. 8I identifies that the reliability and security of communications is vital for continuous system operation and is the key infrastructure upon which all other critical infrastructures depend. The framework identifies eight 'ingredients' that categorise different vulnerabilities which can manifest in a system, namely:

- **Human**. Intentional and unintentional behaviours, physical and mental limitations, education and training.
- **Policy**. Agreements, standards, policies, and regulations defining the behaviour between entities and governments.
- **Hardware**. Electronic and physical components that compose the network nodes, including circuits, fiber optics, and semiconductor chips.
- **Software**. Creating, maintaining, and protecting code, including physical storage, development and testing, version control, and control of code delivery.
- **Networks**. Topological configurations of nodes, synchronisation, redundancy, and physical and logical diversity.
- **Payload**. Information transported across the infrastructure, traffic patterns and statistics, and information interception and corruption.
- **Environment**. Temperature-controlled buildings, harsh conditions such as outside terminals and cell towers exposed to weather conditions, among others.
- **Power**. Internal power infrastructure, batteries, grounding, cabling, and backup generators.

The vulnerable areas outlined by each ingredient may vary in their applicability to IoT systems. However, 8I provides a complete understanding of all the potential vulnerabilities of IoT systems.

Errors

An *error* is a manifestation of an active fault in the system that appears during program execution or testing in which the logical state of an entity differs from its intended value [150, 110]. An error is *detected* if its presence is indicated by an error message or signal, and *latent* if undetected [16].

There is not a one-to-one relationship between a fault and an error: a fault may have multiple errors, and those errors may not be unique to a single fault in terms of how they manifest and are detected by a system [97]. For example, Malik et al. [137] explored the problem of data plane link failures in *software-defined networks* (SDNs). Vulnerabilities, such as human error, natural disasters, and system overload would cause identical network crash faults that would share a common data-loss error. However, regardless of which fault was the root cause, rerouting data flows through a working data link was the common solution to handle the faults. If understanding the root cause is crucial to provide the most appropriate recovery strategy, errors can be used to trace a fault by means of *fault diagnosis*, by analysing failure symptoms via sensor data analysis and signal processing [80, 205].

Error Propagation When an error is present in a system, it can be successively transformed into other errors that spread over multiple components (Figure 2.3), where the *speed* of propagation and *depth* (i.e. the number of affected processes) are primary measurements of the severity of propagation [16, 13]. The greater the depth, the more difficult it is to tolerate, which is why software-based detection techniques require careful placement of error detectors to cover the critical error-propagation paths and prevent excessive propagation [166]. A system's vulnerability to error propagation is often tested using *fault injection*, whereby faults are intentionally activated to assess the coverage and latency of error-detection mechanisms in place [125, 213]. However, up to 72% of injected faults cannot be considered representative of residual software faults because they are consistently detected during tests [147].

The propagation between two software components is an example of dependency resulting from functional interactions (e.g. data exchange), and the halting of software

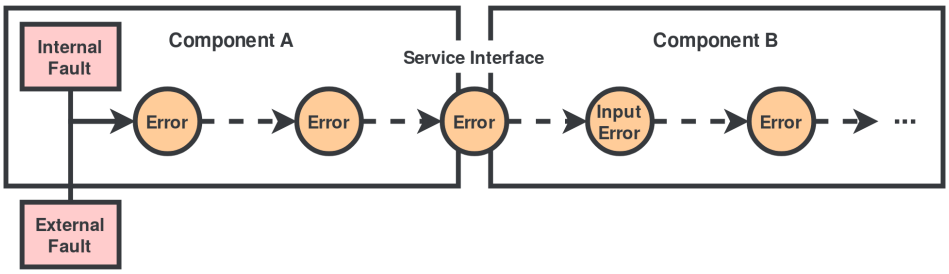

Figure 2.3: An example scenario of error propagation between two components, adapted from Figure 10 in [16]. Fault activation is shown with a solid arrow, and error propagation with a dashed arrow.

activities following a hardware crash is an example of dependency via a structural interaction [102]. When exception handling, if an exception cannot be handled and its error cannot be masked, it can be thrown (and therefore propagate) to the next outer exception, which further widens the scope of the error in the system and becomes increasingly more difficult to contain it [123].

2.1.3 Means

As defined at the start of the chapter, FT is a means of achieving dependability. However, there also exist three other means in FT literature, namely [16, 122]:

- **Fault Prevention**. To prevent the occurrence of faults at the development stage by using methodologies that help to minimise fault introduction.
- **Fault Removal**. To reduce the number and severity of faults at the development stage by means of testing.
- **Fault Forecasting**. To estimate the present number, the future incidence, and the likely consequences of faults via probabilistic estimation.

The purpose of FT is distinct from these other means for the following reasons. FT is implemented during the development stage, but only takes effect at runtime and, like fault prevention, aims to improve the trust in the system's ability to meet its specification. It relies upon redundancy and defensive programming techniques that are able to detect, assess, and recover from errors, as well as treat the underlying fault that caused errors in order to avoid faults and mask errors from the rest of the system

[163, 122]. Shifts towards FT are driven by the observation that fault avoidance does not deliver sufficient dependability against unexpected failure scenarios that system designers could not anticipate during initial system design [185].

2.2 Stages

The process of implementing FT is usually described by four discrete stages that, together, constitute the means by which FT support can enable system faults to be tolerated. The terminology used to describe the stages, and the stages themselves, often differ slightly across literature. Therefore, in this thesis, they are simplified and conflated into the following four stages: detection, assessment, recovery, and treatment, discussed next.

2.2.1 Detection

The detection stage is where FT attempts to detect erroneous system state. This occurs due to an active fault that cannot be detected directly i.e. an active fault can only be identified by the error(s) that it causes [122]. Errors are typically detected by *checking* the system state e.g. by routine means of an audit (checksum) check, or via special components that are designed to detect errors [91]. Lee et al. [122] outline a broad classification of checks that the majority of error-detection measures use, namely:

- **Replication**. Using alternative implementations of a system against which the activity of the system can be checked for consistency. That is, the output from each replica is compared to detect the presence of an error as manifest through an inconsistent output from some replica(s) (e.g. triple-modular redundancy).
- **Timing**. If the specification requires timing constraints of the provision of service, then the service can monitor whether the constraints have been met. If they are not, a 'timeout' may be triggered to indicate this.
- **Reversal**. This involves taking the output from a system, calculating what

the input should have been in order to produce that output, and comparing it against the actual input to see if there was an error.

- **Coding**. Within an object, redundant check data is maintained in some fixed relationship with non-redundant data, where corruption of either form of data can therefore be detected (e.g. via parity checks).
- **Reasonableness**. These checks test whether the state of various system objects are 'reasonable' based on the intended usage and purpose of the objects. For example, an angle θ in degrees is only acceptable if its value is $0 \leq \theta \leq 360$.
- **Structural**. There are two types of structural check: (1) *semantic integrity*, that check the consistency of information contained within a data structure; and (2) *structural integrity*, that check whether the structure itself is consistent.
- **Diagnostic**. These checks differ from the ones covered in that they check the behaviour of the components from which the system is constructed, rather than the behaviour of the system itself. They involve exercising a component with a set of inputs for which the correct outputs are known.

Although system resilience is improved by the extra resources introduced via redundant components, limits on resources demand that designers choose which threats to target more redundant resources, leading to more flexible designs that enable comparison of the risks and costs of different solutions [185]. The benefit of redundancy is that it enables the identification of discrepancies between redundant entities. These techniques come in the following three forms.

Design Diversity

This is the technique in which redundant components (i.e. *variants*) are independently designed and implemented but still adhere to the same system specification, in order to minimise the probability that a fault in one variant exists in the other variant(s) [105]. Design diversity facilitates the implementation of several error-detection checks i.e. replication and reasonableness checks.

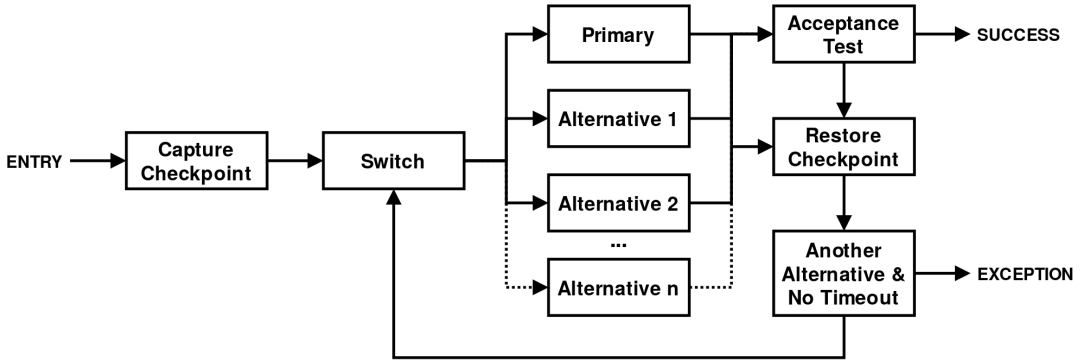

Figure 2.4: The Recovery Block approach.

Adjudication In addition to at least two variants, design-diverse FT requires adjudication. This is a process to decide whether a "correct" result has been produced by the variants that supposedly provides an error-free result after execution [159]. It can come in three types [163]: (1) a *voter*, which compares the results from two or more variants (e.g. majority, mean, and consensus vote); (2) an *acceptance test* (AT), which verifies acceptable system behaviour based on the success of an assertion (e.g. satisfaction of requirements, and reasonableness tests); or (3) some hybrid of the two. Examples of design diversity are as follows [163, 165, 146].

Recovery Block (RcB) This approach attempts to fulfil an AT with multiple software variants that are executed sequentially (Figure 2.4). When a variant fails to pass the AT, it rolls back to the last checkpoint, checks to see whether any timeouts have occurred, and attempts to fulfil the AT on the next variant. If all variants have been tried without successfully fulfilling the AT, an exception is raised.

As a *backward error recovery* technique, an error detection causes the system to revert to a previously saved (error-free) state via checkpointing, and then re-computes the correct system state using a sequence of variants [196]. It applies *passive replication* that favours a primary variant as the first to be executed, and progressively attempts backup variants in a given sequence until all variants have been attempted. An example RcB algorithm is shown in Algorithm 2.1.

Khan et al. [106] considered a distributed variant of RcB for fault-tolerant wormhole routing where a section of a parallel system that spans between the source and des-

Algorithm 2.1 An example Recovery Block algorithm.

```
alternatives[]                                   ▷ Array of alternatives.
CAPTURE_CHECKPOINT()

for a in alternatives do
    output ← EXECUTE(a)                          ▷ Execute current alternative a.
    if ACCEPTANCE_TEST(output) = true then
        return true                              ▷ Passed acceptance test.
    else
        RESTORE_CHECKPOINT()
        if HAS_TIMED_OUT() = true then
            return EXCEPTION                     ▷ Timeout occurred.
        end if
    end if
end for

return false                                     ▷ All acceptance tests failed, or error occurred.
```

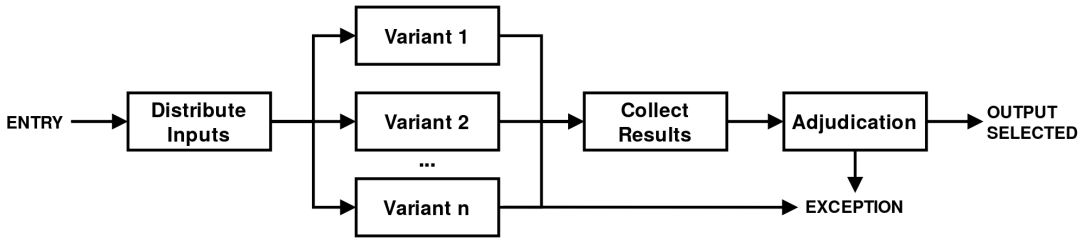

Figure 2.5: The N-Version Programming approach.

tination nodes was dynamically partitioned into overlapping distributed RcB groups. Simulation results indicated that it was able to tolerate both node and link failures.

N-Version Programming (NVP) This approach uses at least two independently designed variants that concurrently produce results, and uses an adjudicator to select a final result (usually via majority vote), or an exception is raised if no result can be determined (Figure 2.5). It uses *active replication*, a technique which gives all replicas the same role without centralised control found in passive replication i.e. the primary-backup technique used in RcB [89]. An example NVP algorithm is shown in Algorithm 2.2.

A recent NVP case study looked favourably on NVP, in that it enabled cross-checking between variants to find bugs, however, the lack of collaboration between the inde-

Algorithm 2.2 An example N-Version Programming algorithm.

```
variants[]                                          ▷ Array of variants.
input                                               ▷ Input data for variants.
outputs[]                                           ▷ Outputs from variants.

for v in variants do
    THREAD_EXECUTE(v, outputs)                      ▷ Concurrently execute variants.
end for

while length(outputs) != length(variants) do        ▷ Wait for all variant outputs.
    if HAS_TIMED_OUT() = true then
        STOP_THREADS()
        return EXCEPTION                            ▷ Timeout occurred.
    end if
    WAIT(1)                                         ▷ Wait for 1 second.
end while

if MAJORITY_VOTE(outputs) = true then               ▷ Adjudicate outputs via vote.
    return true                                     ▷ Passed vote.
else
    return false
end if
```

pendent teams that were developing independent variants was seen to stifle development [126]. *Triple-modular redundancy* (TMR) is similar to NVP but for hardware FT, where three functionally-identical modules can detect a fault in a single module and mask it instantly via a majority vote of the output from each module [14, 122].

The $t/(n-1)$-*Variant Programming* (TN1VP) approach is a similar approach to NVP [212]. It can identify the correct result from a subset of the results of n variants, providing that the number of faulty modules does not exceed t i.e. it can tolerate at least t software faults. Both TN1VP and NVP can tolerate some related faults between variants. However, in general, TN1VP has higher reliability, whereas NVP is better from a safety perspective.

N Self-Checking Programming (NSCP) This approach resembles NVP, however, self-checking can check its own behaviour during execution. NSCP can adapt to partial failures whereby an agreement between one pair but a disagreement between another will cause the agreeing pair to be chosen as the final result; two pairs with

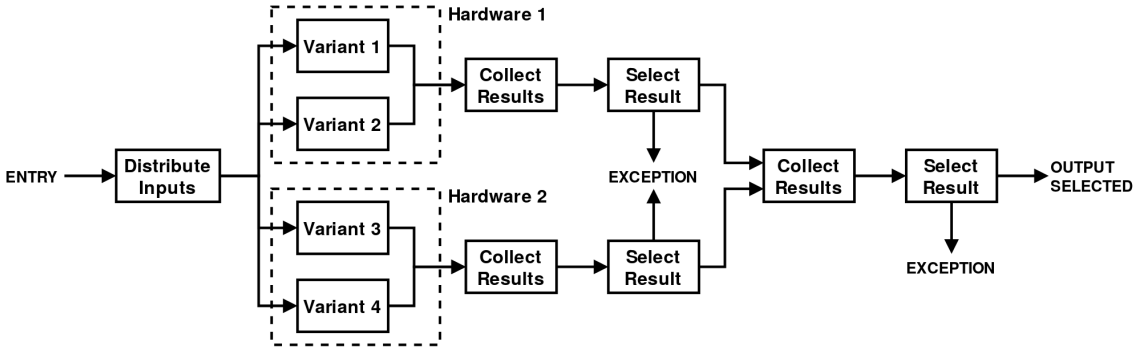

Figure 2.6: The N Self-Checking Programming approach.

Algorithm 2.3 An example N Self-Checking Programming algorithm.

$result\_1 \leftarrow$ RUN($variant\_1$) on HARDWARE 1
$result\_2 \leftarrow$ RUN($variant\_2$) on HARDWARE 1
$result\_3 \leftarrow$ RUN($variant\_3$) on HARDWARE 2
$result\_4 \leftarrow$ RUN($variant\_4$) on HARDWARE 2

if COMPARE($result\_1$, $result\_2$) $= true$ **then**
    $pair\_1 \leftarrow true$ ▷ First pair match.
else
    $pair\_1 \leftarrow false$
end if

if COMPARE($result\_3$, $result\_4$) $= true$ **then**
    $pair\_2 \leftarrow true$ ▷ Second pair match.
else
    $pair\_2 \leftarrow false$
end if

if $pair\_1 = true$ and $pair\_2 = false$ **then**
    $output \leftarrow result\_1$ or 2 ▷ Select first pair.
else if $pair\_1 = false$ and $pair\_2 = true$ **then**
    $output \leftarrow result\_3$ or 4 ▷ Select second pair.
else if $pair\_1 = false$ and $pair\_2 = false$ **then**
    return EXCEPTION ▷ Neither pair match.
else if $pair\_1 = true$ and $pair\_2 = true$ **then**
    if COMPARE($result\_1$ OR 2, $result\_3$ OR 4) $= true$ **then**
        $output \leftarrow result\_1$ or 2 or 3 or 4 ▷ All results match.
    else
        return EXCEPTION ▷ Each pair match but with different results.
    end if
end if

return *output*

different agreed results would raise an exception. It works as follows, with an example algorithm shown in Algorithm 2.3.

NSCP executes two pairs of variants on two separate hardware components. The results from each pair are compared:

- If any pair's results do not match: a flag is set to indicate pair failure.
- If a single pair failure occurred, then the non-failing pair's result is returned as the NSCP result.
- If both failed, then an exception is raised.

If each pair matched, the results from each pair are then compared:

- If both pairs match, the result (shared by all variants) is returned as the NSCP result.
- If both pairs do not match, then an exception is raised.

A simulation analysis by Gokhale et al. [87] showed that, compared to NVP and a distributed variant of the RcB approach, the expected number of failures was highest for NSCP. This was assumed to be because NSCP ran more software variants in parallel than NVP and RcB, which therefore made it more error prone than the other two approaches. NSCP ran four software variants executing in parallel, three for NVP, and two executed sequentially, rather than in parallel, for RcB.

Data Diversity

The technique wherein if one expression of data x fails execution of function $P(x)$, then it is "re-expressed" as $y = R(x)$ for re-execution as $P(y)$, which is repeated until P accepts some diverse form of x, or an exception is raised [163, 127]. Diversity is achieved through the variants of input data that are generated via data re-expression. Data diversity facilitates the implementation of several error-detection checks i.e. reversal, coding, and structural checks.

Nguyen-Tuong et al. [151] considered data diversity in N-variant systems to protect against data corruption attacks. Data was transformed in the variants so that

identical concrete data values had different interpretations. That is, in order to corrupt data without detection, an attacker would need to alter the corresponding data in each variant in a different way while sending the same inputs to all variants.

Rizwan et al. [172] proposed data re-expression algorithms that could be used to tolerate some hardware faults. For example, if memory faults occurred, data diversity could be used to customise memory allocation for storing. Gashi et al. [83] exploited redundancy in the SQL language by re-expressing one or more SQL statements into different but logically equivalent statements in order to produce redundant executions. This helped to reduce the risk of a failure being repeated when the re-expressed statement was executed on the same or another replica of the same database management system.

Retry Block (RtB) This approach uses an AT and backward error recovery to achieve FT, which means that it closely resembles RcB. The main difference is that RtB executes the primary alternative multiple times with re-expressed inputs, and only switches to a backup alternative if a timeout occurs before it is able to achieve an output that fulfils the AT. The backup alternative executes with the original input. If it fails to fulfil the AT, an exception is raised. An example RtB algorithm is shown in Algorithm 2.4.

Temporal Diversity

The technique in which event execution is performed at different times, such that if an input fails at time t, it is re-executed at a later time $t+n$ [163]. Temporal diversity facilitates the implementation of several detection checks i.e. timing and diagnostic checks. An example algorithm of how temporal diversity could be implemented is shown in Algorithm 2.5.

Lewis et al. [124] considered how load balancing served as an example of temporal diversity by modifying the algorithm being used by the load balancer every 20 seconds. They showed that, by adding this diversity, they were able to decrease failure rates in handling client requests. Lathia et al. [120] used temporal diversity as a mechanism for retaining user interest in recommendation systems by examining how a number

Algorithm 2.4 An example Retry Block algorithm.

```
input                                                    ▷ Input data.
timeout ← false

result ← PRIMARY(input)                                  ▷ Execute primary alternative.

while ACCEPTANCE_TEST(result) = false do
    if HAS_TIMED_OUT() = true then
        timeout ← true
        BREAK                                            ▷ Timeout occurred.
    end if
    result ← PRIMARY(REEXPRESS(input))                   ▷ Primary with re-expressed inputs.
end while

if timeout = true then
    result ← BACKUP(input)                               ▷ Execute backup alternative.
    if ACCEPTANCE_TEST(result) = false then
        return EXCEPTION                                 ▷ All alternatives failed.
    end if
end if

return result
```

Algorithm 2.5 An example temporal diversity algorithm.

```
input                                                    ▷ Input data.
while EXECUTE(input) = false do                          ▷ Execution failed.
    timeout ← HAS_TIMED_OUT()
    if timeout then
        return false                                     ▷ Timeout occurred.
    else
        WAIT(5)                                          ▷ Wait for 5 seconds.
    end if
end while
return true                                              ▷ Execution succeeded.
```

of characteristics of user rating patterns affected diversity, e.g. profile size and time between rating.

Çiço et al. [54] presented the exploitation of temporal diversity for *virtual machine* (VM) instantiation in the cloud via URL calls between subsystems. *Representational State Transfer* (REST) requests were performed to initialise VMs. However, if transient errors occurred, the problem could be mitigated with temporal diversity by repeating the URL request with the same input data several times until a correct result was obtained.

2.2.2 Assessment

The assessment stage encompasses several notions in FT literature. Firstly, it is the period after error detection where the underlying fault that caused the error(s) is hypothesised, based on contextual information (e.g. the erroneous component, and the time of occurrence). This is termed the error assessment (or error diagnosis [163]) and helps to identify the best error-recovery procedure to execute given the hypothesised fault.

Another part of the assessment stage is to understand the extent of the damage caused by the fault activation, because the initial error might have propagated (Section 2.1.2) before being detected. This is termed damage confinement [122], or error containment/isolation [163, 91], and it serves to ensure the entire scope of erroneous behaviour is quarantined and the error is isolated to a unit of mitigation before applying holistic error recovery to the unit.

For example, Hu et al. [95] proposed a structural health-monitoring system for bridges that used data from a connected *wireless sensor network* (WSN) that contained accelerometers, strain gauges, and temperature sensors for detecting bridge vibrations and strains. An important error assessment, therefore, is to determine whether dangerous levels of vibration and strain are a matter of bridge integrity loss or sensor malfunction. Moreover, in a star topology network, it is challenging to assess whether devices have failed, or congestion is the cause. Plus, if devices have failed, it is difficult to determine which have failed, because a gateway failure will

cause all other devices to appear to have crashed, a problem which Álvarez et al. [10] proposed to minimise with their replicated gateway solution.

2.2.3 Recovery

With the error(s) detected and assessed, the recovery stage serves to transform the current erroneous system state into an error-free state wherein normal system operation can continue [122]. A similar notion, *error mitigation*, is also an error-processing technique. Whereas error recovery substitutes an erroneous state for an error-free state, error mitigation attempts to remove, or mitigate, an erroneous state without transitioning to a new state e.g. data correction [91].

The primary strategy for error recovery is via *redundancy*, which is defined as the presence of auxiliary components in a system to perform the same or similar functions as other elements for the purpose of preventing or recovering from failures [40]. The use of sufficient redundancy can enable a system to recover without explicitly declaring error detection and therefore enables the system to *mask* the error, or provide architectural dynamic reconfiguration, for fault handling [17, 123].

In hardware, power electronic converters are always based on hardware redundancy plus associated control strategies, and it is shown that some FT methods, such as redundant switching states and neutral shift, are easily implemented and cost effective, but cannot maintain the-full rated power after faults [216]. In aerospace design, Bennett et al. [25] considered redundancy in electric drive for aircrafts that were able to maintain a constant torque despite aircraft power supply and control signal failures, in order to ensure adherence to stringent aerospace safety, reliability, and FT standards.

In software, critical infrastructures, such as energy, transportation, and nuclear industries, require highly reliable software that can be achieved via the cloud, where data redundancy, availability, and survivability are attainable when essential system components are isolated or lost [7]. The reliability of *web services* is achieved via redundant services working in parallel, which are identical in functionality so that, if one fails, it can be replaced without complete service loss [44, 88]. *Microservices*, a

service-oriented architecture (SOA) subset, are an enabler for scalable, agile and reliable software systems, and therefore the coupling, integration, scalability, monitoring and development of microservices helps to achieve reliability [94]. Having redundant microservices means that, if one fails, other instances will still be available to take the load of the failed microservice. This is also useful for load balancing whereby many microservice instances are available simultaneously and can scale with demand i.e. *horizontal scaling* [171].

In most error-detection scenarios, a fault occurs, which causes errors to be present in a system, and the error-detection mechanism uses its internal logic to deduce the presence of the errors and trigger a recovery strategy. For safety-critical applications, it is desirable to identify the conditions that lead to faults to prevent predicable errors from jeopardising safe operation [199, 180]. A mixture of both approaches covers predictable and unpredictable faults, discussed next.

Reactive FT

This approach is where the system initiates error recovery *after* an error has been detected, and requires fast decision making with a low-latency connection to the hardware or software at fault. Key approaches are as follows [173, 19, 156, 3]:

- **Checkpointing/Restart**. When a task fails, it restarts from a recently saved (i.e. checkpointed) state, rather than from the beginning, which is an efficient technique for long-running applications.
- **Job Migration**. If a job cannot be completed on one machine, the task is migrated onto another machine to complete it instead.
- **Replication**. Several instances of a task are replicated and run across numerous different resources (i.e. active replication), such that if one fails, the other resources can compensate.
- **Exception Handling**. When the normal flow of execution fails, exceptional conditions can be defined to recover from it.

- **Self-Healing**. When multiple instances of an application are running on various machines, it automatically handles failure of application instances.

For example, in large-scale WSNs where thousands of sensors are scattered over a wide area, there is an increase in the probability of network component and data transmission failures. Retransmitting data, or sending data replicas across multiple paths, provide effective reactive approaches based on redundancy to mitigate the effects of the failures [53].

Proactive FT

This approach is where the system initiates error recovery *before* an error has been detected by reacting before a fault is activated and, therefore, mitigating the fault's imminent system error(s) that might have potentially led to service failure(s) [70]. Proactive FT and fault forecasting (Section 2.1.3) are similar concepts. However, the purpose fault forecasting is to estimate the present number, the future incidence, and the consequences of faults [119], whereas proactive FT is a *runtime* approach to preempting imminent errors.

Preemptive migration is similar to the reactive-FT strategy of job migration. However, with this approach, failures are prevented by *preemptively* migrating parts of a system away from hardware or software that is expected to fail imminently [70, 144]. For example, a common use is to examine the behaviour and utilisation of *virtual machines* (VMs) such that, when a VM is predicted to fail imminently, services are migrated to new VMs to avoid downtime [158, 198].

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 [56]. *Software rejuvenation* stops ageing software, cleans its internal state, and resumes it, where the downtime is masked using FT strategies [35]. In modern cloud solutions, software rejuvenation is often tied to VM preemptive migration, whereby fallible VMs are replaced by new VM processes brought online to replace them before they fail [134, 198]. The ageing effects on software are the result of error accumulation, in terms of leaked resources

or corrupted state [56]. By examining the failure distribution of a system to monitor software ageing indicators (e.g. memory leaks), the software rejuvenation trigger interval can be optimised by predicting the *time to (resource) exhaustion* (TTE), in order to maximise system availability or minimise operational costs [218].

Load balancing is also a form of proactive FT. That is, whenever the load of CPU and memory exceeds a certain limit (e.g. 75% utilisation of CPU), the load from the CPU is transferred to another CPU that has not exceeded its limit, which prevents the original CPU from being overwhelmed with traffic and potentially crashing [3].

2.2.4 Treatment

The treatment stage encompasses the techniques that can be used to target a specific fault that has caused error(s) which required detection, assessment, and recovery, by helping to ensure that the effects of the fault do not immediate recur [122]. It is the final stage of FT that removes the fault from the system, typically by means of software updates or patches [91].

Bondavalli et al. [29] proposed a mechanism to discriminate intermittent and permanent faults against low rate, low persistence transient faults with a device called α-count that attempts to: (1) keep components in the system that have just experienced transient faults; and (2) quickly remove components affected by permanent or intermittent faults.

Meling et al. [142] proposed the *Distributed Autonomous Replication Management* (DARM) framework that improved system dependability via a self-managed fault-treatment mechanism that adapted to network dynamics and changing requirements. DARM provided self-healing and self-configuration, where objects were relocated or removed to adapt to failures (i.e. job migration), or controlled changes such as scheduled maintenance (e.g. software and operating system upgrades). Error recovery decisions were distributed to each individual group deployed in the system, which eliminated the need for a centralised manager and enabled groups to perform fault treatment on themselves.

2.3 Summary

This chapter started by placing FT within the wider notion of dependability, and also explored similar means of achieving dependability that existed in literature: fault prevention, removal, and forecasting. Key FT concepts were also defined, most crucially: failures, faults, and errors, and how these three concepts relate to each other. The ways in which dependability could be quantified were also explored, and *availability* and *reliability* were identified as being two key attributes that can be improved using FT methods.

With regard to how FT is implemented, four stages were identified: (1) *detection*, where FT attempts to detect erroneous system state(s); (2) *assessment*, where the underlying fault that caused the error(s) is hypothesised; (3) *recovery*, where the current erroneous system state is transformed into an error-free state; and (4) *treatment*, which explores techniques that can be used to target the fault which caused the error(s) which were recovered from.

Chapter 3

Fault Tolerance in IoT

This chapter provides a comprehensive review of proposed FT solutions in literature that specifically target the IoT domain. The sixteen systems chosen for review are intended to be representative and were selected to take into account scale, heterogeneity, portability, and application. They range from small-scale, decentralised systems to large-scale smart city applications, to provide a broad insight into how FT is applied across diverse IoT systems.

The search string "*iot fault tolerance*" was used when searching for these solutions, in order to keep them relevant to IoT. The review focusses on the motivations of the work, the targeted domain, proposed mechanisms to provide the FT necessary to achieve dependability, and system validation (if any). They are in ascending order from the first author's surname and year of publication.

The proposed solutions are summarised (Section 3.2) by considering the failures addressed, the vulnerabilities identified as having allowed faults to be present, the detection and recovery strategies used, and an overview of how the proposed solutions were evaluated.

3.1 Review

3.1.1 Self-Learning Group-Based Fault Detection (SGFD)

Liu et al. [128] proposed a framework for online sensor fault detection that was targeted at monitoring IoT systems in an industrial setting (e.g. an oil field, as

explored in their evaluation). Their system focussed on three types of data-value fault that were assumed to occur due to sensor failure, namely: outliers, stuck-at faults, and spikes.

To detect these faults, they proposed a mechanism called *statistics sliding windows* (SSW) that created a series of windows, wherein one contained recent sensor data that regressed into historical windows. For example, for sensor s, the current value v_c and the previous m values would form the current window $W_c = \{v_{c-m}, v_{c-m+1}, ..., v_c\}$ in the set of k windows of equal length $H_v = \{W_1, W_2, ..., W_k\}$. Values discarded by W_c would populate a buffer window W_r that, upon reaching size m, would join set H_v as W_1, and W_k would be dropped. A stuck-at fault would be detected if, for current window c, $\sigma^2 = 0$. A spike would be detected if σ^2 were large enough to pass a given threshold.

They considered how the system could detect data trends by comparing the current trend vector with historical ones to detect both rational and irrational events. A *rational* trend meant that the sensor value transformed smoothly from one level to another and was caused by a 'rational reason'. An *irrational* trend meant that the value changed when something was wrong with a device. They proposed the *status transform window* (STW) to define a time interval and calculate the cosine angle between the current trend and existing vectors, respectively, whereby a new trend was identified if the angle were large enough, given some threshold.

The self-learning framework architecture comprised three modules: (1) *Application DB*, the interface for a high-layer application which retrieved sensor fault predictions and user feedback; (2) *Detection Thread*, a background service that contained the main detecting process; and (3) *Self-Learning Thread*, which used user feedback to revise its trend vectors. This was coupled with the *group-based fault detection* algorithm that applied the aforementioned concepts to detect data-value faults.

Their evaluation was a simulation that used real data from an oil field. Experimental results showed that the system could detect 95% of data-value faults in the simulation data, which contained 751.68 million samples from 5800 sensors. Their solution did

not explore error-recovery mechanisms, and was only concerned with detecting data-value faults in an industrial IoT system.

3.1.2 Services Choreography (SerCho)

Cherrier et al. [46] explored the idea of services choreography to distribute application logic across a network and avoid orchestrations (i.e. centralised control) to achieve a more efficient use of energy in a constrained network. They built upon their original work of the *D-LITe* framework [45] that used *finite-state transducers* (FSTs) to express system logic as a variety of states through which each underlying 'object' (i.e. physical device) could transition. For the choreography to work, the synchronisation between objects was required. As an object changed its state, other objects might have needed to change their states too.

To perform all necessary state transitions, an initial state was changed and the rest that need changing were done so in a cascade: the object with the initial state change observed its '*to-be-checked*' list and notified other objects about the state to which they should transition also. This was performed until the state change of all relevant objects was completed. They proposed a mechanism to address state desynchronisation by simply putting objects back into a compliant and coherent state if they were discovered to be in incorrect states.

Their experimental study consisted of 4 objects: 1 generator that sent 12 events (1 per second) and waited 10 seconds before repeating (which was an arbitrary waiting time chosen by the authors); and 3 counters organised as a cascade (i.e. in series), which received and counted events before sending them to the next counter to do the same. Each counter should have received all 12 events. Random message erasure was simulated in the experiment at rates of 0%, 5%, 10%, and 15% erasure. The first experiment applied the checking system in the 10-second waiting period to resynchronise any counters that were not in their correct states, whereas the second experiment did not.

As counters received data, the resynchronisation required increased with each device. At 10% error rate, the correction mechanism failed because check messages were

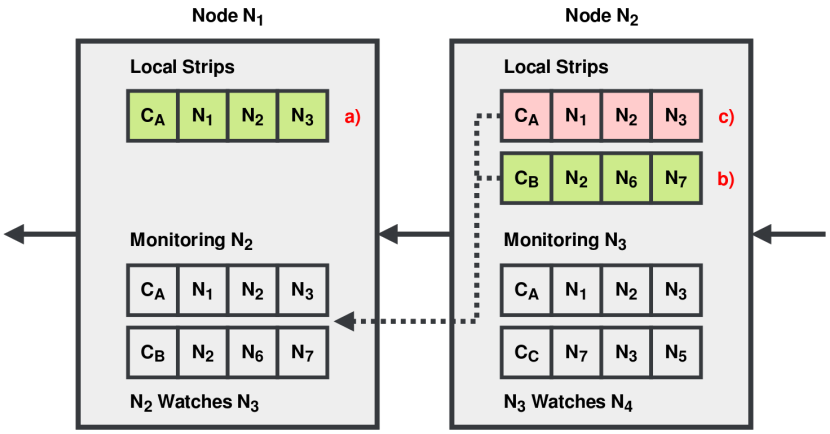

Figure 3.1: An example scenario of the decentralised FT mechanism proposed by Su et al. [186].

lost, which demonstrated how the proposed solution was not scalable if it could not work reliably in a system with so few devices at a 10% error rate. Furthermore, the check frequency showed that making checks more often when the error rate increased resulted in a higher valid state count. At 5% error rate, there was an average of 50% valid states if checks were made after every event, meaning that very frequent checks were needed to keep track of failed data transmission.

3.1.3 IoT/M2M Middleware (M2M-Mid)

Su et al. [186] demonstrated a decentralised FT mechanism that distributed application 'components' (i.e. services) across underlying devices. Each component was delegated to a specific device with the redundant devices waiting in standby to fulfil the component (although redundant devices could be fulfilling different components in the meantime). A 'redundancy level' determined how many devices were delegated to provide component redundancy. The purpose behind a decentralised design was that it provided fail-over on device failure in a way that alleviated a central body from orchestrating recovery strategies, therefore removing any SPoF.

As shown in Figure 3.1, devices were stored in a *strip* (i.e. a queue), where the first in the queue fulfilled the component's data needs and the queued devices took over if the primary device failed. To keep strips up to date, a heartbeat protocol was used in a 'daisy chain' (i.e. ring topology) around the devices. When a device had not

received a heartbeat message in sufficient time, it triggered the process of updating strips across the network to remove failed devices from the system and to delegate alternative devices that can provide any necessary components.

The performance of the proposed mechanism was measured by three metrics. Firstly, message overhead during device failure. Of the ten devices used in the demonstration (mapped unequally to four components), it took ~550 bytes of data to recover from a failed node. The second metric of average recovery time for nodes was ~2500ms. The third metric combined detection time with recovery time, and results showed that the total time was no greater than a heartbeat period.

3.1.4 Smart Control Algorithm (SCA)

Choubey et al. [51] presented a smart home architecture consisting of a variety of sensors in a home that were connected to a *Local Master* (LM), i.e. a fog device. The LM connected to a remote *Cloud Server Farm* that accumulated data from all connected homes. The purpose of their proposed FT mechanism was to attempt to avoid data blackouts when devices failed by predicting data that could not be generated by physical sensors.

When the system initially started, all sensors were active while the system collected a sufficient amount of data, after which the system would identify relationships between data using *correlation coefficient*. Any correlation coefficient beyond $\pm$ 0.7 indicated that one data type could predict another due to a sufficiently a high correlation between them. Data types were placed into two sets: one set for use as input to an *artificial neural network* (ANN) situated on the LM, and another set that could be predicted by the ANN.

After correlations had been found, the system transitioned into a *smart control algorithm* (SCA) state, where the LM put devices to sleep that could be predicted in order to conserve energy (Figure 3.2). Specifically, temperature was put to sleep because it could be predicted by the other four sensors. When a predicted temperature value deviated from its last real-world value beyond some threshold, a temperature sensor reactivated for its real-world data before being put to sleep again. The SCA

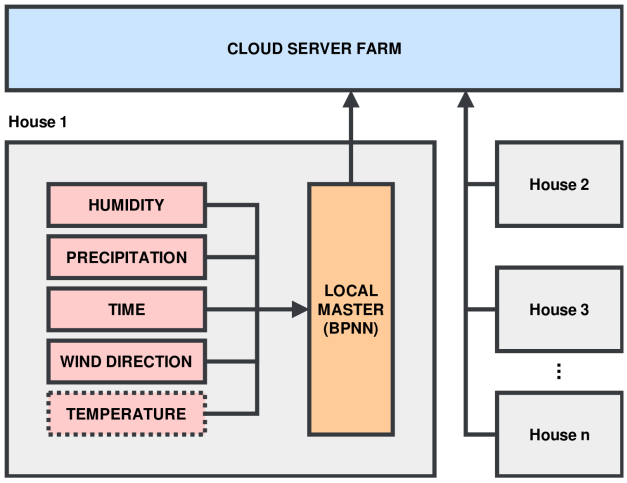

Figure 3.2: An example scenario of the smart home architecture proposed by Choubey et al. [51].

could also temporarily predict sensor data when performing maintenance on failed sensors as a means of providing FT.

Tests demonstrated training an ANN to predict temperature data using humidity, precipitation, time, and wind direction, after correlations were identified between temperature and the other variables. Results showed that predicted temperatures deviated from actual temperatures by no more than $\pm$ 2°C.

3.1.5 6LoWPAN Health Monitoring (6LoW-HM)

Gia et al. [86] explored FT in a healthcare scenario by providing backup routing from node to sink, as well as employing a mechanism to maintain connectivity in the case of a connection failure between node and sink. The authors outlined a desire for healthcare systems to adhere to strict QoS requirements for data transmission rates, as defined by the ISO/IEEE 11073 standards group. For example, heart-rate readings were expected to be transmitted in < 3 seconds, and maintaining these critical requirements was the motivation for their work.

The system architecture comprised a star topology of nodes, a gateway that consisted of sink nodes and a *system-on-chip* (SoC) board, and a back-end server. The gateway was responsible for providing FT mechanisms as well as collecting node data to transmit to the back-end. The gateway monitored data flow and detected inactivity

from sensor nodes via timing checks. On detected sensor inactivity, the protocol was to 'ping' the affected device, but if this yielded no response, the ping was repeated through a different sink node to rule out any individual sink node being the cause of ping failure.

Four testing scenarios were conducted involving a standalone gateway and seven sensors, all of which were positioned at varying distances from each other. The extent to which the FT mechanism was tested included disabling a sink node and discovering that the system was still able to receive the necessary data in all four scenarios. This demonstrated that sink node redundancy could provide the appropriate error-masking capabilities, however the evaluation provided little in the way of quantitative measures and instead demonstrated a simple failure scenario.

3.1.6 FT Programming Framework (FTPF)

Hu et al. [96] devised a framework for application developers to apply FT via exception handling, facilitated by languages such as Java and Python due to the exception-handling functionality featuring in those languages. When faced with an error, it was detected and annunciated by raising an exception. A snapshot of sensor data was stored for backward error recovery via checkpointing and a relevant error handler was notified on error detection.

Their evaluation was conducted using Raspberry Pis, each with three means with which devices could communicate between themselves: Wi-Fi, Bluetooth, and ZigBee. Unlike with many other solutions, their FT mechanism was situated on devices themselves with no centralised control. Their experiment involved eight nodes that transferred sensor data between themselves, during which three devices' communication components failed, triggering errors in other devices when they could not connect to the faulty devices. Non-faulty devices would attempt to reconnect to the faulty devices via an alternative medium (e.g. from Wi-Fi to Bluetooth) and resynchronise between themselves so that devices knew which new mediums to use to send future data.

Results showed that the failure detection time was between 35-45ms. However,

because a device could only know about a failure when it tried to connect to the faulty device, the detection time could be much higher (~200ms); a faster and more consistent detection time could have been attained if a heartbeat protocol were in place, as with the solution from Section 3.1.3. The recovery time was more consistent at ~3300ms. The message overhead to resolve issues was proportional to the number of nodes with errors.

The work used alternative communication mediums as redundant hardware for fail-over, demonstrating that they could isolate faults down to the individual components to prevent replacing an entire device when only a specific component was faulty. However, with the experiment being conducted using Raspberry Pis and a high-level programming language, there was little insight into whether the solution was practical for systems with more constrained hardware.

3.1.7 Cloud Application Placement Problem (CAPP)

Spinnewyn et al. [183] considered the problem of efficiently placing applications on a physical network prone to both node and link failure, where applications were composed of services on a substrate network. They investigated: (1) the degree to which availability-awareness improved the efficiency of application placement on an unreliable substrate network; and (2) the degree to which placement could be improved using redundant services and virtual links. QoS was a key requirement of their work as they wanted to maximise the number of placed applications while satisfying a required level of availability R for each application.

Their approach proposed active-active redundancy to enable node and link replication via application 'duplication' (i.e. replication). Duplication provided not only redundancy, but also enabled applications to share physical components, such as CPU, memory and networking resources between themselves. The decision to accept an application request was determined by whether there was at least one duplicate available for it, expressed as:

$$\forall a \in \mathbf{A} : O^a \leq \sum_{d \in \mathbf{D}} G^{d,a} \tag{3.1}$$

Where **A** was the set of request applications, O^a was the acceptance of an application a (1 iff accepted), D was the set of duplicates, and $G^{d,a}$ was the placement for duplicate d of application a (1 iff placed). With regard to availability awareness, for a duplicate to be available, each of the individual components it used must have been available, and a component was used if it hosted any of the duplicate's services or virtual links.

They evaluated the performance of their algorithm by simulating a cloud environment with unreliable nodes. They considered three metrics, namely: (1) the *CPU load factor*, which was the ratio of total CPU demand of all application requests to the total amount of CPU resources; (2) the *placement ratio*, which was the ratio of applications that met the availability requirements to the total requests; and (3) the *computation time*, in seconds. They compared the algorithm with a maximum of 1, 2, or 3 duplicates per application, and experiments were conducted using varying availability levels of 0%, 90%, and 99%.

Results showed that more duplicates led to higher availability, and thus a higher placement ratio. When the availability level was 0%, all algorithms performed equally as well, but at 99%, the test with 3 duplicates outperformed the rest. The computation time was shown to be highly dependent of the required availability level, where the time increased as the level of redundancy increased.

3.1.8 UbiFlow

Wu et al. [209] proposed the UbiFlow system that used SDNs in IoT for ubiquitous flow control and mobility management. It adopted distributed controllers to divide urban-scale SDNs into different geographic partitions, where all jobs related to mobility management, handover optimisation, access point selection, and flow scheduling were executed by the coordination of distributed controllers.

The authors identified that the main issues related to the application of software-defined IoT were: (1) the operation and coordination of a distributed control plane when component failure or traffic congestion occurred, by providing data replication, flow scheduling, and load balancing; (2) link and node capabilities in IoT systems

were highly heterogeneous and application requirements were correspondingly different; and (3) the performance metrics of interest in IoT multi-networks went beyond bandwidth consumption due to more heterogeneous and time-sensitive traffic that was concerned with delay, jitter, packet loss, and throughput.

In the UbiFlow architecture, multiple controllers were deployed to divide the network into several partitions, which represented different geographical areas. IoT devices in a partition associated with different access points (e.g. Wi-Fi, Cellular) were connected to local switches to request various types of data flow (e.g. text, audio) from the corresponding data server. Switches between partitions were partially interconnected so that network information recorded in different controllers could be exchanged via switches to achieve network consistency and robust maintenance. Each controller maintained a 'finger' (i.e. routing) table for routing between controllers.

The authors considered FT from many perspectives. To handle controller failure, they adopted data replication by copying data from a local controller to others so that the system could find a new successor that could still provide the control service. If there were a finger-table failure, the system would choose an alternative path through which to reroute. Rerouting also occurred if access points failed.

UbiFlow was evaluated based on both its flow scheduling and mobility management performance via simulation and real testbed experiments, which were in the context of an urban environment in London consisting of several parks, universities, and museums. When compared against other common SDN scheduling algorithms, UbiFlow was able to provide significantly higher throughput due to its load balancing scheme mitigating packet loss during high load, and adding more controllers increased the throughput almost linearly. Their real-world testbed produced similar results to their simulations.

3.1.9 Modalities via Virtual Services (ModVS)

Zhou et al. [220] took a different approach to hardware redundancy by using sensors of different *modalities* to provide backups for failed sensors. The approach identified compatibilities between different sensors via *regression analysis* in order to ascertain

whether it was possible to combine data from multiple different sensors into 'virtual services'. For example, a microphone in a quiet room could be used for intruder detection when the primary *passive infrared* (PIR) sensor failed. The study compared a linear and non-linear model to assess which performed better with different modalities. Ten devices were deployed with light, PIR, Kinect [217], sound, and ultra-sound sensors.

Firstly, modalities for light sensors involved a culmination of other redundant light sensors because no other sensor types could fulfil light data. Results showed that individual light sensors that were distant from the failed sensor performed badly by themselves but achieved greater results when combined into a virtual service. Secondly, when comparing presence-detection modalities, a non-linear model achieved better results due to the underlying relationship between sensor data being non-linear.

One drawback to the work's regression approach was 'causal relationships'. For example, if human presence made light actuators switch on, then the system might have perceived light sensors as a presence modality because the room was always brighter when presence was detected; of course, this did not mean that light sensors were actually detecting presence. In addition, simultaneous sensor failures or failures of unrecognised sensors would result in system resynchronisation, by performing a complete device remapping and redeployment to find new backups for required services. This meant that the regression scheme needed to be overwhelmingly supported in order to minimise the use of this expensive fail-safe procedure.

3.1.10 Z_{ij}-Routing

Ali et al. [9] proposed the decentralised, stochastic Z_{ij}-routing algorithm that aimed to increase the packet delivery ratio and decrease the delivery delay in IoT systems with unreliable communication links. Their work focussed on how data was routed via multiple hops across nodes in the network, and the chosen routing path was determined by assigning higher transition probabilities to neighbouring IoT devices that had higher link reliability and distance ratio. Specifically, an absorbing Markov

chain was used to calculate the *expected delivery ratio* (EDR) and *expected delivery delay* (EDD) from source to destination.

They defined link reliability as the ratio between the number of packets received by some node N_j and the number of packets transmitted by some node N_i. They expressed the EDR, assuming random path selection, as:

$$\sum_{R_k \in R} \prod_{(i,j) \in R_k} p_{ij} L_{ij} \tag{3.2}$$

Where R was the set of paths from source to destination, R_k was the kth path, (i, j) were the links along the path R_k, and p_{ij} was the probability of selecting the given link. The EDD was given by:

$$(\mathbf{I} - \mathbf{C})^{-1} \times \mathbf{1} \tag{3.3}$$

Where $\mathbf{C}$ was a matrix of transition probabilities between states, $\mathbf{I}$ was an identity matrix, and $\mathbf{1}$ was a column vector with all values equal to 1.

Their proposed routing algorithm defined the parameter Z_{ij} for a pair of IoT devices N_i and N_j that considered: (1) the packets flowing out of the sender's neighbouring area; (2) the latency of packets in a sender's neighbouring area; and (3) the distance of IoT device N_j to the destination device. After computing Z_{ij} for each pair of neighbouring devices, the transition probabilities for their links were calculated as:

$$p_{ij} = \frac{Z_{ij}}{\sum_{k \in m_i} Z_{ik}} \tag{3.4}$$

Where m_i was a group of IoT devices with which N_i could directly communicate.

To evaluate the performance of their algorithm, they generated simulations of 100 random directed acyclic networks with random link reliability values, and then calculated the average EDR and EDD for them. The algorithm was compared against three reference decentralised routing algorithms, namely: (1) random walking, where each IoT device had equal transition probability i.e. $p_{ij} = 1/m_i$; (2) L_{ij}-routing,

which only considered link reliability and not all of the aforementioned characteristics of Z_{ij}-routing; and (3) reliability expander, which measured the aptitude of packets flowing out of the sender's neighbouring area and the latency of packets in the sender's neighbouring area.

Results showed that a higher number of IoT devices in the network decreased the EDR due to an increase in the number of hops during routing. However, Z_{ij}-routing's EDR was significantly better due to transition probability decision making. Similarly, the EDD became larger as IoT devices increased because of the larger number of hops. However, Z_{ij}-routing still had the lowest delay due to its being able to determine the best route for packet delivery.

3.1.11 Modalities for Graceful Degradation (ModGD)

Chilipirea et al. [48] also explored the notion of modalities but as a means of replacing sensors with alternatives that were not directly compatible. The authors cited that many identical sensors in close proximity was unrealistic, and so they identified sensor 'capabilities' even if they were not as suited to the role as the original device, thus providing a degraded service rather than no service.

With a sensor's capability as the defining metric, sensors could be ordered and compared for particular roles based on quality and accuracy, enabling an ordered relationship to be formed from the most capable sensor to the least capable. For example, identifying a specific person was a greater capability than detecting the existence of a person, which was greater than simple motion detection. The failure to start one sensor component meant that another could be activated in its place.

The ordered relationship of capabilities demonstrated how modalities offered a form of graceful degradation, whereby, upon device failure, a 'less capable' sensor could take its place. This meant that the degradation of a system could be predefined, and switching to less capable sensors could be adopted as an error-recovery strategy.

The main drawback to this work was how the modalities were manually sourced and ordered, unlike in [220] where regression strategies identified the best candidates.

As well, they provided no experimental study demonstrating the effectiveness of the solution, and instead provided hypothetical failure scenarios.

3.1.12 NewIoTGateway-Select (NIoTGS)

Karthikeya et al. [103] explored the infrastructure of a large-scale 'IoT Smart City' scenario and proposed the *NewIoTGateway-Select* algorithm to determine the minimum number of gateways necessary to reduce deployment costs and provide redundancy. The algorithm considered gateway and link failures by ensuring there were at least k alternative routes via *k-coverage* and *k-connectivity* algorithms, respectively. The purpose of this work was to maintain connectivity to the Internet and to have gateways that covered all *coordinating devices* (CD), e.g. cluster heads.

NewIoTGateway-Select first computed a set of locations for placing gateways, and then used this set to assign a value denoting the number of CDs containing the location in their transmission range. Candidate locations with the largest number of CDs in range were chosen until all CDs were eventually covered by some gateway. To protect against gateway failures, a *2-coverage* algorithm could be used to ensure every CD was covered by at least 2 gateways. To protect against link failures, the *2-connectivity* algorithm could also be used to ensure that there were at least 2 independent paths between gateways.

Simulation results showed how the 2-coverage algorithm roughly doubled the number of deployed gateways, but that the total number of gateways was still drastically lower than the amount that would have been used without NewIoTGateway-Select. The 2-connectivity algorithm, which explored link failures, was not discussed.

3.1.13 Smart Cities Architecture (SmartCA)

Abreu et al. [2] also considered smart cities by proposing a novel IoT architecture designed specifically to improve the resilience of smart city infrastructures. Their contribution was an architecture that took design and implementation into consideration and the protocols and technologies that support key resilience features in the different layers of the architecture. The authors acknowledged how smart cities

required good end-to-end network quality and asserted that the cloud and 'cloudlet' (i.e. the fog) were enabling technologies to realise a resilient IoT smart city solution. The cloud processed resource-demanding applications and the cloudlet delivered real-time services with minimal, one-hop latency.

Their proposed architecture had three layers:

1. **IoT Infrastructure**. The lowest layer of the architecture that dealt with physical devices deployed in the smart city. Data collected by 'IoT islands' (i.e. clusters of devices) was sent to the IoT Services layer (below) to be processed and analysed.

2. **IoT Middleware**. This layer encompassed common functionalities and abstraction mechanisms for easier interaction between layers. Some important functionalities included:

   - *Heterogeneity Manager*. This provided language-agnostic communication for the upper layers.

   - *Communication Manager*. This determined how data was exchanged between services, applications, and smart objects. It supported having different *virtual networks* (VNs) over the same physical infrastructure, which enabled the overlaying of network topologies.

   - *Virtualized Device Manager*. This provided mechanisms to identify, discover, and locate services within the IoT Infrastructure layer.

   - *Resilience Manager*. This attempted to provide robustness and supervision to the IoT Infrastructure layer and addressed any faults that occurred.

3. **IoT Services**. This layer managed applications and services that supported the smart city and provided an analysis of the data collected by sensors using Big Data techniques.

A key focus of the work was the use of cloudlet hardware that was close to, or one hop away, from the underlying devices in the IoT infrastructure layer for the deployment

and virtualisation of crucial components provided by the cloud. The cloudlet could provide some cloud-like services if the cloud were unavailable, which would provide a level of redundancy for cloud services. While they acknowledged that the cloud was only realistically capable of processing resource-demanding applications, the cloudlet could deliver real-time services 'in the last mile', even for worst-case scenarios.

No real-world implementation was provided for their work, but they covered two scenarios of faults occurring and how the system would recover from them, both of which were conducted in the middleware located in the cloud.

Scenario 1 This scenario explored the failure of underlying devices and how they affected the VNs that used the devices. An error would occur if the system monitor checked the status of the VNs and a 'failure' status was returned. The recovery strategy would be to reconfigure and resynchronise the VNs to include a new *virtual sensor* that would replace the affected sensor.

Scenario 2 If heavy traffic congestion occurred such that the QoS rating of the system dropped below a specified threshold, an alternative data path would be provided through which to reroute traffic.

3.1.14 Privacy-Preserving Data Aggregation (PPDA)

Lu et al. [129] proposed the *lightweight privacy-preserving data aggregation scheme* to support data aggregation for heterogeneous IoT devices. Their system model consisted of: (1) an IoT network of *heterogeneous devices* $D_1, D_2, ..., D_N$ equipped with sensors that periodically transmitted a sensing output x_i; (2) a *fog device* deployed at the network edge that aggregated device data $x_1, x_2, ..., x_N$ and forwarded this to (3) a *control center* in the cloud that performed data analytics according to application requirements; and (4) a *trusted authority* that bootstrapped the system and managed the IoT devices, fog, and control center.

If there were a malicious attack on the system via the injection of false data, the fog would execute source authentication to filter out false data locally, without sending

them on to the control center. False data could also cause external error propagation by inciting erroneous decision making at the control center.

As IoT devices would periodically send their data to the control center via the fog, the system divided this time period into w time slots where, at each time slot, the IoT device would report its data. The trusted authority would build one-way hash chains and add signatures to all hash chains' heads in order to ensure that all chains were valid for authentication. When the fog received the current hash value h_{is} in time slot T_s, it checked the validity against the hash value $h_{i(s-1)}$ that was previously verified in the last time slot, to ensure the data had not already been accepted. If this were the case, h_{is} would be rejected. A message authentication code was additionally computed, where an invalid code also led to data rejection.

Their evaluation did not include a demonstration of their proposed FT support. Instead, it simply covered the communication overhead and computational costs incurred by applying their solution.

3.1.15 CEFIoT

Javed et al. [99] proposed the CEFIoT architecture to provide FT using state-of-the-art technologies that were deployed on both the cloud and the edge, namely: (1) the lightweight containerisation software Docker[1]; (2) the data publish-subscribe platform Kafka[2]; and (3) the container-orchestration system Kubernetes[3].

The architecture had three layers: (1) *Application Isolation*, that wrapped individual processes into separate independent blocks and configured them to operate as a single isolated application, sending data from source to sink; (2) *Data Transport*, that buffered and replicated stream data across the system for logical data flow; and (3) *Multi-Cluster Management*, that monitored the previous two layers by assigning data processing to physical nodes and load balancing.

The authors attempted to address the issue of having separate implementations for the cloud and edge by using Docker across the infrastructure, which enabled com-

[1]https://www.docker.com
[2]https://kafka.apache.org
[3]https://kubernetes.io

putations to be pushed to the edge to minimise bottlenecks at the cloud and reduce latency. In layer (2), they attempted to solve the data-FT issue using Kafka because it replicated data across many nodes and buffered data when the system failed to connect to the cloud, which meant that the system provided graceful degradation without data loss. In layer (3), they used Kubernetes for on-the-fly automatic reconfiguration of the processing pipeline, which handled both hardware- and network-connectivity failures.

They evaluated CEFIoT using a surveillance camera system, where they clustered five Raspberry Pi nodes together that collected images from cameras attached to each node, and replicated images across nodes. The image were then sent to the cloud for further processing and storage. The system was able to tolerate a two-node failure in the node cluster because Kubernetes shifted processes onto different nodes, which kept the system available and the data processing pipeline alive. By using Kafka, if a node were damaged, the data would be retrieved from another node that had a replica of the image.

3.1.16 Personal Healthcare Devices (PerHD)

Woo et al. [208] proposed a reliable IoT system for *personal healthcare devices* (PerHDs) that adhered to: (1) *ISO/IEEE 11073*, which is an international standard for PerHD communication; and (2) *oneM2M*, which is an international standard for IoT systems. The system used PerHDs to gather data and transmit it to a *Middle Node-Common Service Entry* (i.e. a gateway) using the ISO/IEEE 11073 protocol. It was at the gateway where messages were handled, converted into a oneM2M-compliant format, and then passed to a resource manager that executed the necessary operations according to the converted oneM2M message. The gateway provided efficient communication between PerHDs and a central PerHD *management server* from which PerHDs could access data.

The authors outlined safety concerns involved with losing PerHD data, and so they proposed an additional FT algorithm to increase system reliability. They attempted to accomplish this at both the gateway and the management server by linking

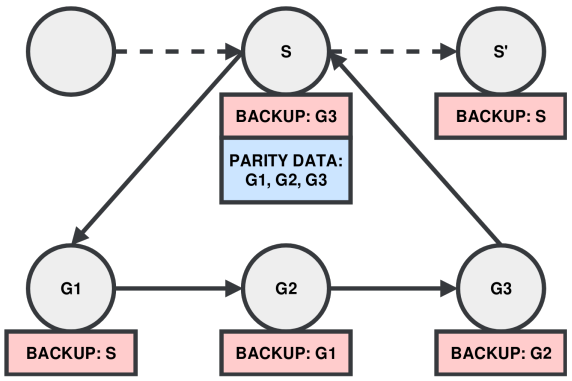

Figure 3.3: An example scenario of the gateways in the IoT system proposed by Woo et al. [208].

gateways in a daisy chain, so that a gateway would have a backup copy of its data stored on the gateway immediately ahead of it; PerHDs were not included in the FT mechanism because of computational limitations.

Multiple levels of gateways were used, where an upper-layer gateway (or the management server if there were not another layer of gateways) stored parity data of all gateways in the layer below it, as well as a backup copy of the data for the last gateway at the end of the daisy chain (Figure 3.3).

The authors explored several failure cases to demonstrate how the FT algorithm worked. They demonstrated multiple hypothetical scenarios using three gateways G1, G2, G3 and an upper gateway (or server) S. G1's backup existed on G2, G2's was on G3, and G3's was on S, as shown in Figure 3.3.

Case 1 If there were *one* fault at G1, then, once back online, it would call on S to find out which gateway from which to request its backup data (i.e. G2), so that it could recover and resynchronise with the system.

Case 2 If there were *two* non-consecutive faults at G1 and G3, then S would provide itself as backup for G3 and S would indicate to G1 that G2 provided backup for G1.

Case 3-1 If there were *two* non-consecutive faults at G1 and G2, then G3 would recover G2 and S would recover G1 by using parity data and G2's and G3's data.

It would then send the recovered data to G1 directly. S would request that G1 send its data to G2 so that all gateways would have backup copies once again.

Case 3-2 If there were *two* non-consecutive faults at G1 and S, then S would call on the next server in *its own* daisy chain, S', to recover its backup and parity data. G1 would be recovered using backup data from G2. S would request that G3 send a copy of its data to S, so that it would have the necessary backup.

Case 3-3 The case where G3 and S fail was omitted from the work because it followed a similar procedure to that in *Case 3-2*.

Their experimental study was a test between 1 server and 3 gateways, configured like in Figure 3.3. The failure demonstrated in their experiment was the same as that in *Case 1* and was successful at employing its backup recovery data to the failed device, with a transfer of 4182 bytes of backup data to accomplish the recovery.

3.2 Summary

3.2.1 Failures

Table 3.1 shows the failures (Section 2.1.2) that were identified in the reviewed solutions. IoT systems are data driven, so most concern was placed on ensuring the timely arrival of sensor data, and the primary failures under consideration were therefore: (1) whether hardware had crashed and would cause a data blackout [51]; (2) whether devices were failing to send or receive data [186, 96]; and (3) whether data in transit would arrive late and consequently be less valuable [208].

Few solutions paid attention to the actual data itself. That is, whether data values were 'correct' and, if not, whether an erroneous state transition might be caused due to erroneous data. For example, Liu et al. [128] considered three forms of erroneous data: outliers, stuck-at faults, and spikes. It is through data that IoT systems derive their value, and so response failures (Section 2.1.2) should be taken further into consideration to ensure data integrity in IoT systems.

Table 3.1: The failures addressed in the reviewed FT solutions (checkmarked). Little focus had been on data-centric i.e. response value failures.

Name	**Section**	**Omission**		**Crash**	**Timing**	**Response**	
		Send	*Receive*			*Value*	*State Transition*
SGFD	3.1.1					✓	✓
SerCho	3.1.2	✓			✓		
M2M-Mid	3.1.3	✓	✓		✓		
SCA	3.1.4	✓	✓	✓	✓		
6LoW-HM	3.1.5		✓	✓	✓		
FTPF	3.1.6	✓	✓		✓		✓
CAPP	3.1.7	✓		✓			
UbiFlow	3.1.8	✓		✓	✓	✓	
ModVS	3.1.9	✓	✓		✓		
Z_{ij}-Routing	3.1.10	✓			✓		
ModGD	3.1.11			✓	✓	✓	
NIoTGS	3.1.12	✓		✓			
SmartCA	3.1.13	✓			✓		
PPDA	3.1.14	✓				✓	✓
CEFIoT	3.1.15	✓		✓			
PerHD	3.1.16			✓			

3.2.2 Vulnerabilities

Table 3.2 shows the vulnerabilities (Section 2.1.2) that were considered in the reviewed solutions. The solutions sought to ensure the continued service of hardware in order to generate the necessary sensor data, and also to ensure that hardware was sufficiently capable of transferring data from its source to its destination.

For this reason, hardware and network vulnerabilities were the priority for most of the FT solutions, with particular consideration given to how power failure and human error could contribute to failure. For example, [86] used gateway monitoring to detect sensor node inactivity via data-flow analysis, [183] duplicated entire applications to enable node and link replication, and [220] identified compatibilities between various sensors so that they could be used to back up each other to some (degraded) extent.

Fault-prone software is a well-established dependability concern, such as with the issue of residual software faults [148], and is an equally legitimate threat to IoT systems. However, it was not widely acknowledged in any of the reviewed systems.

Table 3.2: The vulnerabilities identified in the reviewed FT solutions (checkmarked). Context-aware, data-centric FT was not prevalent in the reviewed solutions, and so environment vulnerabilities were scarcely addressed.

Name	**Section**	**Hardware**	**Software**	**Networks**	**Payload**	**Environment**	**Power**	**Human**	**Policy**
SGFD	3.1.1	✓			✓				
SerCho	3.1.2	✓	✓	✓				✓	
M2M-Mid	3.1.3	✓					✓		
SCA	3.1.4						✓		
6LoW-HM	3.1.5	✓		✓					
FTPF	3.1.6	✓	✓	✓					
CAPP	3.1.7	✓		✓					
UbiFlow	3.1.8	✓		✓					
ModVS	3.1.9	✓		✓			✓		
Z_{ij}-Routing	3.1.10			✓					
ModGD	3.1.11	✓							
NIoTGS	3.1.12	✓		✓					
SmartCA	3.1.13	✓		✓				✓	
PPDA	3.1.14				✓			✓	
CEFIoT	3.1.15	✓		✓		✓		✓	
PerHD	3.1.16	✓		✓					

Furthermore, as with Table 3.1, vulnerabilities in data and the environments in which IoT systems were situated were not widely considered, despite correct sensor data collection being the fundamental requirements for effective IoT decision making. However, an environment vulnerability was considered in the evaluation of [99] when they collected images from five cameras in their security system, and failure to ensure this jeopardised the security of the room being monitored.

3.2.3 Error Detection

Table 3.3 shows the error-detection checks (Section 2.2.1) that were considered in the reviewed solutions. Timing checks were the most prevalent check, used to minimise data loss and data transmission failures. Data loss occurred when a device did not receive data within an acceptable time, whereas data transmission failure occurred when a device failed to send data altogether, which is why timing checks were so prevalent. These two failures contributed to the primary error of untimely data, as well as the fundamental requirement shared by every IoT system: to generate data and ensure its reliable transmission to data sinks. Both of these failures could be caused by the failure of a single device, but because hardware was so distributed in the reviewed systems, ambiguity existed with regard to what specifically is the fault and, thus, how to accurately identify and treat it. This left an open issue of where data errors were best handled e.g. at the component(s) sending the data [186, 96], or the component(s) receiving it [86], or both.

QoS requirements were acknowledged in [86, 2] as the driving force for having more stringent checks on the timeliness of data transmission. Traffic congestion was identified as a direct threat to QoS because congestion persistently affected the data transmission rate. The systems concerned with QoS were mainly smart city and healthcare applications that adopted three-tier IoT infrastructures consisting of devices, gateways, and a back-end system. This indicated that large-scale, standards-based systems were where QoS was of greater concern.

Table 3.3: The error-detection checks used in the reviewed FT solutions (checkmarked).

Name	Section	Replication	Timing	Reversal	Coding	Reasonableness	Structural	Diagnostic	Unspecified
SGFD	3.1.1					✓		✓	
SerCho	3.1.2		✓				✓	✓	
M2M-Mid	3.1.3		✓						
SCA	3.1.4					✓			
6LoW-HM	3.1.5		✓			✓			
FTPF	3.1.6		✓			✓			
CAPP	3.1.7								✓
UbiFlow	3.1.8		✓					✓	
ModVS	3.1.9	✓	✓					✓	
Z_{ij}-Routing	3.1.10					✓			
ModGD	3.1.11		✓			✓			
NIoTGS	3.1.12								✓
SmartCA	3.1.13		✓					✓	
PPDA	3.1.14		✓		✓				
CEFIoT	3.1.15		✓						
PerHD	3.1.16				✓				

Table 3.4: The error-recovery mechanisms used in the reviewed FT solutions (checkmarked).

Name	Section	Replication	Job Migration	Modalities	Checkpointing	Ping	Retry	Exceptions	Other
SGFD	3.1.1								✓
SerCho	3.1.2						✓		
M2M-Mid	3.1.3	✓	✓						
SCA	3.1.4		✓						✓
6LoW-HM	3.1.5	✓				✓			✓
FTPF	3.1.6		✓		✓			✓	
CAPP	3.1.7	✓							
UbiFlow	3.1.8	✓	✓						
ModVS	3.1.9		✓	✓					
Z_{ij}-Routing	3.1.10		✓						
ModGD	3.1.11		✓	✓					
NIoTGS	3.1.12	✓							
SmartCA	3.1.13	✓	✓						✓
PPDA	3.1.14								✓
CEFIoT	3.1.15	✓	✓						✓
PerHD	3.1.16	✓							

3.2.4 Error Recovery

The most common approaches to recovering from system errors involved some kind of redundancy mechanism (Table 3.4). For example, approaches included: (1) hardware or software replication, so that one component failure was masked by other active component(s) [183]; and (2) service migration to new components and isolating the failed components, which also served to mask error(s) [186].

Some mechanisms were used as a means of recovering a previous state prior to an erroneous state transition. In [96], checkpointing was adopted to restore device variables, whereas [208] downloaded an entire backup of a device's data once it had been restored. In [46], when objects were supposed to transition to new states but did not, additional message passing occurred to check objects and put them into their required states to 'resynchronise' the system.

In [51], tactical data exploitation methods over hardware were used to provide redundancy. They found the relationships between different types of data through an analysis of historical sensor data. Predictions were made as to what missing data should have been, providing a kind of 'data redundancy' in that it compensated for a lack of hardware capable of generating actual real-world data. The solutions exploring modalities in [220, 48] offered graceful degradation by exploiting relationships between unrelated sensors, so that redundancy was achieved without requiring identical hardware. This avoided cumbersome hardware redundancy by exploiting data as much as possible in order to minimise the number of devices being deployed.

3.2.5 Evaluation

Table 3.5 shows the technologies and evaluation approaches that were considered in the proposed solutions. IoT's distributed nature resulted in them implementing FT support at varying distances from data sources. IoT devices seldom carried the responsibility of FT, except in [96] where the recovery approach updated a device's internal state without consensus between devices.

In decentralised solutions where the edge, fog and cloud were not used, FT mech-

Table 3.5: The technologies and evaluations used in the reviewed FT solutions (check-marked).

Name	**Section**	**Technology**		**Evaluation**		
		Edge/Fog	*Back-End*	*Simulation*	*Testbed*	*Scenario*
SGFD	3.1.1		✓	✓		
SerCho	3.1.2			✓	✓	
M2M-Mid	3.1.3			✓	✓	
SCA	3.1.4	✓	✓	✓		
6LoW-HM	3.1.5	✓	✓		✓	
FTPF	3.1.6			✓		
CAPP	3.1.7		✓	✓		
UbiFlow	3.1.8			✓	✓	
ModVS	3.1.9		✓		✓	
Z_{ij}-Routing	3.1.10			✓		
ModGD	3.1.11		✓			✓
NIoTGS	3.1.12			✓		
SmartCA	3.1.13	✓	✓			✓
PPDA	3.1.14	✓	✓	✓		
CEFIoT	3.1.15	✓	✓		✓	
PerHD	3.1.16	✓	✓	✓	✓	

anisms were often distributed. For example, [46] synchronised state between devices to avoid a centralised orchestration, and [186] used a ring topology with a heartbeat protocol to check neighbouring sensors for failures. The portability of decentralised solutions was limited because they relied upon stringent protocols and topologies in order to function correctly, and were not demonstrated to cope well beyond small-scale solutions.

Detecting faults is challenging with a limited system view, so FT was often pushed towards the back-end. The 'back-end' is left as a broad term because some solutions, like [220], did not specify the details of where remote storage and computing would take place, though many solutions delegated this to an off-network cloud system. Using a remote back-end forced systems to depend on an external system to provide FT, leading to a higher latency and a SPoF.

In [2], the SPoF issue with the cloud was acknowledged as a reliability threat, and therefore some cloud services were passed down to the edge. Then, during cloud unavailability, the edge provided a subset of cloud services as a form of graceful

degradation. This demonstrated FT-support portability, by showing how it could fluidly shift across the infrastructure to provide graceful degradation.

Fundamentally, data from low-level hardware are the driving force of all IoT systems, so it is the question of how reliably the data can be collected and how effective FT support is at ensuring system reliability. FT support must therefore be delegated to some entities (e.g. the edge, fog, cloud) that have a larger (or entire) view of the system and can react rapidly to errors due to the heightened system visibility.

Chapter 4

Fault-Tolerance Framework Design

Resilience and availability are important design considerations to ensure dependability in IoT systems [21]. However, IoT systems are highly distributed, scalable, and heterogeneous, which raises the question of where best to provide FT support. This is important because underlying IoT hardware is typically error prone due to a reliance on battery power and unreliable wireless data transmission.

In this chapter, an FT-support architecture based on *microservices* is proposed that enables FT support to be 'plugged into' existing IoT systems. Microservices break down the monolithic *service-oriented architecture* (SOA) into smaller applications with individual responsibilities that can be deployed, scaled, and tested independently [192, 71, 65, 188].

The proposed microservices architecture provides a scalable means of applying real-time and predictive FT support to IoT systems. The data extracted from system monitoring provides *fault patterning*, where faults are assessed with respect to system context so that the system can learn to identify when fault activations are likely to occur based on prior experience with faults. By identifying correlations between faults, the system can proactively handle them before they activate.

4.1 Fault-Tolerance Architecture

There are many factors that influence where FT support is best placed in an IoT system. These include:

- **Architecture**. Most IoT systems attempt to abstract out underlying hardware

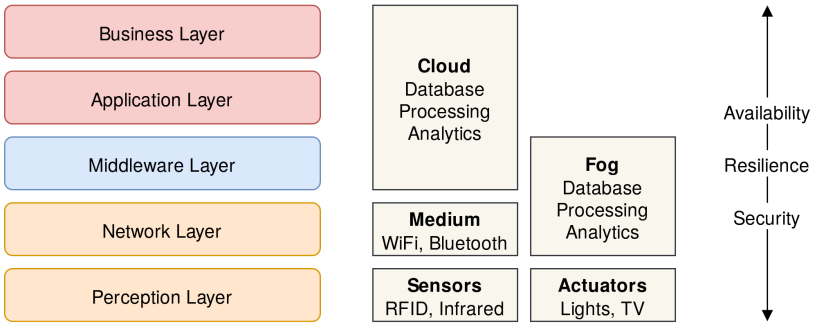

Figure 4.1: The generic 5-tier IoT architecture (left) and corresponding hardware components (middle) that define the infrastructure of IoT systems. Availability, resilience, and security are applicable across the IoT system infrastructure (right).

using middleware so that a homogeneous representation of the system can be provided for applications. Generic 3- and 5-tier IoT architectures have been proposed [210, 107] where middleware sits between devices and applications in order to expose data to applications and hide device complexities (Figure 4.1). However, there is ambiguity in terms of what hardware maps to which layer, so IoT architectures require a greater focus on logical and physical views so faults can be better understood.

- **Scale**. FT support needs to be able to work effectively in a network of potentially millions of devices. Many have proposed the edge/fog to alleviate data flow to the cloud so that transient storage and analytics can occur at the edge of the device network to provide cloud-like services with low latency [55, 62, 177, 136]. However, the fog is distributed like the devices from which data is collected, so reliability concerns still apply.

- **Fault Containment**. FT support should be pushed as close to the potential error source as possible, to minimise error propagation and reduce detection latency. Bauer [22] describes a 'hierarchy of containers' for fault containment. For example, a subroutine is contained within a transaction, which is within a user session, which is within an application (i.e. the 'outermost' container). If a container cannot handle an error, an outer container can try to.

The proposed architecture is an integration of four microservices, where two provide FT support in complementary ways: (1) *Real-Time FT*, which provides real-time

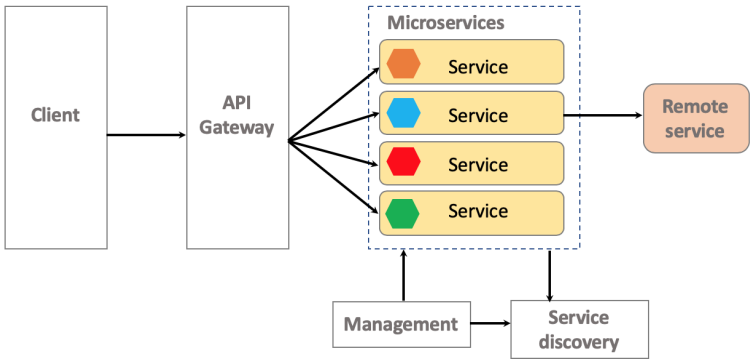

Figure 4.2: A reference microservices architecture.

data stream analysis using CEP for reactive FT; and (2) *Predictive FT*, which provides predictive analysis using ML for proactive FT. The primary goals of the architecture are to expose as much of the system state as possible and to adopt a standard method of acquiring the interfaces of system services. In fulfilling these goals, the system can provide services that allow open access to as much system information as possible. This enables FT support to seamlessly 'plug into' the system and acquire all of the necessary interfaces to provide effective support, by consuming the data it needs to detect and recover from errors. The architecture addresses the following three categories of interoperability [152].

Platform Interoperability Existing IoT platforms encounter many challenges due to diverse technologies in use e.g. operating systems, programming languages, and architectures. This, in turn, hampers the ability for developers to build cross-platform and cross-domain IoT applications [33, 152]. Arunkumar et al. [12] stated that platform interoperability could be achieved by *platform source portability* and *machine image portability*, the latter of which would enable application portability across multiple cloud infrastructures via virtualisation. Microservices offer developers the ability to not only deploy their system across clouds, but to also push portions of the system towards the network edge, which would provide an intermediary 'fog layer' between the sensor network(s) and the cloud.

Microservices are way of designing software applications as suites of independently deployable services. A microservices-based application architecture comprises a collection of small, autonomous, and self-contained services which are built to serve a

single business functionality or capability (Figure 4.2). Each microservice provides an API endpoint, which is often but not always a stateless REST API, that can be accessed over HTTP(S).

Syntactic Interoperability Syntactic interoperability refers to the interoperation of the format and data structures used during information exchange between heterogeneous IoT system entities [152]. In recent years, REST web services and APIs have become pervasive in Web, mobile, cloud, and IoT applications [189]. They are often coupled with the *JavaScript Object Notation* (JSON) data format because, together, they form a lightweight alternative to previously popular technologies, such as *Simple Object Access Protocol* (SOAP) and *Extensible Markup Language* (XML) [39]. An example protocol is the *OpenAPI Specification* (OAS) [153], which defines a standard language-agnostic description for REST APIs, enabling machines to understand the capabilities of a service without requiring access to additional documentation.

As with many microservice architectures, communication within the proposed FT architecture is conducted via a REST architecture style, and data is exchanged using the JSON format. Other protocols suitable for IoT include: *Constrained Application Protocol* (CoAP), *MQ Telemetry Transport* (MQTT), and *Extensible Messaging and Presence Protocol* (XMPP). However, the advantage of REST is that virtually all cloud platforms support it [79], making it the ideal choice for encouraging interoperability across IoT systems.

Semantic Interoperability Kiljander et al. [108] defined semantic interoperability as the technologies that enable information meaning (i.e. *semantics*) to be shared by communicating parties. They stated that this requirement had not been problematic for human users of communication systems because the semantics of information were processed *by* human users; hardware simply mediated the information. However, in IoT, devices also become 'users' that need to communicate directly with each other and interpret the meaning of information at runtime.

An example protocol to facilitate this requirement is the *Web Thing API* [143] that provides a standard approach for describing physical devices. It is designed to al-

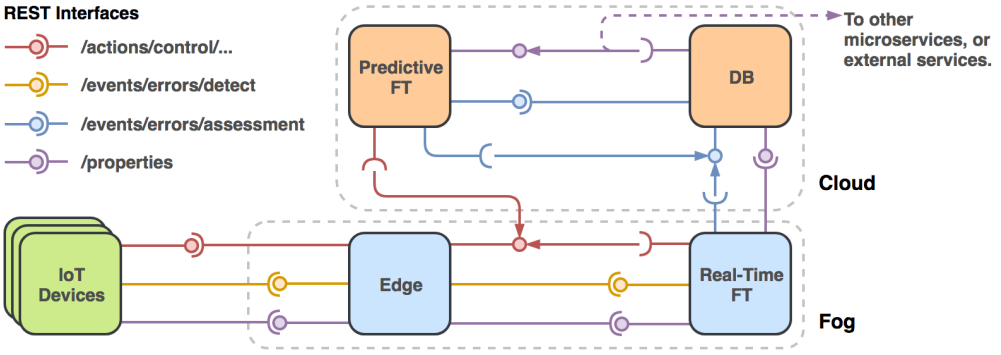

Figure 4.3: The interfaces between IoT devices and the four proposed microservices.

low access to device properties, request the execution of actions that the device can perform, and subscribe to events that occur within the device. The proposed architecture's REST interface (Figure 4.3) is based on three overarching categories defined in the Web Thing API, namely:

- **/properties**. This describes attributes of an IoT 'thing' as well as internal information about system devices and microservices.
- **/events**. This describes events that occur within the system and on IoT devices. The **/events/errors/detect** interface is where devices can use REST to `POST` errors that they detect or receive. The **/events/errors/assessment** interface enables microservices to annunciate that they have provided some kind of recovery to handle an error.
- **/actions**. This describes system functionality to be performed by a microservice. The **/actions/control/...** interface enables a microservice to control the functionality of another.

When a microservice registers itself with a service broker, it defines which categories (i.e. properties, events, actions) it wishes to subscribe to. Other microservices, after registration, emit data to the subscribers. The microservices are discussed next.

4.1.1 Microservices

Edge This microservice provides an entry point for an IoT device to pass its properties (Table 4.1) to the wider system, and for other microservices to interact with devices. It provides REST interfaces for devices to submit their properties and errors

Table 4.1: The data transmitted when sending a property using the **/properties** interface.

Field	Description	Example
sourceAddr	The IP address of the property source.	192.168.1.2
entryAddr	The IP address of the microservice that first received the property.	192.168.1.3
name	The name of the property.	Temperature
type	The type of the property.	Number
unit	The unit for the type.	Celsius
value	The value of the property.	20.5
timestamp	When the property was created, in epoch milliseconds.	1520168977817

and to receive commands for controlling IoT device actuators. Edge relays all of its received data to other microservices that are subscribed to it, and any actions that are sent to Edge are either executed internally or relayed to an IoT device.

Real-Time FT This microservice acts as a 'firewall' that only permits 'reasonable' properties from reaching DB (below) once the stream of properties have been passed through the CEP engine (Figure 4.4), along with any error-detection events that may also exist in the data stream. It analyses streams of 'primitive events' (i.e. properties, error events), then combines them to define and detect a number high-level, complex situations (i.e. newly detected errors) [60]. CEP provides an intelligent way to handle errors because it can enable an IoT system designer to define recovery strategies based on many errors rather than just one. For example, if five IoT devices fail within three seconds, the CEP system might consider that the gateway to which the devices connect has failed, rather than the devices themselves.

Errors can be produced based upon properties alone. For example, if a property's value spikes and deviates from the average, the system can tell the Edge to isolate the device producing the property. Additionally, errors can be produced by combining properties and errors for more intelligent error analysis and recovery.

One key benefit of the microservices architecture is that, if one microservice crashes, it does not bring down the entire application. However, other failure types (Section 2.1.2) have the potential to cascade into other areas of the system because of

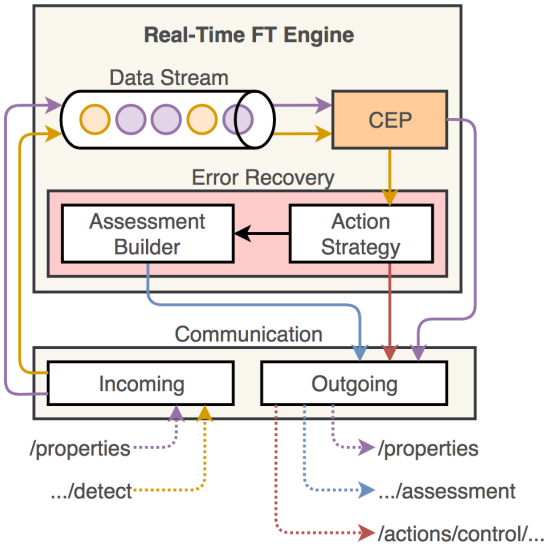

Figure 4.4: The architecture of the Real-Time FT microservice. Purple arrows indicate the flow of properties data, yellow for detected errors, blue for error assessments, red for error recovery actions, and black for internal data.

data (or a lack of it). Checking data *reasonableness* is a challenge in IoT because what constitutes 'reasonable' data is highly context dependent. For example, high temperatures in an office might be reasonable at 1pm, but not at 1am.

DB This microservice is a back-end database service that receives data for (authorised) services to subscribe to. Predictive FT (below) subscribes to DB to consume its properties and error-assessment events.

Predictive FT In IoT, data is constantly flowing from sources to sinks, capturing the latest state of the system and its physical environment. To preempt faults, predictions must be made using this live, continuous data stream. For this, an *online learning* (OL) approach is applicable. In OL, a sequence of hypotheses $f = (f_1, ..., f_{m+1})$ are produced over time, where f_1 is an arbitrary initial hypothesis and f_i for $i > 1$ is the hypothesis of the $(i-1)$th example [109].

Moreover, there also exists *batch learning* (BL), where a single predictor is generated based upon an entire dataset. OL is trained incrementally on continuous stream data, and the algorithm updates and adapts on-the-fly [76], whereas BL can adapt to change if the training and launching of each new algorithm is automated. However, continually training a BL algorithm from scratch on *all* current and prior data is

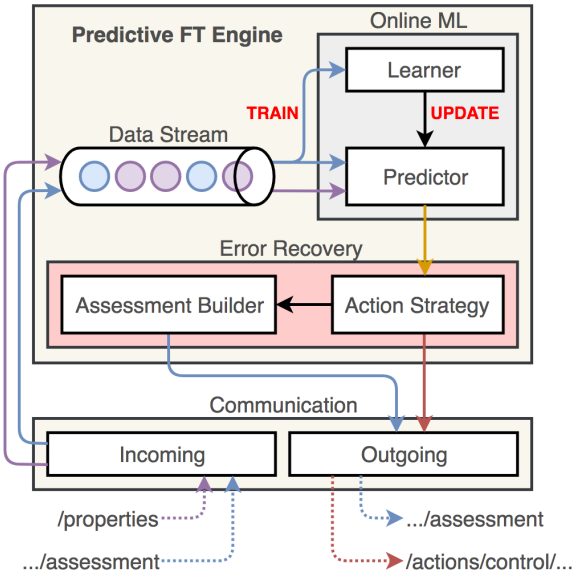

Figure 4.5: The architecture of the Predictive FT microservice. Arrow colours are the same as in Figure 4.4.

a computationally expensive process that requires far more storage space for this ever-expanding dataset.

OL can discard data once it has used it, but is more susceptible to *concept drift*, whereby the input distribution with which an OL algorithm is trained changes and the algorithm's accuracy lowers over time [85]. Current OL techniques exist as extensions of established algorithms (e.g. Support Vector Machines, Bayes), ensemble learning variants (e.g. Online Random Forests), and algorithms that are online by design (e.g. K-Nearest Neighbor) [85, 112].

Predictive FT receives error assessments and properties which are then fed into Learner (Figure 4.5) to train the algorithm to: (1) *identify* errors and the system state(s) that lead to them; (2) *learn* how the system attempts to recover from errors; and (3) *evaluate* the effectiveness of the recovery strategies, so that only effective recovery strategies are learned from. Using this knowledge, fault patterns can be generated and, using subsequent system data, help to probabilistically infer whether errors are likely to happen imminently in the future.

Butzin et al. [38] identified the fog as an enabling technology for containerisation in IoT, which is a key tool for deploying microservices, and proposed distributing microservices across the fog and cloud (Figure 4.3). The fog provides a network

Table 4.2: The data transmitted when sending an error detection event using the **/events/errors/detect** interface.

Field	Description	Example
sourceAddr	The IP address of the error source.	192.168.1.2
scope	Whether the error occurred due to internal or external factors.	Internal
ingredient	The ingredient that applies to the error.	Hardware
category	The error category with respect to the ingredient.	FRU
scenario	The type of failure with respect to the category.	Sensor Failure
fault	The (hypothesised) cause of the error.	PIR Sensor
persistence	Whether it was a transient, intermittent, or permanent fault.	Permanent
description	A human-readable description of the error.	"Cannot activate PIR sensor."
timestamp	When the error was detected, in epoch milliseconds.	1520168977817

with a gateway to a subset of services without long-range connections to the cloud, enabling low latency and rapid response times for the Real-Time FT microservice. As Predictive FT is expected to be in the cloud, it can be a shared service for all system clients, where error assessments are 'crowdsourced' to improve the ML models over time.

4.1.2 Error Events

The interfaces of the proposed architecture (Figure 4.3) enable FT support via the exchange of important event data. Properties are used for data-centric error detection. Error detection triggers Real-Time FT to perform an assessment of how the error should be recovered from, and passes this on to Predictive FT so that it can learn what faults Real-Time FT encountered and how it attempted to tolerate the faults. Key system events are as follows.

Error Detection As shown in Table 4.2, the error-detection event consists of the error source, followed by several categories that make a top-down analysis of the error, down to the affected hardware or software that triggered it (i.e. the *fault*). The format is based upon the approach proposed by Bauer [22], where ingredients

Table 4.3: The data transmitted when sending an error assessment event using the **/events/errors/assessment** interface.

Field	Description	Example
pattern	The properties and errors that caused the detected error.	Table 4.1 & 4.2
error	The detected error.	Table 4.2
actions	The actions taken to recover from the detected error.	Section 4.1.2
approach	Whether a reactive or proactive recovery approach was used.	Reactive
timestamp	When the assessment was created, in epoch milliseconds.	1520168977817

from the 8I framework (Section 2.1.2) are mapped to error categories and scenarios to achieve a systematic approach of defining test cases. Bauer also provides error categories, namely: *field-replaceable units* (FRUs), programming, data inconsistency, redundancy, system power, network, application protocol, and procedural errors.

Error Assessment The error assessment event is the product of the Real-Time FT microservice handling error events and inferring errors based on prior data. As shown in Table 4.3, it comprises a list of errors and properties that are used to detect a new error, as well as actions taken to try to recover from the new error. The chosen actions are based upon the hypothesised fault that potentially caused the error. A fault can only be *hypothesised* rather than directly identified because only the manifestations of a fault (i.e. its errors) are detectable by an FT-support system, and errors are not unique to any one specific fault [122, 80].

Actions When recovering from faults, systems can employ *backward* and *forward* error recovery mechanisms, where the former tries to restore a previous error-free state, and the latter tries to move into a new, error-free state [163]. In IoT, data, and the services that rely on it, help to create virtual entities that resemble physical entities in the real world by monitoring their states with sensors and actuators [21]. Therefore, forward error recovery is the ideal option to keep the system focussed upon the latest data and environmental states.

Erroneous sensor data can hinder system performance. If DB stores erroneous data,

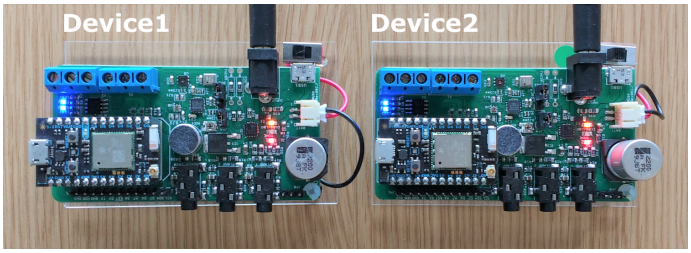

Figure 4.6: Device1 and Device2 used in the example scenario.

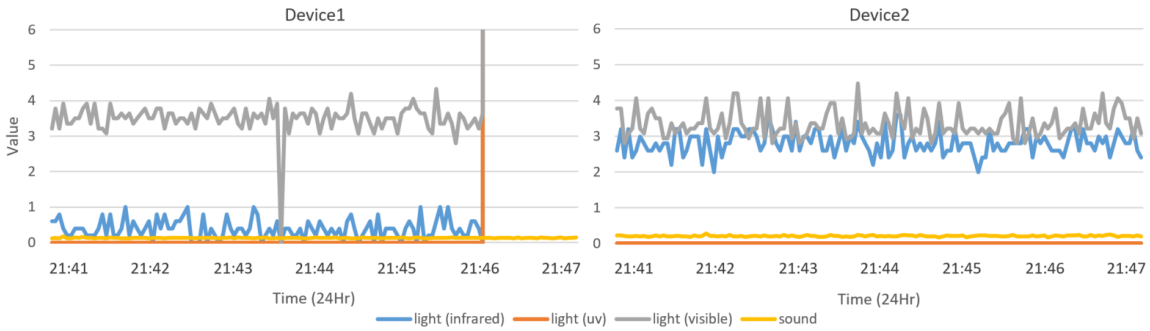

Figure 4.7: Data showing four properties over a six minute period on Device1 (left) and Device2 (right). Device1 experiences a stuck-at fault from 21:46 onwards, whereas Device2 is running as normal at that time.

it can harm other services that rely on its data. If Predictive FT consumes erroneous data, its predictive models could suffer from concept drift. To combat this, the **/actions/control/block** interface on Edge is called by Real-Time FT to block data from, and interactions with, IoT devices. If the CEP system (Figure 4.4) flags a property as erroneous, then its `sourceAddr` and `name` (Table 4.1) are sent to Edge at `entryAddr` to block it and prevent further erroneous data from propagating through the system.

4.1.3 Scenario

To demonstrate Real-Time FT and Predictive FT, a simple experiment was set up with live data that was generated using two IoT boards, *Device1* and *Device2* (Figure 4.6). Each board had 15 sensors, which supported the 802.11 Wi-Fi protocol and were controlled by a STM32F205, 120Mhz ARM Cortex M3 processor. The experiment used infrared, *ultraviolet* (UV), and visible light sensors, and a microphone for sound detection. Two experiments were performed, where the devices were left in operation

for two weeks in both experiments. They were given constant power so errors and faults that were not power-related could be observed.

In the first experiment (Figure 4.7), visible light dropped to 0 between 21:43 and 21:44 on Device1 before returning to normal. Then, a stuck-at fault occurred on all light sensors on Device1, where their values exceeded 100 and remained constant until the end of the experiment. This pattern occurred *in both experiments on Device1*, where a drop in visible light happened minutes before a major stuck-at fault on all light sensors. In this scenario, the proposed system could perform the following:

1. The CEP system (Figure 4.4) identifies Device1's visible light drop to 0 and flags it as erroneous data, because the value deviates from the last 10 seconds of data by a significant margin. An error assessment targets Device1's visible light sensor as the cause and considers it to be a transient fault. The action taken is to drop the data, as the values return to normal thereafter.

2. When the major stuck-at fault occurs, the assessment identifies the three light sensors as the cause. Real-Time FT contacts the Edge microservice via the interface **/actions/control/block** in order to block the three light properties from Device1. The sound property is still accepted as it is not producing erroneous data.

3. Each time the last two steps occur, there are two error assessments. Predictive FT receives these assessments each time and identifies the fault pattern that the first error is often followed minutes later by a stuck-at fault on Device1.

4. When the drop to 0 occurs, Predictive FT identifies a high probability of a stuck-at fault and notifies Edge to activate a replica to produce these properties so that, if/when Device1 has the predicted stuck-at fault, another sensor is ready to take over. Otherwise, Edge can just block the properties.

4.2 Vulnerabilities, Faults, and Failures

Current FT-support implementations in IoT systems are static, tightly coupled, and inflexible. For example: (1) they are designed for a specific architecture and ap-

plication [86, 208]; (2) they do not scale beyond small (decentralised) solutions [186, 117]; and (3) they provide solutions to specific faults, such as link failures [103]. This is problematic because IoT systems are expected to continuously evolve in order to handle new services, features, and devices that had not been anticipated when the system was first designed, and FT support needs to evolve with these changes.

Additionally, many adopt FT solutions already widely explored in distributed systems, such as hardware redundancy [86], check-pointing [96], and traffic re-routing [103]. Their error-detection approaches are implemented in an application-specific manner, where faults are identified *a priori* by system designers, with checks that only detect errors in specific scenarios and contexts. To address these shortcomings, the *Vulnerabilities, Faults, and Failures* (VFF) framework is proposed.

The VFF framework generically categorises system defects and their potential effects on a system. It is designed to consider the relationship between system vulnerabilities, faults, and failures, where these three attributes help to categorise defects so that an erroneous scenario common to all IoT systems has a common expression in VFF. That is, each scenario can be described as: *a vulnerability v, exploited by fault f, may lead to failure s*. This facilitates the design of modular, reusable error detection and recovery techniques to generically handle system defects common to all IoT systems. Also, by understanding the vulnerabilities, faults, and failures that can occur in a given system, it provides a basis for considering what errors can occur for each VFF tuple, and then what errors are common among tuples.

4.2.1 Attributes

The attributes of the VFF framework are based upon three concepts discussed in Chapter 2, which are briefly outlined next.

Vulnerabilities The 8I framework (Section 2.1.2) was developed for conducting vulnerability analysis on internal and external aspects of a system and identifies that the reliability and security of communications is vital for continuous system operation [168]. What makes the 8I framework so useful for VFF is that it is com-

prehensive enough to consider not only product-attributable vulnerabilities, but also ones attributable to enterprise-level and external vulnerabilities also [22].

For the remainder of the thesis, the eight 'ingredients' of the 8I framework are referred to as *vulnerabilities*, specifically: human, policy, hardware, software, networks, payload, environment, and power.

Faults A *fault* is the adjudged or hypothesised cause of error(s) [163], and therefore a fault activation is the precondition necessary to provide an FT-support system with errors to detect. Faults can be placed into three major groupings [16]: development, physical, and interaction faults.

If a system vulnerability has been identified, then the ways in which it can cause damage to a system in terms of development (i.e. software), physical (i.e. hardware), or interaction (i.e. human or environment) can be explored, and the errors that might occur therefore.

Failures Failure semantics categorise allowable server behaviours that occur in distributed systems so that developers can understand the likely failures a system might exhibit, in order to develop relevant recovery strategies. They include [73, 190]: omission, crash, timing, response, and arbitrary failures. The VFF framework does not include the arbitrary failure because it simply refers to the arbitrary occurrence of any of the other previously defined classes [73].

A fault and its corresponding errors are what enable failures to manifest [98, 122]. Given a particular vulnerability and fault grouping, the failure attributes help to further categorise a system defect in terms of the ways in which the system might fail due to the fault's errors. This enables the failure semantics of the system to be properly understood and defined [73].

4.2.2 Applicability

In Table 4.4, the applicability between the attributes of VFF to IoT systems is outlined. Each checkmarked table cell represents an applicable tuple of a vulnerability,

Table 4.4: Applicability between vulnerabilities, faults, and failures (checkmarked).

Vulnerability	**Fault**	**Failure**			
		Omission	Crash	Timing	Response
Hardware	Development	✓	✓	✓	✓
	Physical		✓		✓
	Interaction		✓		✓
Software	Development	✓	✓	✓	✓
	Physical				
	Interaction		✓		✓
Networks	Development	✓	✓	✓	✓
	Physical	✓	✓	✓	✓
	Interaction	✓	✓	✓	✓
Payload	Development		✓		✓
	Physical				
	Interaction		✓		✓
Environment	Development				
	Physical				
	Interaction	✓	✓	✓	✓
Power	Development				
	Physical		✓		
	Interaction		✓		
Human	Development				
	Physical		✓		✓
	Interaction		✓		✓
Policy	Development	✓	✓	✓	✓
	Physical		✓		✓
	Interaction		✓		✓

fault, and failure that can be addressed by the proposed FT framework. Development faults consider how residual software bugs can cause erroneous behaviour in a deployed system. Physical faults refer to hardware failures that are not caused by an interaction with the hardware i.e. naturally occurring failures. Interaction faults are erroneous situations, accidentally or maliciously caused by a human, environment, or external system.

With hardware vulnerabilities, all fault types have the potential to influence crashes in hardware, and introducing erroneous responses from the hardware. That is also the case for software, without the physical faults. In hardware and software development, all failure types are possible if the design and development of them contains defects. Networks vulnerabilities can introduce all failures if the network is poorly developed, endures physical problems, or is (accidentally or maliciously) tampered with. Payloads have the potential to cause harm if they are able to exploit underlying system defects (e.g. buffer overflow).

Hardware systems are vulnerable to environmental conditions. This is especially important in IoT, where hardware could be situated in a wide range of harsh, dynamic environments. A lack of power will cause crashes to any affected hardware and software and can occur spontaneously or by system interaction. Human interaction with an IoT system might cause crashes to, and alterations in, hardware and software. The policies put in place during the development and maintenance of the system can influence these types of failures also.

4.3 Summary

IoT systems are highly distributed, scalable, and heterogeneous, which raises the question of where best to provide FT. In this chapter, an FT-support architecture based on microservices was proposed to enable FT support to be 'plugged into' existing IoT systems. The architecture integrated four microservices, where two provided FT support in complementary ways: the first provided real-time data stream analysis using CEP for reactive FT, and the second provided predictive analysis using ML for

proactive FT. It was designed to address three types of interoperability: platform, syntactic, and semantic interoperability.

Furthermore, the VFF framework was proposed to generically categorise system defects by vulnerabilities, faults, and failures, in order to categorise defects so that an erroneous scenario common to all IoT systems has a common expression in VFF. This facilitates the design of modular, reusable error detection and recovery techniques to categorically handle defects in IoT systems.

Chapter 5

Complex Patterns of Failure

To define a generic FT-support solution for IoT systems, it is important that errors are defined in a language-agnostic way, so that FT can be applied as a software pattern. It must also be interoperable and portable, so that FT support can be easily 'plugged into' any existing IoT system. This is facilitated by an adherence to standards and protocols that allow easy inclusion of FT-support and for data to be easily exposed to it.

This chapter proposes *Complex Patterns of Failure* (CPoF) as the means of detecting and recovering from system errors, where error-detection events are defined as *nondeterministic finite automata* (NFAs) to be implemented in NFA-based *Complex Event Processing* (CEP) systems. The chapter proposes NFAs that are closely related to error checks in FT literature for the detection of errors in IoT systems via data-stream analysis facilitated by CEP.

Coupled with the tuples from the VFF framework (Section 4.2), system defects can be systematically defined so that, for VFF tuples, there are corresponding NFA(s) to handle them. Furthermore, error-detection events can be recursively fed back into the CEP data stream for reuse in other NFAs, a process termed *Complexity via Recursion* (CvR), which concludes this chapter.

5.1 Complex Event Processing

IoT systems demand continuous information processing and analysis from a wide variety of geographically diverse and potentially unreliable sources. Modern scientific

applications are driving the need for heterogeneous sensor networks and methods to detect events in real time that arise from complex correlations of measurements made by independent sensing devices [191]. Event detection is a key requirement of intelligent monitoring systems along with data preparation and visualisation. However, many legacy systems are built on relational databases that require data storage and indexing before being processed, which creates a significant latency issue that is unsuited for real-time, reactive event processing [187].

The core challenge when analysing high-velocity streaming data is how to infer the occurrence of interesting and *complex* situations in the environment. For example, the complex situation of a *fire* may be derived from the correlation of raw numerical IoT sensor data, such as a rapid increase in temperature and then smoke detection following shortly thereafter [68].

CEP has been proposed as a means of inferring complex situations in data and is considered to be the paradigm of choice for the development of monitoring and reactive applications [36]. It accomplishes this by processing heterogeneous streams of input data (i.e. *primitive events*) and, via the analysis of patterns in primitive events, inferring the existence of high-level, complex phenomena (i.e. *composite events*) [60]. CEP operates not only on sets of events, but also *causal* relationships between events. The ability to analyse causality is beneficial in many circumstances, such as in distributed systems where events between nodes have various relationships to one another e.g. A *happened before* B [130].

While many existing CEP systems are designed for general-purpose event processing [90, 32, 58], there also exist ones built for specific purposes. For example: (1) mobile-driven CEP that automatically adapts event processing to the user's location [154]; (2) predictive CEP that can anticipate disruptive events caused by incomplete or uncertain information [149]; and (3) systems suitable for various layers of the typical IoT infrastructure (Figure 4.1) e.g. edge [50], fog [131], and cloud [139].

5.1.1 Nondeterministic Finite Automata

This thesis focusses on NFA-based CEP systems because NFA is the established mechanism on which most CEP systems are based [75]. NFAs were introduced in 1959 by Rabin and Scott [164] in response to the rising popularity of *finite-state machines* (FSMs) wherein a finite number of internal states could be used for memory and computation. NFAs enable the construction of automata that are more powerful than conventional FSMs because they allow several state transitions at each state, offering more a versatile machine that is closer to the ideal of the Turing machine while ensuring only a preassigned amount of tape.

Events are typically processed as follows [60]. Instances of NFAs, or *runs*, are created at runtime to handle potential instances of complex events. When a new event e arrives, the CEP system checks whether it can satisfy the transition predicate for the current state of any existing runs. If it can, then a run transitions to its next state until it reaches its final (i.e. accepting) state, which causes a composite event to be generated. Runs are 'halted' (i.e. removed) when they are no longer able to proceed to a next state and are not in the accepting state. A more comprehensive demonstration of this process is provided in Section 6.1.1.

While NFA-based CEP is the predominant approach, there exist other means of CEP in literature. For example: (1) tree structures [141], where query expressions are transformed into an internal tree representation and primitive events are stored in leaf nodes; (2) event derivation under uncertainty (e.g. inaccurate, unreliable, or lost data) using a Bayesian network to derive new events [204]; (3) FSMs [6], which provide a simpler way to express complex events than NFAs; and (4) *pushdown automata* (PDAs) [41], which provide a more capable automata model than NFAs.

5.1.2 Language Specification

For CEP systems to perform the pattern-matching necessary to detect composite events, a core language $\mathcal{L}$ must exist for pattern querying, which includes necessary constructs to be useful in real-world applications and each event represents an occur-

rence of interest [215]. The typical language specification for CEP systems include the following operators [59]:

- **Single-Item Operators**. To *filter* and *transform* items. For example, temperature items can be filtered to keep those within a given range, and transformed to convert those in Celsius to Fahrenheit.
- **Logic Operators**. To define rules that combine the detection of several items:
  - **Conjunction**. Similar to the "AND" operation: items $I_1, I_2, ..., I_n$ are satisfied when all of them have been detected.
  - **Disjunction**. Similar to the "OR" operation: items $I_1, I_2, ..., I_n$ are satisfied when at least one of them has been detected.
  - **Repetition**. A special case of the conjunction operation: item I is satisfied when it has been detected m times and not more than n times, where $n > m$.
  - **Negation**. Similar to the "NOT" operation: item I is satisfied when I is not detected.
- **Sequences**. Similar to logic operators but items are *order dependent*. That is, items $I_1, I_2, ..., I_n$ are satisfied when they have been detected in the specified order (e.g. by timestamp).
- **Iterations**. A special case of sequences where the length of the sequence is not known *a priori*, enabling unbounded sequences. The 'Kleene plus' operator is used to express *one or more* items I.
- **Windows**. Specify portions of input flow to be considered. This constraint is designed to limit items to those only within a given time frame, and to ensure the termination of unbounded iterations:
  - **Time-Based**. The bounds are defined as a function of time. For example, all items in the last 30 seconds.
  - **Count-Based**. The bounds depend on the number of items I in the window. For example, the last 10 items to arrive.

- **Event Selection**. Events can be widely dispersed over one or more input streams and are therefore not always *contiguous*. The following strategies have been proposed to handle this:
  - **Strict Contiguity**. The most stringent strategy as it requires input events to be contiguous i.e. a second matching event follows directly after the first without any non-matching events in between.
  - **Skip till next match (STNM)**. Skip irrelevant events until the next relevant event occurs to match more of the pattern.
  - **Skip till any match (STAM)**. Enables nondeterministic actions: a relevant event causes the current partial match to be cloned into a new instance that skips the current event to reserve opportunities for additional future matches.

5.1.3 Context Awareness

A key motivation for using CEP to provide FT support in IoT is because of its ability to realise *context-aware* computing. A *context* is any information that can characterise the situation of an entity (i.e. people, places, objects), and a system is context-aware if it uses context to provide relevant information and services to the user [1]. The system stores context information linked to sensor data so that data interpretation and *machine-to-machine* communication can be done easily [157].

Hasan et al. [92] considered context awareness over large-scale sensor networks via dynamic enrichment of information flows, which were combined with CEP, as the means to realise situation awareness. Barbero et al. [20] proposed the *Concept Reply* IoT platform that provided support for context-aware application deployment throughout the low-level and middleware layers of IoT systems. It contained a reasoning framework and event-based processing agents that incorporated CEP for content-based filtering. Nallaperuma et al. [145] proposed *Incremental Knowledge Acquisition and Self Learning* (IKASL), an unsupervised incremental learning algorithm for detection and adaption of concept drifts in data by monitoring changes

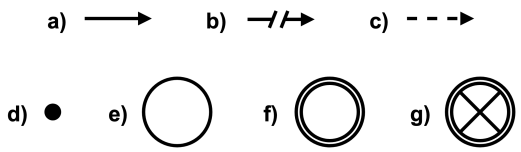

Figure 5.1: NFA diagram symbols: (a) transition between states using STNM; (b) same as (a), omitting some intermediary states; (c) arrow pointing to composite event(s) produced after run acceptance; (d) a starting point; (e) a state; (f) an accepting final state; (g) a non-accepting halt state.

in context i.e. location, time, activity and identity. They evaluated its efficacy on a motor-traffic dataset to detect drifts in the number of vehicles on the road.

Maarala et al. [133] focussed on the issue of providing and acquiring knowledge in IoT environments with a study on Semantic Web technologies that could facilitate context-awareness, interoperability, and reasoning in IoT. They identified that the publish/subscribe message-exchange scheme supported topic and content-based message routing and aggregation methods, and enabled context-based information fusion from multiple heterogeneous data sources. This supports the decision to use CEP systems for context-aware computing as it extends the functionality of publish/subscribe systems by increasing the expressive power of the subscription language, in order to consider complex event patterns that involve the occurrence of multiple related events [59].

5.2 Error-Detection Automata

The CPoF approach involves designing error-detection NFAs that can be used across NFA-based CEP systems to check for erroneous system behaviour via real-time data analysis. The checks are designed to be modular, reusable, and able to detect low-level data errors that can occur in IoT systems.

5.2.1 Automata Model

The automata model in this thesis is based on the model described in [179], and uses the standard NFA notation shown in Figure 5.1. 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: (1) a value e^v, i.e. its data; (2) an origin e^o, i.e. where it was generated; and (3) a timestamp e^t, i.e. when it was generated. All currently accepted events are in a history set H, accessible by the current state of a run. The STNM selection policy is assumed for state transitions because strict contiguity is unsuitable for analysing high-volume, heterogeneous data in IoT. S_A is the accepting state of an NFA.

A dashed arrow (Figure 5.1c) points from S_A to a composite error-detection event d that represents the detected error, and an error-recovery event r that represents the attempted recovery strategy executed to handle the error; recovery is not mandatory. The events that caused a composite event to be generated are called the *pattern* of the event, which is equal to H when a run's state is S_A. State S_F is the halting (i.e. non-accepting) state that causes a run to halt when it transitions to S_F.

A run might transition to S_F, regardless of its current state, to provide *state clearance*. This is where runs are prematurely halted to prevent the accumulation of runs that cannot realistically reach their S_A states, thus mitigating out-of-memory errors. State clearance can be implemented using: (1) a *time window*, that halts on a time elapse; and (2) an *until* predicate, that halts if true.

5.2.2 Checks

Lee et al. [122] defined an error-detection classification scheme for checking fault-tolerant systems. It identified seven error-detection checks: reasonableness, timing, reversal, replication, coding, structural, and diagnostic checks. The first four of these are explored in this section. The other three are not explored because the error-detection checks proposed in this section can also be used for coding, structural, and diagnostic checks, and therefore do not require their own automata.

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 [122]. The three data unreasonableness types explored in [128] are considered, namely: (1) *outliers*, where e^v exceeds some threshold ϵ; (2)

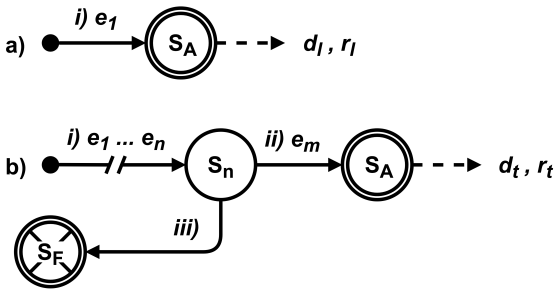

Figure 5.2: Reasonableness NFAs: (a) limit checking; and (b) trend checking.

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 5.2a), 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 [98] 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 can be considered unreasonable. The NFA in Figure 5.2b is proposed for trend checking, which calculates the slope between all relevant events that occur within time window t.

If the slope between events e_1, e_m, or an aggregate of n prior events $f(e_1, ..., e_n), e_m$, surpasses a given 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 $\{e_1, ..., e_n, e_m\}$.

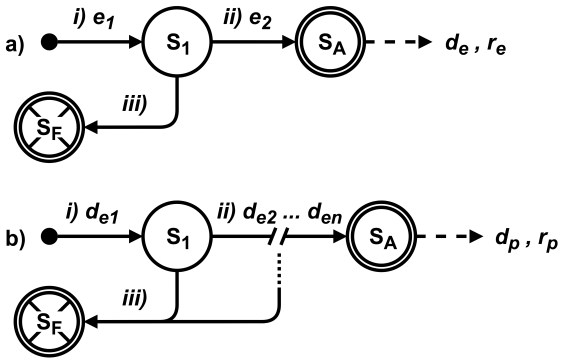

Figure 5.3: Timing NFAs: (a) performance checking; and (b) persistence checking.

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 [163]. The proposed checks are designed to 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 time ϵ.

Performance Checking Identifying 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 5.3a). 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 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 [11]. Thus, the time between e_1 and $e_2 = e^{\emptyset}$ can be calculated instead.

Persistence Checking Persistence is classified as [16, 110]: (1) *transient*: arbitrary faults that cause erroneous behaviour 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

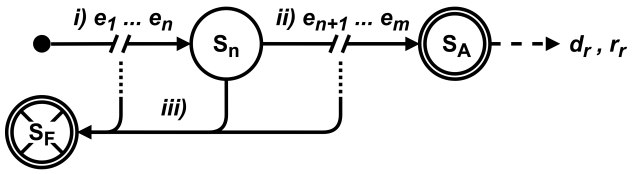

Figure 5.4: Reversal NFA for correlation checking.

event d refers some other error event d', then d is considered *transient*, because it is independent from any other detected system errors.

The automaton from Figure 5.3b considers how d_e from Figure 5.3a 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. Intermittent persistence can 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.

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 [122]. This check has predominantly deterministic applications (e.g. reading back what was just written to disk). However, the proposed correlation check considers how it can be used in scenarios to check for a relationship between two (sets of) events, where the causality between events can only be inferred, discussed next.

Correlation Checking Given $n \geq 1$ system events $e_1, ..., e_n$, there are $n \geq 1$ erroneous events $e_{n+1}, ..., e_m$ that occur afterwards within a given time frame, or halt otherwise (Figure 5.4). If they do occur, the system 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 [16].

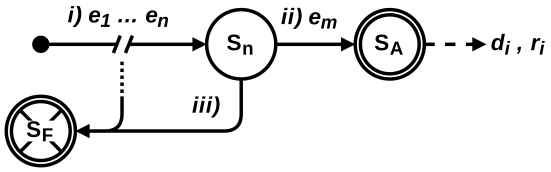

Figure 5.5: Replication NFA for inconsistency checking.

Replication

Replication checks utilise redundancy of the activity being checked. They assess discrepancies between the outputs of replicas to identify erroneous behaviour, particularly faults in physical components of hardware systems e.g. TMR (Section 2.2.1) [122]. IoT systems are expected to have large sensor networks producing comparable data that is heavily context bound. For example, the natural light in a room causes similar daily patterns, and deviations from the expected pattern can be detected without redundancy, simply from context alone.

However, the temperature of machinery might change depending on how much it is used, for example. No redundancy in temperature sensors, coupled with erroneously low temperature readings, might lead to a hardware failure if the machinery were to overheat. Having multiple sensors producing temperature readings would enable checks between readings to look for inconsistencies between temperatures, so that faulty sensors can be isolated and replaced.

Inconsistency Checking As discussed in Section 2.2.1, NVP is a form of static redundancy that provides error masking by executing n independently developed programs, identical in specification, and accepts the majority result from the n outputs [37, 203]. An NVP-like design can be adapted as an *inconsistency check* NFA (Figure 5.5) that is able to identify any inconsistency from the output of n sensors producing equivalent sensor data.

The inconsistency check performs an assessment of m events to check for inconsistencies between event values. The first n events are collected with the predicate: $\forall \dot{e} \in H | \dot{e}^o \neq e^o$, which ensures that the value from current event e has a different

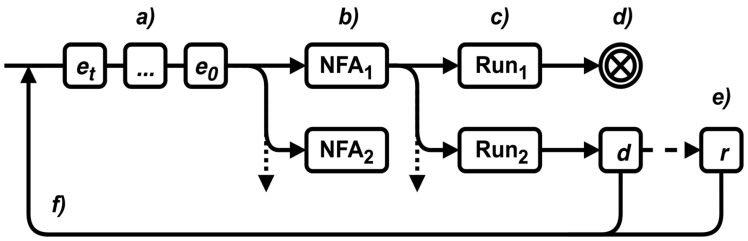

Figure 5.6: The process of CvR.

origin to all events currently consumed in event history H. The state transition for state S_n (Figure 5.5) then has the following predicate:

$$\forall \dot{e} \in H | \dot{e}^o \neq e_m^o \wedge \neg(a \leq std(\forall \dot{e}^v \in e_m \cup H) \leq b) | 0 \leq a \leq b \tag{5.1}$$

This checks the standard deviation of the values of all events to see whether it is *not* within some acceptable range. If the standard deviation is unreasonably high, then it suggests a data error, perhaps caused by sensor malfunctions or environmental errors. For discrete data, where *any* variation in value constitutes a data error, one can apply the above predicate with $a = b = 0$.

5.3 Complexity via Recursion

Instead of defining complex, monolithic NFAs to handle application-specific error scenarios, it would be favourable 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. This proposed process is referred to as *Complexity via Recursion* (CvR).

For example, a common failure in IoT systems is sensor failure, which could be checked using the performance checking NFA from the previous section, by assuming that the time difference between a sensor's last event and the current time is greater than some threshold. However, a gateway failure will cause all of its connected sensors to appear to have failed to FT support outside of the network, which would

result in the FT support an error-detection event for each sensor that has now appeared to stop producing data. Using CvR, a more complex NFA could be created to correlate the error-detection events from each sensor and, due to these events occurring in rapid succession, infer that it was actually the gateway that failed.

The process of CvR is as follows (Figure 5.6):

(a) A stream of events enter the CEP system over time.

(b) Each event is passed to the NFAs. When an event fulfils the predicate to transition to the first state of an NFA, 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 by other runs.

This approach to error definition means that common errors that are universal to any IoT system have reusable NFAs to check for them. This makes FT support easier for system designers because they simply need to identify the failures that occur in their system and pair them with appropriate NFAs.

5.3.1 Proactive Detection

Fülöp et al. [78] suggested that the value of a composite event would be at its highest if it were *proactively* generated. That is, before the real-world complex phenomena being detected had occurred. The event's value decreases over time as the composite event is generated further from the time of the complex phenomenon. This concept is applied to the FT solution proposed in this thesis, as follows (Figure 5.7):

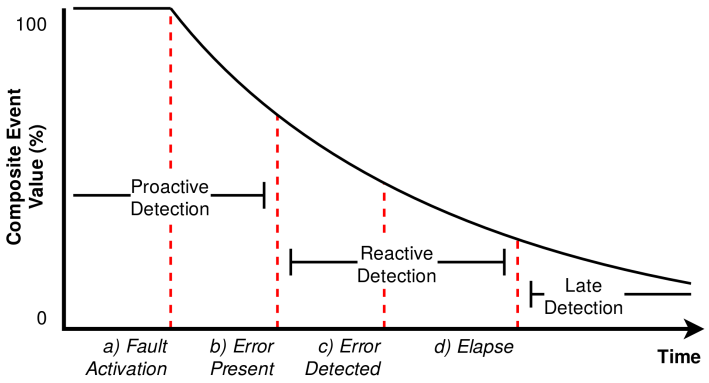

Figure 5.7: The value of generating a composite error-detection event over time when faults and errors occur in a system.

(a) Before fault activation, an error event is most valuable because it can prevent or mitigate erroneous system behaviour and reduce the overhead of error recovery.

(b) After fault activation, an error event's value begins to decrease with time the longer that its underlying fault remains active and untreated.

(c) Reactive FT is able to detect its error(s) due to their observable effects on system data.

(d) After time t has elapsed, a detection event would be considered *late*, and to detect the error thereon would become less valuable over time.

CEP provides reactive recovery because the error must have already happened in order to detect how it manifests in system data. For proactive recovery, ML is proposed as the means to learn from CEP in terms of the errors detected and recovery provided during CEP runtime. CvR is proposed as the basis for providing proactive support via supervised learning techniques to predict imminent error events using two correlation check NFAs (Section 5.2.2), as follows.

Check 1 The first correlation check NFA receives event e_1 and is followed by a second event $e_2 \in d$ within some time t, which reaches S_A and produces an error event d_r. Event d_r acts as a 'positive' classification label 1 for the dataset of a supervised learning model.

Check 2 The second correlation check NFA generates event d'_r when e_2 does *not* occur after e_1 in time t, i.e. where $e_2 = e^{\emptyset}$ (Section 5.2.2). Event d'_r acts as a 'negative' classification label 0.

Prediction With events d_r, d'_r, the prediction $\|P(e_2|e_1)\| = 1$ can be made when e_2 is likely to follow e_1 in time t and, in response, proactively performing recovery as though $e_2 \in d$ had just been reactively detected. Conversely, the prediction $\|P(\neg e_2|e_1)\| = 0$ can be made when e_2 is unlikely to occur in time t.

5.4 Summary

This chapter proposed CPoF as a means of detecting and recovering from system errors, where error-detection events were defined as NFAs to be implemented in NFA-based CEP systems. This was advantageous because it enabled errors to be defined in a language-agnostic way and promoted interoperable and portable FT design that could be easily 'plugged into' any existing IoT system.

The proposed NFAs were based on four error-detection checks from FT literature: reasonableness, timing, reversal, replication. From these checks, six NFAs were defined: (1) *limit*, which checked if an event was *not* within some defined limits; (2) *trend*, which checked the first derivative of an event and then checked if it was *not* within some defined limits; (3) *performance*, which identified event timeliness errors; (4) *persistence*, which checked the persistence of a system error as to whether it was transient, intermittent, or permanent; (5) *correlation*, which checked the correlation between system events; and (6) *inconsistency*, which checked for discrepancies between events that should have produced equivalent data.

The correlation check NFA was crucial in providing proactive FT via CvR. It could be implemented as a means of using composite events are training data for supervised learning models, in order to learn from the types of error correlations that could occur within an IoT system.

Chapter 6

Implementation

The key concepts proposed in Chapters 4 and 5 lay the foundation for an interoperable FT-support system that is able to both reactively and proactively protect against system errors via data-centric error checking. This chapter describes how the FT-support system was implemented and presents a use case that is specifically relevant to current IoT trends. The use case is of a sufficient scale, such that it can demonstrate a large array of convincing failure scenarios that the proposed FT support should handle.

A novel NFA-based CEP system, *BoboCEP*, is first proposed as a means of providing resilient FT support at the edge via the active replication of partially completed complex events. Then, two vertical farming testbeds are presented: the first is a small-scale testbed that was later scaled up to a medium-scale testbed with more hardware and failure scenarios. Finally, a dataset from a real-world vertical farm is described, with a brief explanation of how CPoF can be applied to it.

6.1 BoboCEP

Few CEP systems have been designed specifically for IoT and, to the best of my knowledge, there does not exist a CEP system that is designed to provide a resilient FT-support framework for IoT. *Long-Term CEP* (LTCEP) enabled long-term events to be detected without excessive overhead by splitting event detection into online and offline detection, which significantly reduced redundancy in intermediate state and data [132]. *Event Sharing CEP* (ESCEP) reduced wasted energy consumption

and increased processing efficiency using a hashing algorithm to decompose complex events into several intermediate events, which enabled them to be shared more easily [211]. *EdgeCEP* is a fully distributed CEP engine for the collaboration of devices at the edge network, where task assignment and delivery was accomplished using tabu search and a heuristic flow-based greedy move algorithm [50].

EdgeCEP addressed many of the design and functionality considerations for distributed edge processing. However, it is important to provide a generic means of applying FT support to IoT systems that can be pushed as close to the fallible sensor network as possible, and where FT support is, itself, fault tolerant. CEP-based FT support needs to provide *resilient* distributed long-term event processing, as with LTCEP, but where the current state of FT support is actively replicated to protect against arbitrary edge device failures.

BoboCEP is a CEP system that can detect, assess, and recover from complex, erroneous system behaviours that are detected via patterns in primitive events. Error definitions are expressed as NFAs (Section 5.1.1) and composite events represent detected errors in an IoT system. BoboCEP is designed to be distributed across the network edge on k software instances. Each instance maintains the current state of partially completed runs via *active replication*, so that $(k-1)$ instances can fail without complete FT-support service loss. This helps to facilitate: (1) *long-term error detection*, because partially completed runs maintain the same state across the edge, so they can be continued and completed on any device; and (2) *load balancing*, by having devices send their data uniformly to any edge device, as facilitated by active replication.

6.1.1 Event Processing

BoboCEP implements the automata model described in Section 5.2.1. Defining a complex event requires the definition of a series of state transitions that lead to a final state S_A, which results in a composite event being generated to represent a complex phenomenon. The events that caused a composite event to be generated are the *pattern* of the event.

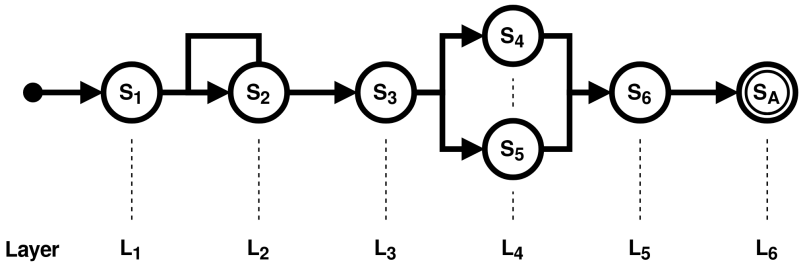

Figure 6.1: The relationship between automaton states (S_1, ..., S_A) and their labels (L_1, ..., L_6).

Due to nondeterminism in BoboCEP automata, a pattern categorises its events based upon what *layer* an automaton is current at, where each layer L contains one or more states that may be accessed nondeterministically (Figure 6.1). State S_2 is a looping state, making it a nondeterministic state because it can transition to either S_3 or back to S_2 again. Every event that causes a loop back to S_2 will be associated with layer L_2. S_3 can transition to both S_4 and S_5, which also makes this a nondeterministic state. Events for S_4 and S_5 will be associated with layer L_4.

Process The process of fulfilling an automaton in BoboCEP is similar to most other NFA-based CEP systems, as discussed in Section 5.1.1. An example of the fulfilment process in BoboCEP is described next and visualised in Figure 6.2. For this example, it is assumed that: (1) in order to transition to some state S_n, an event must be received with value $e_t^v = n$, where t is the time that the event was received; (2) a layer L_n corresponds to each state S_n; and (3) $S_4 = S_A$.

At time $t = 0$, the automaton is currently not in any state, but an event with value $e_0^v = 1$ triggers a transition to S_1. Likewise, at time $t = 1$, event $e_1^v = 2$ causes a transition to S_2. At $t = 2$, another event $e_2^v = 2$ is consumed that causes S_2 to nondeterministically loop back to itself. The automaton remains in the same state, but layer L_2 for S_2 is now associated with two events in its history of consumed events: e_1 and e_2. At $t = 3$, event $e_3^v = 3$ nondeterministically moves the automaton to state S_3. At $t = 4$, event $e_4^v = 1$ is not an event that can cause a transition to S_4. When an event is encountered that does not cause a transition, BoboCEP responds depending on which *contiguity policy* is being used, namely:

- **Strict Contiguity**. All matching events are strictly one after the other,

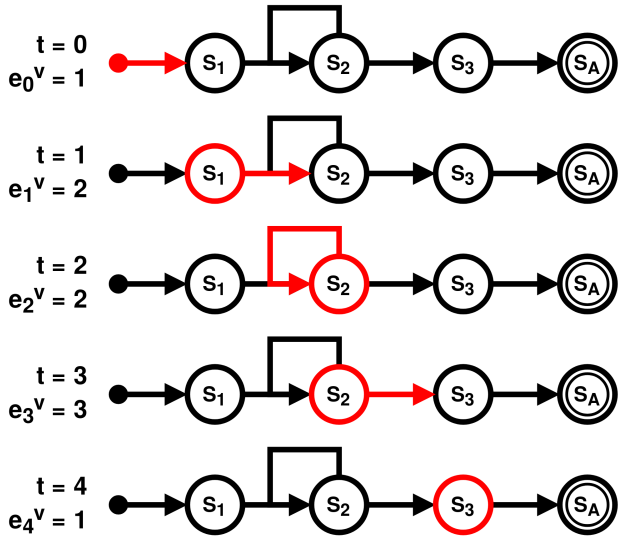

Figure 6.2: An example of run processing, adapted from Figure 2 in [60].

without any non-matching events in-between. If an event does not match, the run halts.

- **Relaxed Contiguity**. All non-matching events are ignored; the run simply waits for a matching event. This policy is equivalent to STNM in Section 5.1.2.
- **Nondeterministic Relaxed Contiguity**. The same as relaxed contiguity, but will ignore non-matching events in circumstances where are multiple next states to transition to. This applies to S_2 in Figure 6.2 because it can transition to both S_3 and back to S_2. This policy is equivalent to STAM in Section 5.1.2.

6.1.2 Architecture

BoboCEP's architecture is inspired by the *information flow processing* (IFP) functional architecture proposed by Cugola et al. [59] that describes the main functional components that are common to all IFP systems, of which CEP systems are a subset. Several key components of the IFP architecture have been adopted for the design of BoboCEP (Figure 6.3), as follows.

Receiver

This provides an entry point for data sources to input primitive events into BoboCEP. It is where sensor data can be introduced into a given software instance. It uses an

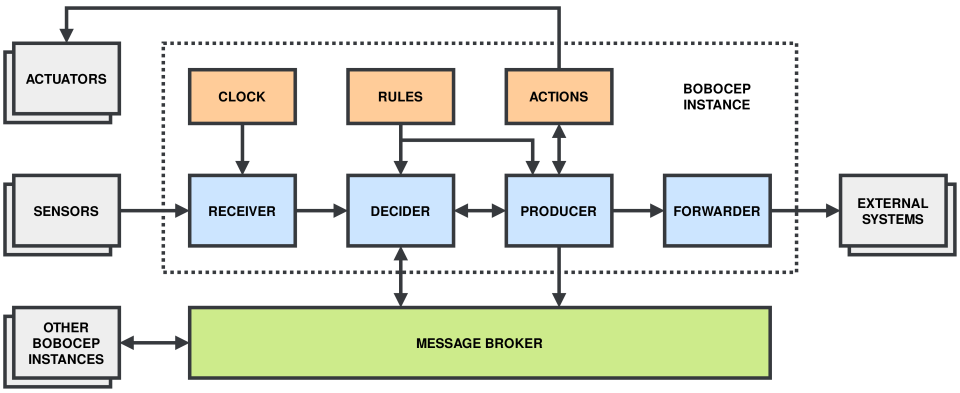

Figure 6.3: The BoboCEP architecture, with the key subsystems from the IFP architecture [59] in blue.

internal *Clock* to order events using timestamps to create a serialised event stream where events are ordered based upon when they entered the system.

Decider

This is the decision-making component of BoboCEP that determines whether composite events are generated or not (Figure 6.4). When a BoboCEP instance first starts, it generates a *handler* for each NFA to be implemented (i.e. the *Rules*). A handler is a container for an NFA and all instances of the NFA that are created at runtime (i.e. runs).

Primitive events from Receiver and events from Producer (below) are passed to the handlers, which might trigger a run instantiation if the current event under consideration fulfils the first predicate of the NFA, or causes state transitions if the event fulfils a predicate in an existing run. If a run reaches its accepting state S_A (Figure 6.6), it will trigger a composite event to be generated by Producer.

Buffer Each handler contains a buffer that efficiently links events together that are shared by all of the runs for a given NFA. It is an implementation of the *Shared Versioned Match Buffer* (SVMB) proposed by Agrawal et al. [4], which is designed to ensure that only one version of an event is stored, even if it is in use by multiple

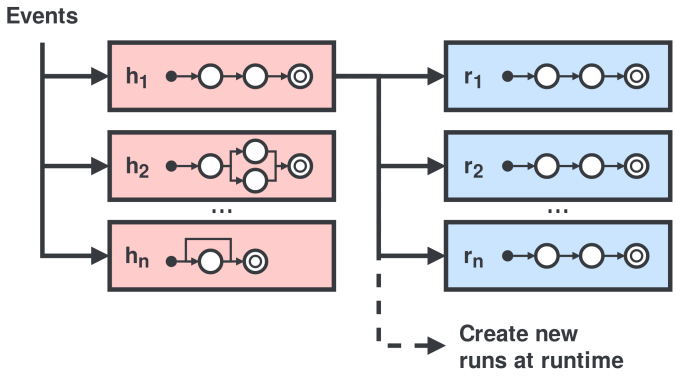

Figure 6.4: Handlers (red) and runs (blue) in Decider. Handler h_1 currently contains runs $r_1, ..., r_n$ and can create more runs at runtime. Solid arrows show events passing through handlers and runs, and the dashed arrow indicates a run `CLONE` operation.

runs, in order to provide a more memory-efficient event processing approach. It has been successfully implemented in existing CEP systems, such as FlinkCEP[1].

The process of SVMB is described with an example that follows on from the example in Section 6.1.1, and assumes a relaxed contiguity policy throughout. If the automaton from Figure 6.2 were a run r_1, and events $e_0, ..., e_3$ were added to a buffer, they would be added as shown in Figure 6.5. Events are categorised by the layer to which they are affiliated, and can be in multiple layers if they are being shared by different runs for different layers.

For each run is a unique *run version* number, and r_1 is allocated version `1.0` when e_0 is added to layer L_1. This version number is used until e_3 causes a state transition to L_3. At this point, run r_1 is then *cloned*, meaning that a new cloned run r_2 (Figure 6.5, orange) will transition to S_3 with version number `1.0.0`, and the original run r_1 remains in its current state S_2 and increments its version number to `1.1`. A clone occurs here because L_2 has a nondeterministic transition: it can either transition to L_3 or loop back to itself. Cloning enables the original run to potentially perform a different transition from its current state in the future.

Event e_4 was ignored (as occurred in the example from Section 6.1.1). Event $e_5^v = 1$ would cause a new run r_3 to be instantiated (Figure 6.5, blue) with version number `2.0`. Event $e_6^v = 4$ would cause run r_2 to reach S_A and finish; e_6 would be ignored

[1]https://ci.apache.org/projects/flink/flink-docs-release-1.7/api/java/org/apache/flink/cep/nfa/sharedbuffer/SharedBuffer.html

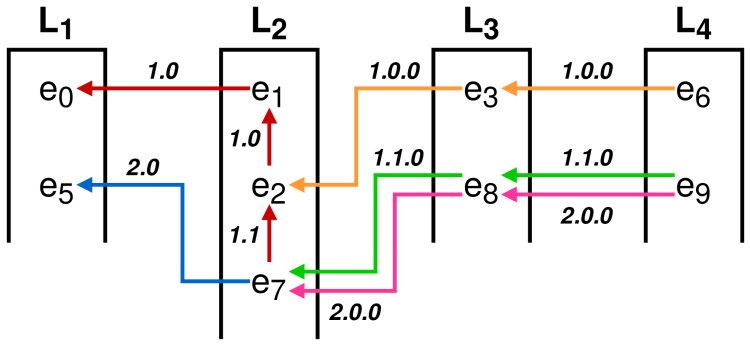

Figure 6.5: An example scenario of the SVMB, with run r_1 in red, r_2 in orange, r_3 in blue, r_4 in green, and r_5 in pink.

by r_1 and r_3. On event $e_7^v = 2$, run r_1 would loop back to S_2 with its incremented version number `1.1`, and r_3 would transition to S_2 with `2.0`. Event $e_8^v = 3$ would cause r_1 to clone run r_4 (Figure 6.5, green) with version number `1.1.0`. Run r_3 would also clone run r_5 (Figure 6.5, pink) with version number `2.0.0`. Event $e_9^v = 4$ would bring r_4 and r_5 to their respective S_A states.

Runs r_1 and r_3 technically cannot reach S_A because they always remain in S_2 and only their clones transition forward. They are only able to trigger more future clones, or halt to S_F via some kind of state-clearance approach e.g. a time window (Section 5.1). The implementation of SVMB in BoboCEP is described in Appendix A.1.

Producer

This is the subsystem that receives notification from the Decider that a complex phenomenon (i.e. an error detection) has occurred, and receives all of the events that were used to infer the existence of this phenomenon (i.e. the *pattern* of events). The pattern, the time of detection, and the NFA, are used to generate a composite event representing the detection of an error.

Producer then triggers the appropriate *Actions* to be executed, as determined by the Rules. These actions enable an assessment into the probable root cause of the detected error, and then enable the execution of an appropriate error recovery strategy based on the results of the assessment. The composite event is recursively sent back to Decider for potential use in the detection of future errors, as well as any actions that were executed during recovery (i.e. *action events*).

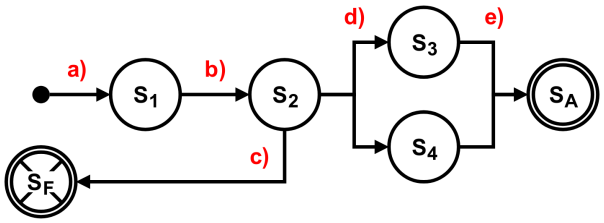

Figure 6.6: An example NFA with different transition types: (a,d) `CLONE`; (b) `TRANSITION`; (c) `HALT`; and (e) `FINAL`.

Forwarder

This enables BoboCEP to deliver the events generated by Producer to external systems (e.g. database systems) for the external consumption of the complex error events inferred by BoboCEP. This enables external systems to be notified on the occurrence of system errors, as well as whatever assessment and recovery attempts were made to handle the errors.

6.1.3 Message Broker

BoboCEP uses a message broker to replicate a state change in one instance to all other running instances. This is to ensure that all instances maintain the same internal state at all times. State changes are sent to a message queue on the broker, which are then broadcast to all other instances, in order to invoke state updates. The system messages exchanged are (Figure 6.6):

- `TRANSITION`. When a run is transitioning state but has only one next state (i.e. *deterministic*), the new state, and the event that triggered the transition, are broadcast to the other instances to force the transition on them also (b).
- `CLONE`. When a run is transitioning state but has multiple next states (i.e. *non-deterministic*), the run is cloned, meaning that the cloned run will transition and the original run remains in its current state (d). This message is also used when a new run is instantiated (a).
- `FINAL`. When a run reaches accepting state S_A, all other instances are signalled to complete their versions of the run, but not to execute any Action associated

with the composite event (e). The instance that initially reached S_A is the one that executes the Action in Producer.

- `HALT`. When a run reaches halt state S_F, a signal is sent to other instances to clear the run from their handlers (c). Whereas S_A leads to composite event generation, S_F does not.
- `ACTION`. When an Action is executed due to composite event generation, it notifies other instances as to whether it was successfully executed or not.
- `SYNC`. When a BoboCEP instance first starts, it synchronises with any other online instances that are already synchronised to retrieve the current state of FT support before starting error processing.

6.2 Vertical Farming Systems

It is predicted that the world population will reach 9 billion by 2050 and 70% will live in urban centres, which will strain the earth's resources along with climate change, specifically the food supply chain [43]. Both land clearing and more intensive use of existing croplands could contribute to the increased crop production needed to meet such demand, however ~25% of global greenhouse gas emissions result from land clearing, crop production, and fertilisation [194].

The problem outlined is the motivation for the *vertical farming system* (VFS) use case in this thesis. VFSs grow produce indoors, where environmental factors (e.g. light, water, nutrients) are tightly controlled to enable better sensing of farming and food processing operations [23, 200]. They are a growing trend in the IoT domain, and it is predicted that IoT device installations in the agriculture sector will increase from 30 million in 2015 to 75 million by 2020 [69].

The number and size of VFSs in Europe has seen rapid expansion in recent years, primarily driven by the drop in price of LED lighting technologies, a growing consumer demand for healthy local produce, and vacant office buildings that arose after the 2007-2008 global financial crisis [111]. Typical features of a VFS centre around saving energy, and include [24]: (1) a recycled water system augmented by rainwater

or water from a desalination plant; (2) automatic temperature and humidity control; (3) solar panel lighting and heating; and (4) tuneable 24-hour LED illumination.

The most efficient method of water consumption in VFS is facilitated via: *hydroponics*, where plants grow without soil by instead using mineral nutrient solutions in a water solvent; and *aeroponics*, where plant roots are hanging in plastic holders and foam material replaces soil [176, 118]. When used together, they can save up to 95% of water compared to traditional farming methods, and can help to eliminate farming waste water that is potentially hazardous to the environment and human health [101]. Touliatos et al. [195] grew lettuce in a VFS, where plants were grown in upright cylindrical columns, as well as a conventional horizontal hydroponic system. Result showed that the VFS produced 13.8 times more crops, calculated as the ratio of yield (kg FW) to the occupied growing floor area (m^2). This demonstrated that VFSs present a viable alternative to conventional horizontal growth systems by optimising growing space efficiency, thereby producing more crops per unit area.

Two VFS testbeds were created to evaluate the proposed FT framework. The first was a small-scale VFS with 4 plants and little component redundancy. This was later upgraded to a medium-scale VFS with 21 plants, more component redundancy, and a larger infrastructure, to demonstrate the scalability of the FT framework. Furthermore, a dataset from a real-world VFS was incorporated to validate the research in this thesis with an industrial VFS. These are discussed next.

6.2.1 Small-Scale Testbed

The motivation behind the testbed was that it would adopt the basis processes of a typical VFS. Namely, it would: (1) water plants when they needed to be watered, to ensure that produce does not become under- or over-watered; and (2) activate grow lights when natural light was not available, to ensure that photosynthesis was always available for the produce.

The small-scale VFS testbed had two shelves, each with two plants (Figure 6.7). Beneath each plant was a water container (d) that contained a USB-powered 44GPH 3.5V water pump that would pump water to its connected plant when activated.

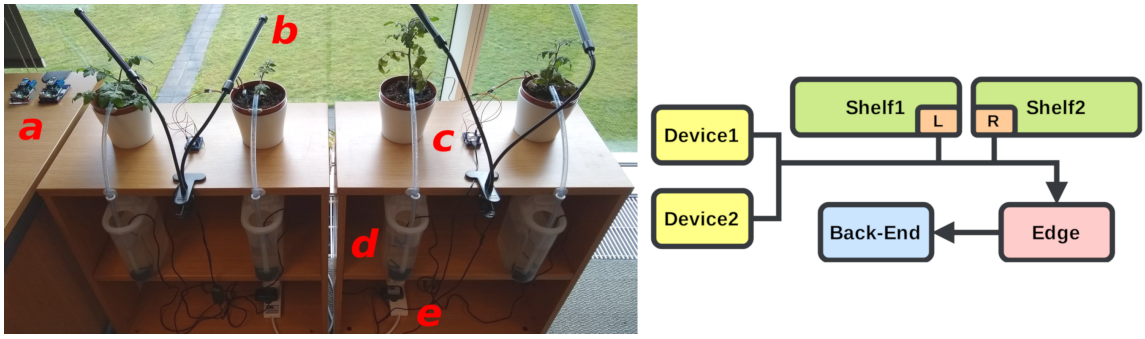

Figure 6.7: *Left*: The small-scale VFS testbed. *Right*: The testbed's infrastructure.

Above each plant were 5V grow lights (b) each with 40 LEDs that help to encourage photosynthesis, germination, and flowering. Both the water pumps and the grow lights were connected to KanKun KK-SP3 Wi-Fi smart plugs (e) that, when activated, would activate their attached actuators.

A Particle Photon Wi-Fi module containing a STM32 ARM Cortex M3 microcontroller (c) was on each shelf: microcontroller *L* for *Shelf1*, and *R* for *Shelf2*. Both microcontrollers had two moisture sensors, one for each plant on its shelf, as well as a *light-dependent resistor* (LDR) for each that measured the light intensity around the shelf. Two multi-sensors, *Device1* and *Device2* (a), as previously described in the scenario from Section 4.1.3, sent infrared-light data every 5 seconds to a Raspberry Pi 3 at the network edge. Device1 was the primary multi-sensor device. Device2 was in a hot standby state and would only activate when Device1 was identified as having failed. If a moisture value were < 0.5, its associated water pump would activate. If an infrared-light value were < 0.2, its associated grow lights would activate, or deactivate if ≥ 0.2.

The edge device ran a microservice that used *FlinkCEP* (v1.4.2)[2] as its NFA-based CEP system to provide real-time, reactive FT. Data received by the edge would finally be sent to a local database service on separate back-end machine one hop away from the edge. The back-end permanently stored all data generated by sensors in the system as well as events generated by the microservice on the edge device. The

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

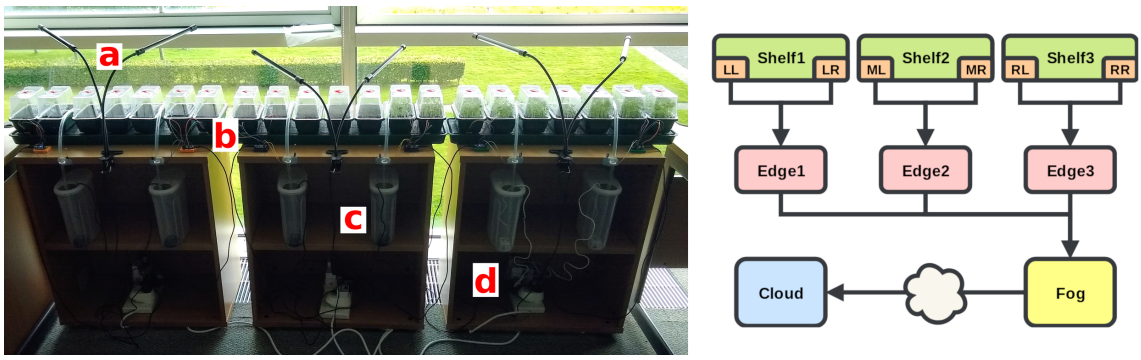

Figure 6.8: *Left*: The medium-scale VFS testbed. *Right*: The testbed's infrastructure and its default configuration, where Shelf*n*'s microcontrollers (orange) send data to Edge*n*.

back-end also ran a microservice that used *scikit-learn* (v0.2)[3] for ML tasks needed to provide long-term, proactive FT.

6.2.2 Medium-Scale Testbed

The purpose of expanding the small-scale testbed was to ensure that the research objectives outlined in this thesis (Section 1.3) could scale to a larger system that had more sensors, more data, and a larger infrastructure that exhibited distributed event processing and load balancing.

The medium-scale testbed had three shelves (Figure 6.8), each with two water containers (c) that pumped water to the reservoir of its *self-watering propagator* (SWP). Each SWP was a tray containing a reservoir on a platform, which was covered in capillary matting that would draw water from the reservoir into the soil of the plant pots resting on top of it. The SWPs enabled a redundant number of water pumps to water several plants at a time by instead pumping to the SWPs. There were 21 plant pots in total, 7 to each SWP. Therefore, two water pumps on a shelf were able to pump to its 7 plants simultaneously. The grow lights (a) and smart plugs (d) were configured as they were in the small-scale testbed.

There were 6 microcontrollers (b), with two on each shelf. Every 3 seconds they would sample: (1) temperature and humidity from two plants on their shelf; (2) water level from the reservoir; and (3) light intensity around the shelf using LDRs.

[3]https://scikit-learn.org/0.20

Table 6.1: A sample of five rows from the IGS dataset.

Name	Format	Value	Sensor	Timestamp
IGS-INV01-T0001-Sen37	Hum	60.450000	HVAC01	2019-07-05 06:49:27.543
IGS-INV01-T0001-Sen36	DegC	32.910000	HVAC01	2019-06-25 18:49:42.497
IGS-INV01-T0001-Sen36	Co2	530.000000	HVAC01	2019-08-02 11:00:36.250
BOT_TANK-CIRC_CURRENT	AMPS	2.365000	WATER01	2019-04-16 12:56:30.937
T0001-Feed	EC	1.236250	WATER01	2019-05-19 04:42:38.223

These payloads would be sent to one of three edge devices each running an instance of BoboCEP: *Edge1*, *Edge2*, and *Edge3*. The data would then be aggregated on a fog device further towards the cloud, which contained the ML models for the proactive FT solution and, like with the previous testbed, also used *scikit-learn* (v0.2) for ML tasks. All data would then be stored in the cloud on a relational database hosted by Amazon Web Services[4].

BoboCEP (v0.35)[5] was developed using the Python (v3.7)[6] programming language and was distributed across the three edge devices running on Raspberry Pi (v2 Model B) hardware. The BoboCEP Receivers (Section 6.1.2) consumed stream data via a Flask (v1)[7] server that enabled microcontrollers to send data ~3 seconds to the Receivers. Shelf1 contained microcontrollers LL, LR, which, by default, routed their data to Edge1; ML, MR to Edge2; and RL, RR to Edge3. On an edge device failure, microcontrollers sending data to the failed device would randomly pick another edge device through which to reroute their sensor data. RabbitMQ (v3.7)[8] was used for the BoboCEP message broker.

[4]https://aws.amazon.com
[5]https://pypi.org/project/bobocep
[6]https://www.python.org
[7]https://palletsprojects.com/p/flask
[8]https://www.rabbitmq.com

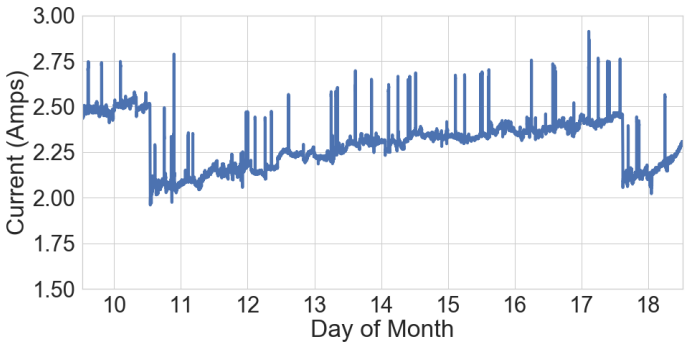

Figure 6.9: Current data over a 9-day period.

6.2.3 Real-World Dataset

In addition to the VFS testbeds, the company *Intelligent Growth Solutions* (IGS)[9] provided a dataset from their real-world VFS. Their solution was a farm containing four towers, each with 350m^2 of growing area per tower. Each growing tray had an integrated watering system that provided a constant water supply, with surplus water filtered and recycled. Each water pump pushed used water through a series of three filters. As filters became clogged with debris, the pump would use more current to maintain the same flow rate. Filters were cleaned and replaced regularly in response to high current rate.

IGS supplied a dataset containing all of the data that they use to ensure the correct functioning of their solution, a sample of which is shown in Table 6.1. Similarly to the testbeds, the data collected included the temperature (*DegC*) and humidity (*Hum*) of the growing environment, but also collected data regarding the system's emissions (*Co2*).

They also collected the water pump current (*BOT_TANK-CIRC_CURRENT*), which was what enabled IGS to determine when to clean the filters in the aforementioned scenario. In the current data (Figure 6.9), the current would gradually rise over several days, until IGS maintenance staff manually cleaned the filters, which would cause a sharp drop to the pump's optimally low current.

In FT terminology, the fault is environmental, caused by the gradual build-up of debris; the error is an unreasonably high current; and the failure would be the

[9]https://www.intelligentgrowthsolutions.com

inability to pump water caused by clogged filters, which might cause a water overflow or a build-up of bacteria. IGS provided this dataset to consider whether the reactive- and proactive-FT solution proposed in this thesis would be able to identify the error in real time, as well as predict when the error would probably occur next.

6.3 Summary

This chapter started with a justification for a new NFA-based CEP system that was able to provide a generic means of applying FT support to IoT systems that can be pushed as close to the fallible sensor network as possible and where FT support was, itself, fault tolerant. BoboCEP, an NFA-based CEP system, was developed to provide these qualities.

BoboCEP was designed to be distributed across the network edge on k software instances, where each instance maintained the current state of partially completed runs via active replication, so that $(k - 1)$ instances could fail without complete FT-support service loss. It used a SVMB to provide efficient event storage, where events were shared across all runs that used it. A third-party message broker was used alongside BoboCEP as the means by which message passing was performed between instances, in order to ensure state synchronisation.

The use case designed for the evaluation of this thesis consisted of two VFS testbeds. The first was a small-scale VFS with limited hardware/software redundancy and a minimal centralised infrastructure. The second was an upgrade to a medium-scale VFS, with more plants, more hardware/software redundancy, and more failure scenarios to detect. Finally, a dataset from a large-scale, industrial VFS was described for use in providing a demonstration that the FT-support solution is effective on real-world failure scenarios.

Chapter 7

Evaluation

In the previous chapters, an FT-support system based around microservices was proposed, where reactive-FT support provides error detection via NFA-based CEP, and proactive-FT support provides error prediction via ML. This chapter presents an evaluation of the FT framework proposed in this thesis using a series of experiments that test the research objectives from Section 1.3. Firstly, CPoF experiments validate whether it is possible to provide effective reactive-FT support via NFA-based error checking. This is then built upon by exploring the various problems that can be solved when proactive FT is also included as part of the FT-support process. Finally, the benefits of BoboCEP specifically are explored, namely, resilient long-term event processing and load balancing, as well as a performance evaluation of the BoboCEP Python implementation.

7.1 Methodology

7.1.1 Techniques

To determine the efficacy of the proposed FT framework, its design and implementation requires evaluation. With software systems, an evaluation can be characterised as a goal-oriented task where actions are performed that result in one or more reported outcomes, which help to improve the quality of the actions or to choose the best action alternative [84]. Some evaluation approaches conducted after system implementation evaluate the general worth of the information system, whereas multiple-criteria evaluation approaches, which include subjective and objective eval-

uations, consider both user and system constraints equally [28]. This section briefly outlines the two common techniques used to evaluate software systems, namely [84]:

- **Descriptive evaluation techniques**. These are used to describe the status and the actual problems of the software in an objective, reliable and valid way. These techniques are user based and can be subdivided into several approaches:
  - *Behaviour based*, where the user's behaviour is recorded while working with a system which produces some kind of data. These procedures include observational techniques and *thinking-aloud* protocols.
  - *Opinion based*, where the aim is to elicit the user's (subjective) opinions e.g. interviews, surveys and questionnaires.
  - *Usability testing*, which is a combination of behaviour- and opinion-based measures with some amount of experimental control, usually chosen by an expert.
- **Predictive evaluation techniques**. These are used to make recommendations for future software development and to prevent usability errors. Its procedures include walkthroughs and inspection techniques. An important distinction between predictive and descriptive techniques is that predictive relies on data by experts who simulate real users, which is hard to apply in a descriptive setting because of objectivity and reliability requirements.

7.1.2 Justification for Evaluations

Given the various techniques for evaluating software systems, appropriate evaluation techniques must be decided upon to help provide some level of measurement to assess whether the key objectives of this thesis have been fulfilled. As such, it is important to reiterate the research objectives from Section 1.3, namely:

1. To identify and classify faults events and fault patterns in IoT systems.
2. To develop a service-oriented fault-tolerance framework for IoT systems that combines both reactive- and proactive-FT support.

3. To incorporate strategies and mechanisms that facilitate effective fault mitigation.

4. To ensure that the framework can scale to more complex IoT scenarios.

The primary motivation for these objectives was to determine whether it was possible to provide pluggable, interoperable FT support for IoT systems that was able to reactively handle errors and then learn how to proactively handle them with enough experience of what errors the system encounters in its lifetime.

Objective 1 was intended to address the lack of consistency with FT frameworks in IoT systems. As shown in the literature review in Section 3.1, the proposed solutions did not have a consistent understanding of how systems could fail and, therefore, led to a large number of bespoke systems with FT support that was not cross-compatible with each IoT system. Objective 1 attempts to address this issue with the combination of the VFF framework and CPoF, to first categorise vulnerabilities, faults, and failures, and then apply them via reusable NFAs (Section 5.2.2). Experiments to evaluate Objective 1 (Section 7.2) consider how the error-checking NFAs can provide reactive FT *without* proactive FT included, to measure the efficacy of reactive-FT support in isolation.

Objective 2 was intended to build upon the findings of Objective 1 by incorporating NFAs into a reactive- and proactive-FT support framework that enabled NFAs to be applied via CEP systems and for predictive analytics to learn from and anticipate errors detected by the CEP system. Experiments for Objective 2 (Section 7.3) consider how correlations between with system context (e.g. time), and correlations with other recent system errors, can help to predict and anticipate future imminent errors. These experiments provide assurance that the two complementary inference-based mechanisms of CEP and ML can provide the desired intelligent FT framework.

Objective 3 was intended to build on Objective 2 by considering how error-recovery mechanisms fit into the FT framework. The experiments for Objective 2 demonstrate effective reactive and proactive error detection, but Objective 3 warrants further emphasis on recovery and how all of the FT stages (Section 2.2) are applicable to the

FT framework. Experiments for Objective 3 (7.4) are similar to those in Objective 2 but with a focus on the formal definition of FT scenarios as functional requirements. A core part of these experiments is to ensure that error detection will occur despite data errors, and that error recovery has been correct given data correlations after recovery had been executed.

Objective 4 was intended to build on Objectives 2 & 3 by considering whether the FT framework has the ability to scale to larger and more complex IoT infrastructures and failure scenarios. The experiments for Objective 3 use the small-scale VFS testbed, and the experiments for Objective 4 use both the medium-scale VFS testbed as well as the dataset from a real-world, large-scale VFS. Furthermore, experiments specifically for BoboCEP are performed (Section 7.5) to consider how it is able to provide long-term event processing and load balancing, which are both key features for CEP-enabled reactive FT to ensure correct support at scale. A performance analysis of the BoboCEP Python implementation is provided to determine how well it copes under heavy loads.

The evaluation experiments help to ensure that the *key qualities* of the proposed FT framework (Section 1.4) are present, namely: (1) *interoperability*: heterogeneous sensor data from a variety of bespoke sensors are used with both FlinkCEP and BoboCEP from the small- and medium-scale testbeds, respectively; (2) *reusability*: the error-detection NFAs (Section 5.2) are reused to detect and provide recovery for a variety of errors; (3) *dependability*: active replication of partially completed complex events via BoboCEP provides a resilient reactive-FT platform; (4) *scalability*: experiments were conducted on the small-scale and medium-scale VFS, and the large-scale, real-world VFS via the provided dataset (Section 6.2).

7.1.3 Predictive Models

Numerous predictive models were used in the experiments of the evaluation, in order to assess the proactive-FT elements of the proposed FT framework. Three classification and three regression algorithms were used during the evaluation, where each scenario would use either a classification or regression model, depending on which

suited the scenario the most. The models used a randomised 75%/25% data split for training and testing. The input to each of the models was a dataset generated using experience from the Real-Time FT microservice(s) providing CEP (Section 4.1.1) Example data included contextual system data from sensors and composite error events generated by the CEP engine itself.

All sensor data had a numerical format which required no preprocessing for the models to interpret the data. However, some models scaled data to the range $[0, 1]$, if necessary. The Predictive FT microservice (Section 4.1.1), which provided the proactive-FT aspect of the FT framework, did not activate a predictive model until at least 15 training instances had been collected for it to train with. This ensured that each model had a minimum amount of data to make reasonable predictions. A summary of all the models' results are shown in Tables 7.1 & 7.4. The algorithms used for the models are discussed next.

Classification

The system context surrounding data-centric error detection is an important consideration to ensure that an error will be detected, and to identify whether erroneous data (e.g. caused by sensor malfunctions, human tampering) will lead to an incorrect error detection being triggered. Classification techniques can be leveraged to predict whether an action is likely to be triggered imminently, given the current data trend. If data is erroneous, it may *not* cause the system to trigger an action when it should, and may put the system into an erroneous state. The following three classification algorithms were used in the evaluation:

- **K-Nearest Neighbor (KNN)**. An instance-based learning approach that uses a simple non-parametric procedure to assign a class label to the input pattern based on the class labels represented by the K-closest neighbours of the vector. KNNs are known for being computationally intensive since they rely on searching neighbours among large sets of n-dimensional vectors for each prediction [104, 82]. In the evaluation, the models used 10 nearest neighbours for classifying new instances.

- **Support-Vector Machine (SVM)**. A linear classifier that attempts to construct a maximum-margin hyperplane to optimally separate data into two categories in order to provide the best generalisation capacity, and are suitable for binary classification problems [182]. In the evaluation, its C hyperparameter was set to 1000.0 and was trained with 1000 iterations. Its features were normalised before training because, with SVMs, feature vector normalisation had been shown to lead to superior generalisation performance [57].
- **Random Forest Classifier (RFC)**. An ensemble learning technique that is a hybrid of the *bagging* and *random subspace method* approaches that uses decision trees as the base classifier [174]. RFC was proposed by Breiman [31] in 2001, who defined it as: a classifier consisting of a collection of tree-structured classifiers $\{h(\vec{x}, \Theta_k), k = 1, ...\}$ where $\{\Theta_k\}$ are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input $\vec{x}$. In the evaluation, the models had 10 trees/estimators and a maximum tree depth of 2.

These algorithms were selected because they have distinct properties, namely: (1) KNN is an instance-based learning technique, whereas SVM and RFC are both model based; (2) RFC uses ensemble learning, where multiple learning machines are trained and their outputs combined [34], whereas KNN and SVM do not; and (3) KNN uses *lazy learning*, where there is no explicit training stage and classification only occurs when queries are made [206], making the model much faster than SVM and RFC which adopt an *eager learning* approach.

Regression

In previous literature, regression models were considered for masking errors when data was temporarily unavailable i.e. data blackouts. For example, Choubey et al. [51] used an ANN to predict temperature using correlations in other sensor data when temperature sensors were not available. Fekade et al. [72] demonstrated that sensor clustering could recover massive amounts of missing sensor data values.

However, there is also the situation of data being *available but erroneous*, which

might cause erroneous system behaviour if the data were used for decision making. Regression techniques enable erroneous data to be masked by predicting more reasonable data to be used instead. The following three regression algorithms were used in the evaluation, and were selected because they have similar properties to the chosen classification algorithms:

- **Ada Boost Regressor (ABR)**. An ensemble-learning approach that trains models sequentially. The first machine is trained with examples picked with replacement from the dataset, and the samples from the models with the largest prediction errors are more likely to be picked as members of the training set of the next machine. By repeating this process, patterns that are difficult are more likely to appear in the training sets [66]. ABR combines the weak hypotheses of each machine by summing their probabilistic predictions and, in experiments, have been shown to improve the performance of ANNs when summing the outcomes of the networks and then selecting the best prediction [67, 77]. In the evaluation, the maximum number of estimators used was 50, which would stop early if a perfect fit were achieved before then.

- **Support-Vector Regressor (SVR)**. Similar to its SVM counterpart, it is characterised by the use of kernels, sparse solution, and VC control of the margin and the number of support vectors. Although it is less popular than SVMs, SVR have been shown to be effective in real-value function estimation [18]. In the evaluation, its C hyperparameter was set to 1.0.

- **Random Forest Regressor (RFR)**. Similar to its RFC counterpart, it grows its trees depending on a random vector Θ, such that the tree predictor $h(\vec{x}, \Theta)$ takes on numerical values, as opposed to class labels, and produces a numerical output [31]. In the evaluation, the models used 10 trees/estimators with no maximum tree depth.

7.2 Experiments: Reactive FT

In these experiments, the concept of CPoF (Chapter 5) was evaluated in isolation, i.e. without proactive FT included, with an assessment of how the error-checking NFAs can be applied to failure scenarios in the small-scale VFS (Section 6.2.1). Specifically, how the VFS attempted to detect and recover from the following two scenarios: (1) when attempting to water a plant with no water left to pump; and (2) when Device1 suffered a 'performance drop' i.e. reduced data-transmission rate.

The purpose of these experiments was to consider: (1) the applicability of the scenario with regard to the VFF framework (Section 4.2); (2) the error(s) that might have propagated if the underlying fault were activated; and (3) the error-detection checks that would be needed to detect and recover from it.

7.2.1 Scenario: Empty Water Container

In this experiment, a service failure was identified whereby a plant was unable to be watered because its water tank was empty. Using VFF, this was classified as: an *environment* vulnerability, exploited by an *interaction* fault, that might have led to a *state transition* response failure. This was because, if the soil were not watered, the subsequent moisture data would not change due to a lack of state change in the physical world.

Firstly, an NFA was needed to identify a trend in moisture data. When soil was watered, the slope between the latest two moisture values should have become very large for a short period before stabilising. For this, the trend check NFA (Figure 5.2b) was used to first consume an initial moisture data event e_1. Then, subsequent moisture events were checked within a time window of 30 seconds. If any second event e_m in this time produced a slope ≥ 0.05, an error-detection event d_t was produced. The slope was calculated as $(e_m^v - e_1^v)/2$.

Another NFA was implemented that checked if there was *not* a trend check within 30 seconds of a water pump action occurring. This used the correlation check NFA (Figure 5.4), as follows. Event e_1 was fulfilled when a water-pump action was suc-

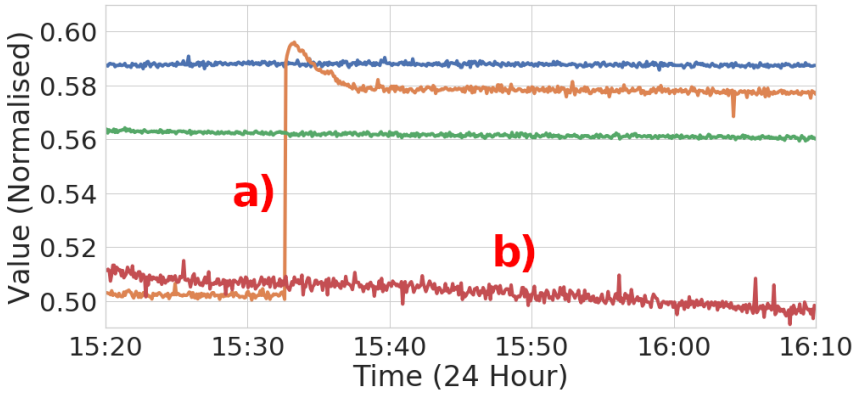

Figure 7.1: Moisture data from the four moisture sensors of the small-scale testbed, represented by different colours.

cessfully executed. If 3 subsequent moisture events were received, with no d_t event in that time, it was assumed that no trend would occur. The run halted if d_t occurred.

In this experiment, the water container connected to the rightmost plant was empty. Its moisture data dropped to value 0.49 at ~15:49 (Figure 7.1b). This surpassed the 0.5 threshold and caused a water-pump action. However, it failed to pump any water, as reflected in the lack of trend change in the data. After 3 additional moisture values were received without a trend check event occurring, the reversal check NFA was fulfilled, indicating no trend after the water-pump action. The recovery strategy was to send an alert message to prompt human maintenance to resolve the issue. For comparison, the data from the centre-right moisture sensor dropped below value 0.5 at ~15:32 (Figure 7.1a), which caused a large trend increase shortly thereafter.

7.2.2 Scenario: Data Transmission Degradation

In this experiment, a service failure was identified whereby the data transmission rate from Device1 started to decrease i.e. performance degradation. Using VFF, this was classified as: a *network* vulnerability, exploited by an *interaction* fault, that might have led to a *timing* failure.

Firstly, an NFA to identify when a drop in performance had occurred was needed. The performance check NFA from Figure 5.3a provided this, where e_1 represented an infrared-light data event from Device1. When the next consecutive infrared-light event was received, e_2, the difference between timestamps was checked to see if the

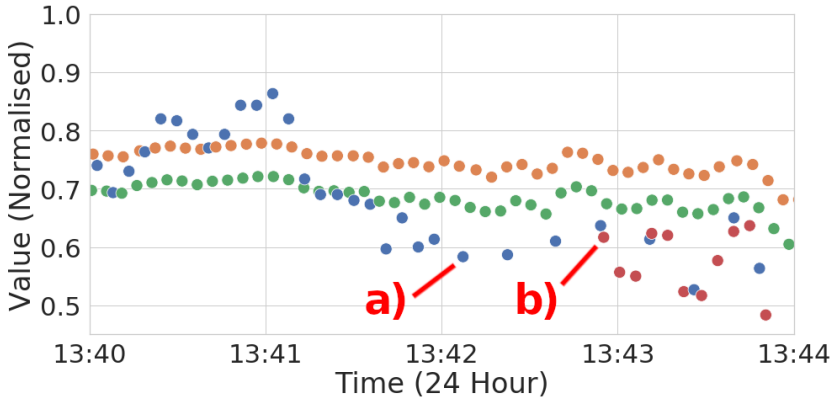

Figure 7.2: LDR data from the two microcontrollers of the small-scale testbed (orange, green), and infrared data from Device1 (blue) and Device2 (red).

time difference was > 10 seconds; this threshold was chosen because it was double the 5 second data rate of Device1. If the difference between events were ≤ 10, the run halted. This NFA produced error-detection event d_e.

No recovery was provided for one performance drop because, in isolation, it was probably a transient error with no clear root cause. Instead, a second NFA was defined to identify intermittent performance drops using the persistence-check NFA (Figure 5.3b) where, if d_e occurred 3 times within 60 seconds, then the persistence of these errors led to new error events d_p, r_p. The recovery was to activated and switch over to Device2, to return the overall rate of infrared-light data to ≤ 5 seconds, as shown in the infrared data that became intermittent at 13:42:10 (Figure 7.2a).

Each of Device1's data events took ~10/15 seconds to arrive. The wide intervals between these events caused d_e errors after each one. When this occurred 3 times in 60 seconds, an intermittent performance error was detected and caused d_p, r_p events, which led to the introduction of Device2 data (Figure 7.2b).

7.3 Experiments: Proactive FT via Correlations

In these experiments, the concept of reactive FT via CPoF was combined with proactive FT via ML models, implemented using the small-scale VFS as before. The scenario was that of battery depletion on Device1, and proactive FT attempted to predict when the depletion was expected to occur via two approaches: (1) correlating

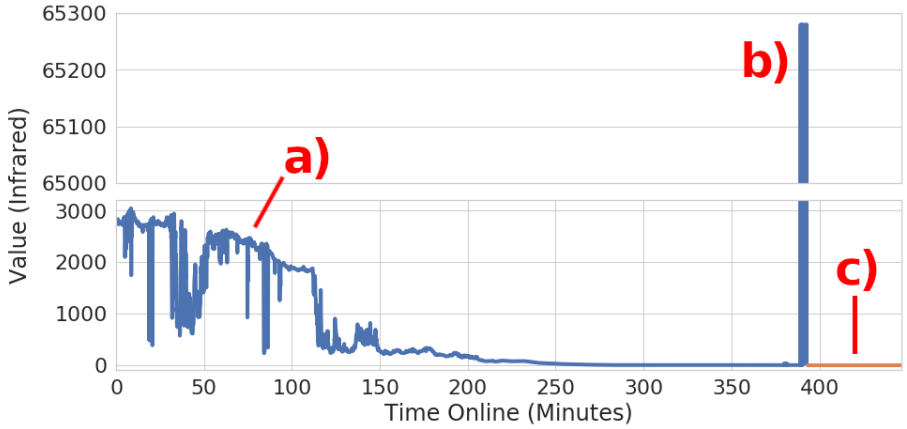

Figure 7.3: Infrared light data from Device1 (blue) and Device2 (orange) during the lifetime of Device1.

Device1's duration online with data loss errors; and (2) correlating erroneous data spikes in Device1's data with data loss errors. This enabled contextual data (i.e. time) and composite events (data spike error-detection event) to provide a means of proactively detecting and resolving the same fault.

7.3.1 Scenario: Battery Depletion

Low battery voltage has been identified as a prevalent cause of data spikes in IoT sensors [42]; a phenomenon that Device1 exhibited in Section 4.1.3. To handle this, a *data spike* error was defined using the trend-check NFA (Figure 5.2b). As Device1's data was sent every 5 seconds, the last 12 infrared-light events from Device1 were used to get roughly the last minute's worth of data from the device. The first 11 events $e_1, ...e_{11}$ 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 NFA (Figure 5.3a). It first checked for event e_1 from Device1, followed by a null event $e_2 = e^{\emptyset}$ (Section 5.2.2) that occurred 15 seconds after e_1. If another Device1 payload 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 Device1. For every 15 seconds that elapsed without Device1

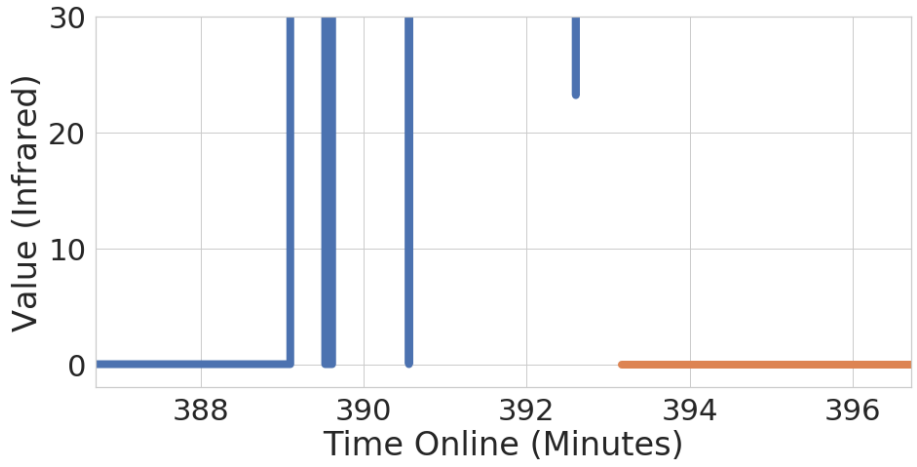

Figure 7.4: Infrared light data from Device1 (blue) and Device2 (orange), with a close-up of battery depletion and switchover from Device1 to Device2.

data, another d_e, r_e was generated. If 3 unsuccessful r_e were generated without any successful pings or Device1 data within 60 seconds, a *persistent data loss* error d_p was generated using a persistence check (Figure 5.3b), and recovery r_p would cause a switchover to Device2.

Error correlations were checked for via the correlation-check NFA (Figure 5.4) in addition to the checks above. Specifically, it 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, a second correlation check was defined to check when d_p did *not* follow d_t within 10 minutes (i.e. $e_2 = e^{\emptyset}$), which produced event d'_r.

Demonstration Device1 ran on full charge until depletion, and the system monitored the infrared-light values generated by it. Throughout the day, the values fluctuated (Figure 7.3a) which produced several data-spike d_t errors. Minutes before battery depletion, the values spiked several times to 65279 (Figure 7.3b).

It also produced three unsuccessful data-loss recovery r_e events, which led to a persistent data-loss error d_p, causing a switchover to Device2 (Figures 7.3c & 7.4). The wait for d_p meant that error recovery took ~**45 seconds** to complete.

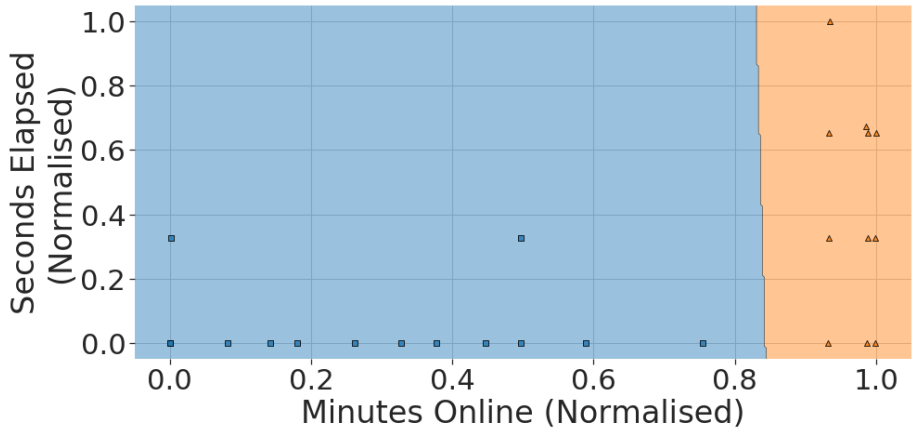

Figure 7.5: Time correlation: linear SVM model to predict persistent (orange) and non-persistent (blue) data loss.

7.3.2 Time Correlation

The first proactive-FT approach to predicting battery depletion correlated data-loss events with Device1's duration online. The intuition was that data-loss events occurring after Device1 that crashed due to battery depletion were going to be persistent. Thus, the system could avoid a persistent data-loss error by preempting the switchover instead.

By repeating the previous demonstration on the small-scale VFS, a dataset was built containing 25 examples of instances when persistent data loss occurred that were either caused by Device1 crashing or due to some other cause (e.g. network congestion). This resulted in 11 examples for persistence likely caused by a Device1 crash, and 14 examples for unlikely persistence. The minutes for which Device1 was online and the elapsed seconds that passed since the last Device1 payload (as determined by data-loss error events) were chosen as two features for this model.

After training the KNN, SVM, and RFC classification models, all models achieved a perfect accuracy of 1.0 because the data points were highly linearly separable. SVM was chosen as the final model because it provided a smooth boundary between both classes (Figure 7.5). RFC and KNN created jagged boundaries that did not reflect the trend well, which might have led to misclassifications in predictions therefore.

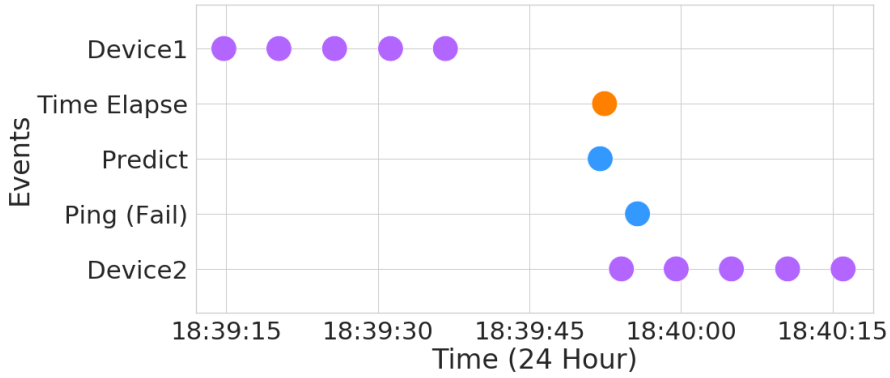

Figure 7.6: Time correlation: data loss error correctly identified as persistent.

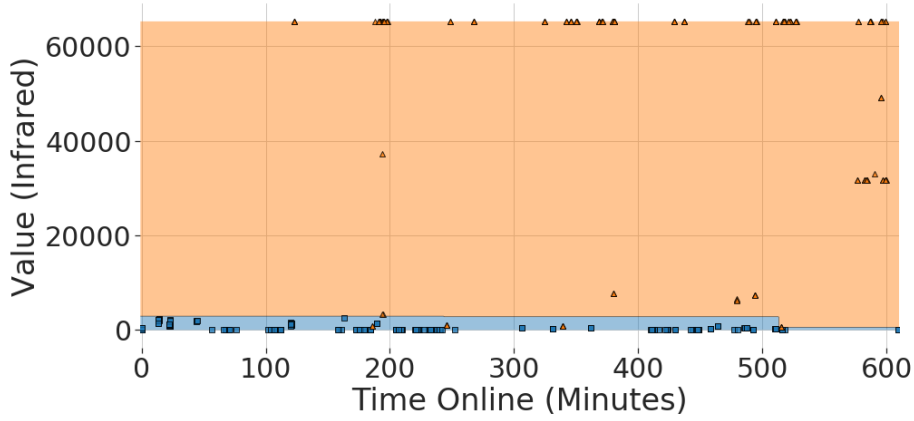

Figure 7.7: Error correlation: RF model to predict persistent (orange) and non-persistent (blue) data loss.

Demonstration Device1 was activated on a full charge with the SVM model running. The device had full battery depletion at ~18:29:37 (Figure 7.6) and had been online for ~495 minutes (~0.9 when normalised in Figure 7.5). After the elapsed period of data loss from Device1, an error occurred 15 seconds after the last infrared-light value from Device1. The model predicted that this error would persist and preemptively switched over to Device2.

This solution provided preemptive error detection that was **~30 seconds faster** than using the previous reactive-FT solution only. However, it was only effective at predicting failure when Device1 was first activated on a full charge. Next, a model was considered that identified the same failure based on error correlations.

7.3.3 Error Correlation

The second proactive-FT approach for predicting battery depletion correlated data-spike events with data-loss events. It exploited data spikes caused by low battery voltage (Figure 7.3b) by detecting them using error-correlation event d_r and executing preemptive switchover to Device2 before battery depletion could occur.

To build the model's dataset, Device1 was activated and left to run until depletion 60 times, with varying levels of initial battery charge, in order to build a dataset of 473 total data spikes. This resulted in 187 examples indicating malfunction that led to an imminent crash, and 286 *not* caused by malfunction i.e. caused by natural light fluctuations instead.

After training models, the RFC model was identified as the superior model, despite having identical accuracy and F1 score to KNN (Figure 7.7). However, KNN showed signs of overfitting whereby examples of persistent data loss (orange) were within the non-persistent data loss boundary (blue), whereas the RFC model provided a clear boundary between both classes.

During data collection, an interesting phenomenon occurred. 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 ~510 minutes online (Figure 7.7) because it caused the model to predict malfunction for spikes in the normal range if the device had been online for a long time.

Demonstration The data from the reactive-FT experiment (Figure 7.3) was inserted 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 7.3a) were correctly predicted as not being caused by malfunction, because they did not pass the decision boundary at ~3500 (Figure 7.7). However, on the first spike to 65279 (Figure 7.3b), the model predicted an imminent malfunction, and triggered a switchover to Device2. This introduced Device2's data before Device1 had fully depleted (Figure 7.8).

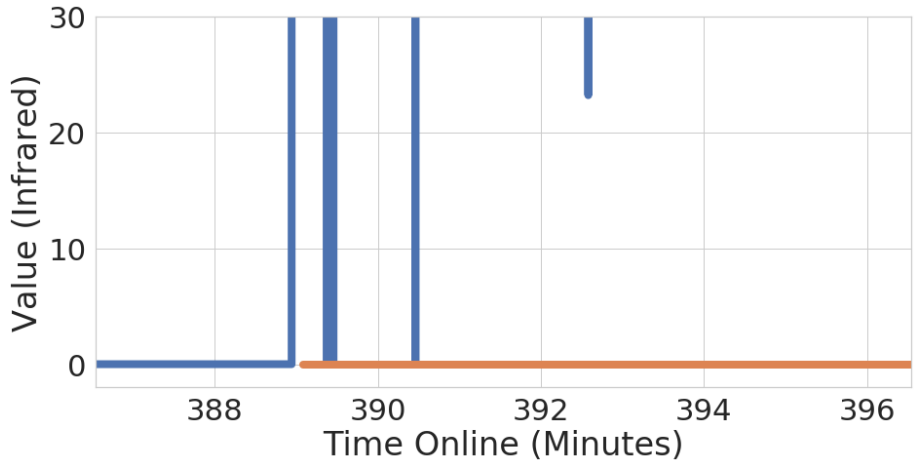

Figure 7.8: Error correlation: a close-up of the switchover to Device2 (orange) before battery depletion had occurred on Device1 (blue).

Table 7.1: Models trained in the experiments from Section 7.3, with the selected models in bold.

Correlation	**Model**	**Accuracy**	**F1-Score**
Time	**SVM**	**1.000**	**1.000**
	RFC	1.000	1.000
	KNN	1.000	1.000
Error	SVM	0.916	0.840
	RFC	**0.979**	**0.964**
	KNN	0.979	0.964

This solution provided preemptive migration that was ~**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 when a prior data spike error occurred, which was not always guaranteed to happen near battery depletion.

7.4 Experiments: FT Requirements

In the last section, reactive and proactive FT were shown to be able to handle the common scenario of IoT device battery depletion using two different approaches. The successful proactive error detection was possible because the expected erroneous data trends occurred as expected i.e. as defined in the NFAs used to detect it.

However, there is also the situation of data being *available but erroneous*, which might cause erroneous system behaviour if used for FT decision making e.g. being

unable to detect specific errors, or "detecting" errors that do not actually exist. In this section, a formal approach to defining reactive and proactive FT in terms of functional requirements is proposed, where the expected trends that occur before detection and after recovery are precisely defined.

7.4.1 Detect, Assess, Recover, Analyse

To provide correct FT support, the system needs to determine what error is being detected, what are the necessary preconditions that need to occur to ensure correct error detection, and also what the expected consequences are of error recovery on the system. This is similar to the *design-by-contract* (DbC) methodology, where software is annotated to formally specify behaviour used for the formal verification of correctness which typically includes pre- and postconditions [193], but is instead designed for inferential FT support.

DARA, a novel *Detect, Assess, Recover, Analyse* framework, is proposed to ensure long-term FT support in IoT systems. DARA achieves this by combining CEP and ML to provide a data-centric approach to FT that is able to learn from and anticipate errors that could hamper the system's ability to provide effective FT support. DARA applies the VFF framework (Section 4.2) to provide a systematic, generic approach to defining and implementing FT support, whereby erroneous conditions that could affect error detection, and evidence that suggests ineffective error recovery, can be formally defined and properly mitigated (Figure 7.9). The four stages of the DARA process are described next.

Stages

Detect To provide 'generic' FT support that can apply to most IoT systems, error definition and detection need to be expressed in a modular, language-agnostic manner, so that problems common to most systems have common solutions that can be reused in different contexts. The NFAs from Section 5.2 provide the desired generic error-definition approach that can be implemented into any NFA-based CEP system to provide a data-centric approach to error detection via real-time stream data analysis. This stage adopts the CPoF approach to error definition and detection.

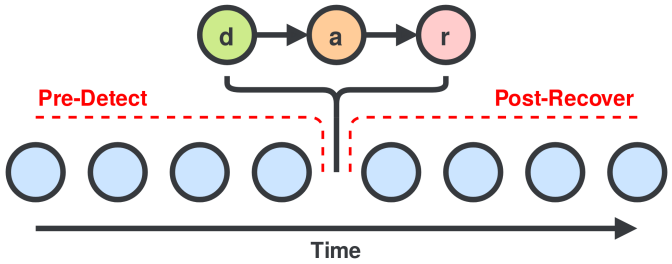

Figure 7.9: The DARA process. System data are blue circles, along with error detection d, assessment a, and recovery r events.

Assess This stage reasons on the possible faults and vulnerabilities that might have caused an error, in order to execute the best recovery strategy to mitigate it. The RcB approach (Section 2.2.1) is proposed for applying assessment, in order to discover the fault that is the root cause of an error, if possible. RcB is a passive replication strategy whereby a series of tests are performed in series, and the first test to pass is the accepted test [219], making it highly applicable for DARA's assessment stage. RcB can attempt several assessments $a_1, ..., a_n$ in series, where the success of an assessment suggests some particular fault and vulnerability were the cause of the detected error under assessment. If all are unsuccessful, a default assessment a_0 is made to represent an unknown root cause.

Recover This stage encompasses many similar notions defined in FT literature: error recovery, error mitigation, and fault treatment [122, 91]. Each assessment has a series of recovery strategies to attempt, which are executed using the same RcB approach used for the Assess stage. When an assessment is chosen, its corresponding recovery strategies $a^r = r_1, ..., r_n$ are chosen for execution. For each unsuccessful recovery r, the next is attempted until success. If all strategies are unsuccessful, or no assessment succeeded, a default recovery event r_0 is produced, representing that the error was unable to be automatically recovered from.

Analyse This stage is a holistic strategy that uses long-term data analysis to ensure that the system is in a correct state via data analysis and error prediction, so that it is capable of detecting errors. DARA treats failures as requirements of the system, alluding to the notion of *failure semantics* i.e. allowable service behaviours

[73]. It uses ML to learn from system data trends, error trends, and system context, to predict whether erroneous data will lead to errors going undetected.

It attempts to proactively predict two erroneous scenarios (Figure 7.9), namely: (1) in the *Pre-Detect* period, whether there any indications that FT support might fail to detect an error; and (2) in the *Post-Recover* period, whether there any indications that recovery had actually failed and therefore might cause error propagation due to the original error remaining unresolved in the system.

7.4.2 Requirements

DARA was applied to two failure scenarios that could occur in the medium-scale VFS (Section 6.2.2), and a third scenario from the real-world VFS dataset (Section 6.2.3). The scenarios were as follows:

- **W1**. In the medium-scale VFS, water was pumped to the water reservoirs that uniformly passed water to each of their 7 plant pots. Two water level sensors were placed within each reservoir. This scenario considered what should happen if a reservoir dried up and needed replenishing with more water from the water pumps.
- **L1**. In the medium-scale VFS, plants relied on natural light when it was available, and switched to the grow lights for photosynthesis when it was dark. This scenario considered what should happen when natural light fluctuations occurred and how the system could detect the optimal time to switch to artificial light, so that the plants always had optimal growing conditions.
- **P1**. In the real-world VFS, water filters ensured that when water was being recycled, it was always clean and without debris. Using the supplied dataset, this scenario considered when the optimal time to clean the filters should have been, so that the water pump's current did not become excessively large when filters were clogged and the pump was forced to work harder and its current increased as a consequence.

Each of these scenarios are expressed with three requirements (Tables 7.2 & 7.3).

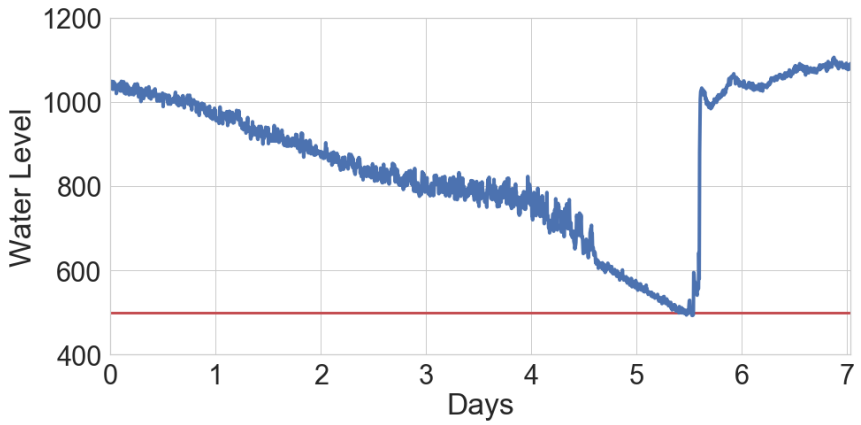

Figure 7.10: W1.DAR: water level data from a reservoir over a 7-day period, with a water-pump activation threshold at 500 (red line).

A **DAR** requirement, which encompasses the Detect, Assess, and Recover stages of DARA that ensures reactive error detection and recovery. A **PRE** requirement, which is one half of the Analyse stage of DARA that ensures correct detection of its corresponding DAR requirement. A **POST** requirement, which is the other half of the Analyse stage of DARA that ensures correct recovery of its corresponding DAR requirement. The three scenarios are explored next.

7.4.3 W1: Reservoir Water Depletion

W1.DAR

This requirement was defined to prevent damage to produce caused by water depletion in reservoirs. When water was low, the pumps activated to replenish the reservoirs. The limit-check NFA (Figure 5.2a) was used to check whether an event's value was within some unreasonable threshold i.e. $0 \leq e \leq 500$. Figure 7.10 shows water level data representative of the type of data produced from the medium-scale VFS (Section 6.2.2). On detection of a low water level after ~5.4 days, the default assessment inferred that this was caused by water depletion.

Recovery was a hardware redundancy approach, enabled by the fact that each reservoir had two connected pumps and water containers (Section 6.2.2): if the CEP system failed to activate one pump, it would try to activate the other. If both failed, the system would notify system admins to perform a manual fix, however, this was a

Table 7.2: Failures and their error events to detect, assess, and recover from.

Failure		**Detect**		**Assess**				**Recovery**
Requirement	*Class*	*Check*	*Description*	*Action*	*Fault*	*Vulnerability*	*Verdict*	*Action*
W1.DAR A reservoir shall replenish its water supply if its water level is less than 500.	Response (Value)	Limit	Water level < 500.	**0.** None	Interaction	Environment	Water depleted.	**1.** Activate primary pump. **2.** Activate backup pump. **0.** Notify fault observer.
L1.DAR The grow lights shall activate if the light intensity in the room is less than 500.	Response (Value)	Limit	Light intensity < 500.	**0.** None	Interaction	Environment	Room is dark.	**1.** Activate grow lights. **0.** Notify fault observer.
L2.DAR The grow light shall deactivate if it has been activate for at least 2 hours.	Response (State)	Performance	Two hours since last L1.DAR event.	**0.** None	Physical	Policy	Overheat period.	**1.** Deactivate grow lights. **0.** Notify fault observer.
P1.DAR System admins shall be notified when current slope increase is ≥ 0.5 over 1 day since last filter clean.	Response (Value)	Trend	Increase ≥ 0.5 over 1 day.	**0.** None	Interaction	Environment	Clogged filters.	**0.** Notify fault observer.

Table 7.3: Pre-Detect and Post-Recover analyses of the failures in Table 7.2.

Pre-Detect			**Post-Recover**		
Requirement	*Approach*	*Action*	*Requirement*	*Approach*	*Action*
W1.PRE The water level shall decline with a slope $\geq$ 500 within 5 days of the last water pump activation.	Classification	**1.** Activate primary pump. **2.** Activate backup pump. **0.** Notify fault observer.	**W1.POST** The water level shall increase with a slope $\geq$ 100 within 1 minute of water pump activation.	Classification	**1.** Retry pump. **0.** Notify fault observer.
L1.PRE The light intensity shall decrease from 12pm to 12am and be $\leq$ 500 within that time.	Regression	**1.** Activate grow lights. **0.** Notify fault observer.	**L1.POST** The light intensity shall increase with a slope of $\geq$ 100 within 10 seconds of grow light activation.	Classification	**0.** Notify fault observer.
P1.PRE The current slope shall be ≥ 0.2 after at least 1 day since the last filter clean.	Classification	**0.** Notify fault observer.	**P1.POST** The current slope shall be ≤ 0.2 within 1 day of the last P1.DAR event.	Classification	**0.** Notify fault observer.

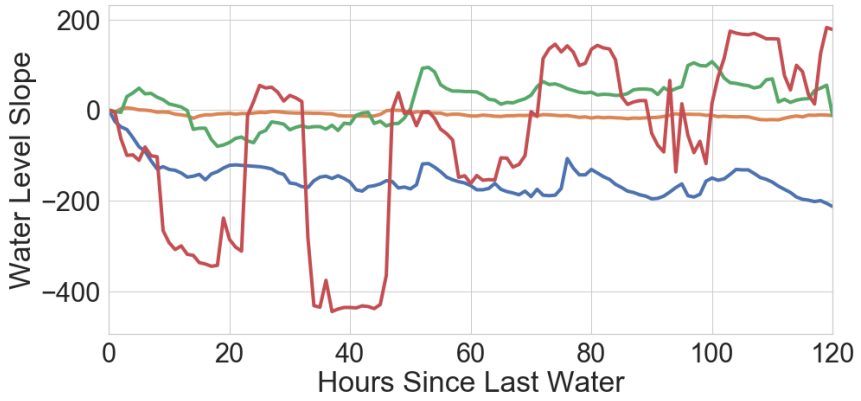

Figure 7.11: W1.PRE: slope of water level data before pump activation, showing reasonable (blue) and erroneous (other) data trends.

last resort because the MTTR would sharply increase when an automated solution was unable to fix the problem.

W1.PRE

There was a clear trend for how water level data should have behaved before passing the 500 threshold, namely: the water level declined over ~5 days i.e. 120 hours. Three circumstances could threaten correct FT support: (1) the slope had decreased too quickly, suggesting human sensor tampering; (2) the slope had not decreased at all in 5 days; or (3) there had been a positive trend, suggesting sensor malfunction. Trend normalisation (Figure 7.11) showed that a reasonable trend's slope was −200 after 120 hours (blue). Other trends (orange, green, red) should be detected early and masked.

A dataset was built containing 18 examples of water level declines over 120 hours, with 2724 data points in total. Any water level decline that surpassed −200 after at least 72 hours was labelled reasonable, and erroneous either if it surpassed −200 before 72 hours, or not at all. This resulted in 9 reasonable and 9 erroneous examples. After training KNN, SVM, and RFC models, the KNN model yielded the best result (Figure 7.12) because its decision boundary best reflected the trend: a decision boundary that decreased over time and remained below 0. Its accuracy and F1-Score of 91.2% narrowly outperformed the RFC model (Table 7.4).

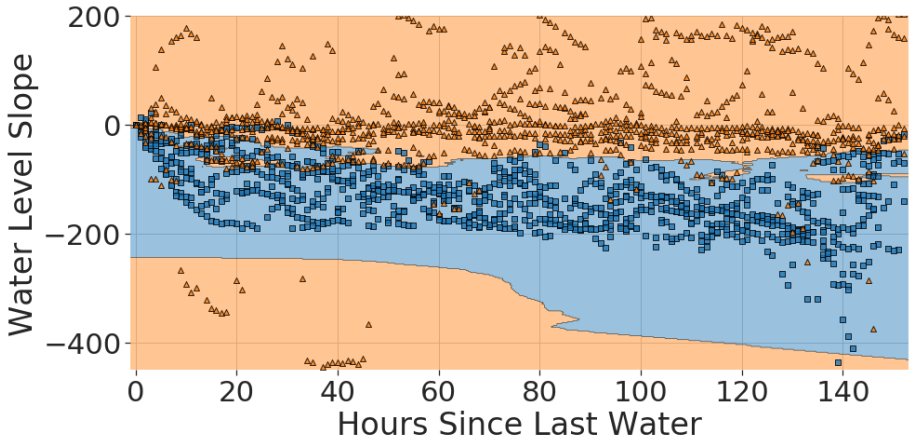

Figure 7.12: W1.PRE: KNN model to predict reasonable (blue) and erroneous (orange) data trends.

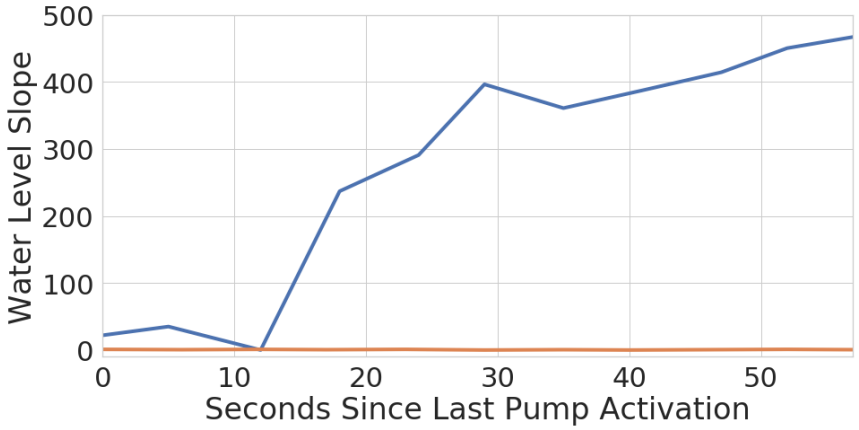

Figure 7.13: W1.POST: slope of water level data after pump activation, showing reasonable (blue) and erroneous (orange) data trends.

W1.POST

After pump activation, the data trend had a large increase over a 60 second period to some higher value, and then stabilised. Conversely, no value change would have suggested a failure to pump water to the reservoir e.g. hardware malfunction, empty water container(s). Trend normalisation (Figure 7.13) showed that a reasonable trend began shortly after activation (blue), and a lack of trend otherwise (orange).

A dataset was built containing 22 examples of water level increases, with 229 data points in total. Any increased with a slope ≥ 100 within the 60 second period was labelled reasonable, and erroneous otherwise. This resulted in 16 reasonable and 6 erroneous examples. On training the models, the KNN model achieved the highest

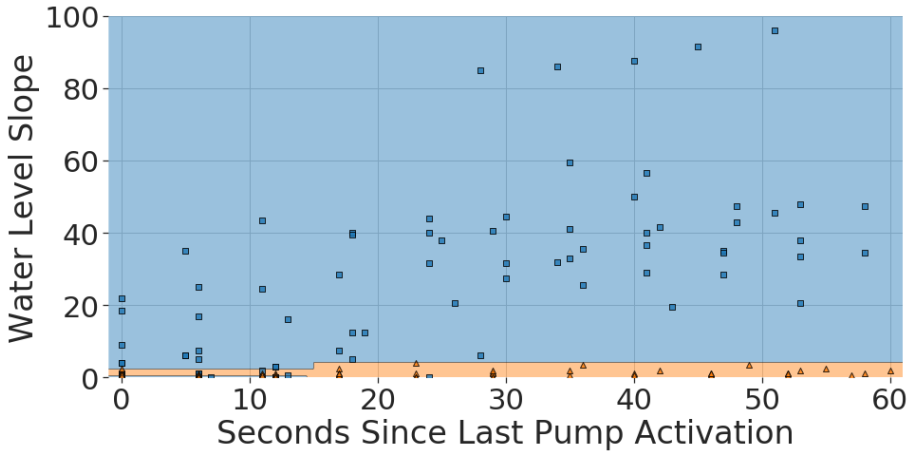

Figure 7.14: W1.POST: RFC model to predict reasonable (blue) and erroneous (orange) data trends.

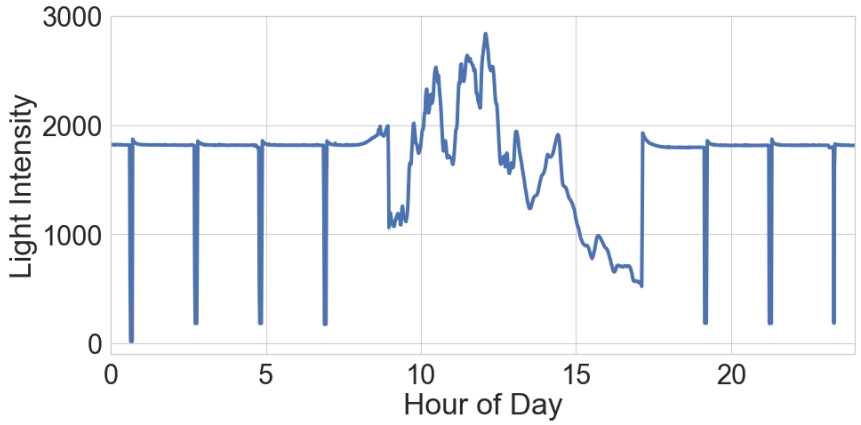

Figure 7.15: L1.DAR: light intensity data over a 24-hour period.

accuracy and F1-Score. However, the KNN showed signs of overfitting by creating small clusters of data points not reflective of the data trend. The RFC model (Figure 7.14) best reflected the trend of the data: a straight horizontal threshold dividing a positive trend from no trend, so this model was selected instead of KNN. The RFC model had an accuracy 87.9% and F1-Score of 91.4%.

7.4.4 L1: Natural Light Fluctuation

L1.DAR

This requirement was defined to increase the growth rate of produce, by activating grow lights when the room was dark to ensure photosynthesis even when natural light was unavailable. The limit-check NFA was used again, to identify when light

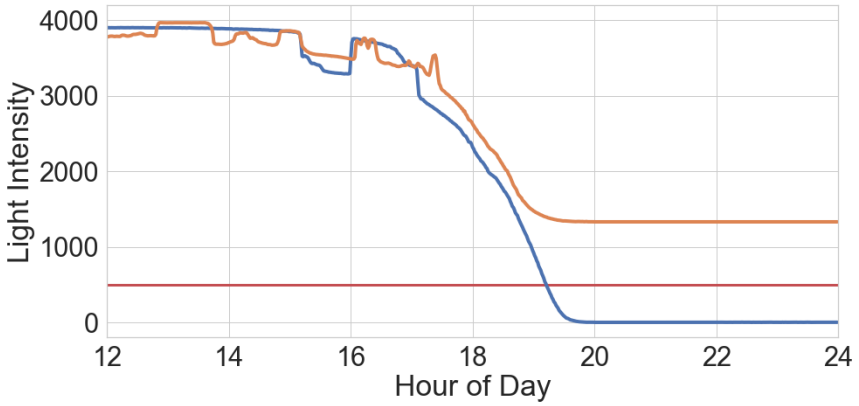

Figure 7.16: L1.PRE: slope of reasonable (blue) and unreasonable (orange) light intensity data from 12pm to 12am (not including data from when lights were activated).

intensity was $0 \leq e \leq 500$. A complementary requirement, *L2.DAR*, was also defined (Table 7.2) that deactivated grow lights by using the performance-check NFA (Figure 5.3a) to identify when 2 hours had passed since the grow lights were last activated. They required deactivation after 2 hours of continuous use in order to prevent damage by overheating, which this check ensured.

Figure 7.15 shows light intensity data that was representative of the type of data produced from the medium-scale VFS. On low light-intensity at ~5pm, the default assessment inferred that this was caused by natural light fluctuations, and would activate the lights. In the hours thereafter, data would frequently drop to 0 when lights were deactivated after 2 hour periods (Figure 7.15, 8pm-6am). After a 5 minute deactivation period, they would reactivate if natural light were still unavailable.

L1.PRE

Before light activation, light data would gradually low after ~12pm and decline below 500 in the evening. Failure to decline below 500 between 12pm-12am would suggest a sensor malfunction. Trend normalisation (Figure 7.16) showed that reasonable light intensity in the evening was < 500 e.g. between 6-8pm (blue). Erroneous data would not do this (orange) and would fail to trigger light activation.

A dataset was built containing 19 examples of light intensity decrease, with 70468 data points in total. These were only sampled at times when lights were deactivated,

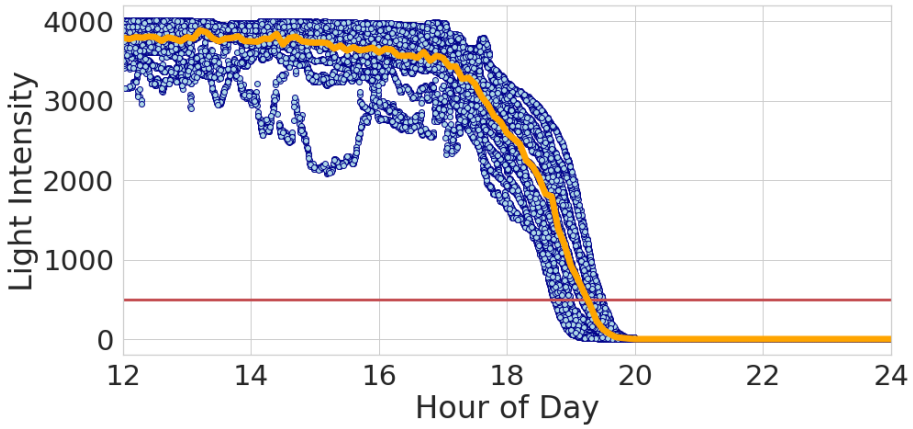

Figure 7.17: L1.PRE: RFR model to predict expected light intensity at a given hour of day (orange).

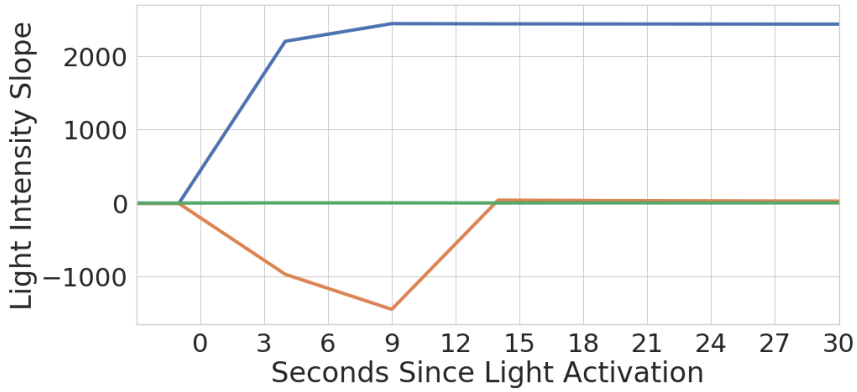

Figure 7.18: L1.POST: slope of light intensity data after light activation, showing reasonable (blue) and erroneous (other) data trends.

so the trend that exists below the 500 threshold (Figure 7.16, red line) could be observed. The RFR model (Figure 7.17) yielded the lowest *Root Mean Square Error* (RMSE) because it was able to create a horizontal line between 8pm-12am, rather than a smooth line that was erroneously < 0, as the ABR model was.

L1.POST

After light activation, the data trend spiked to some higher value and then stabilised. Conversely, no such trend suggested grow light failure. Trend normalisation (Figure 7.18) showed that a trend should have spiked when the next data payload arrived i.e. within 5 seconds (blue); other trends (green, orange) suggested failure.

A dataset was built containing 36 examples of light intensity increases, with 3527 data points in total. Any increase ≥ 100 within 10 seconds of light activation was

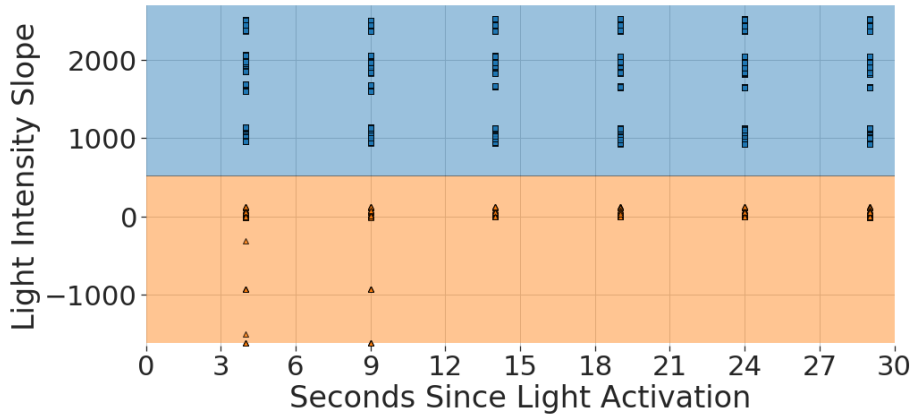

Figure 7.19: L1.POST: KNN model to predict reasonable (blue) and erroneous (orange) data trends.

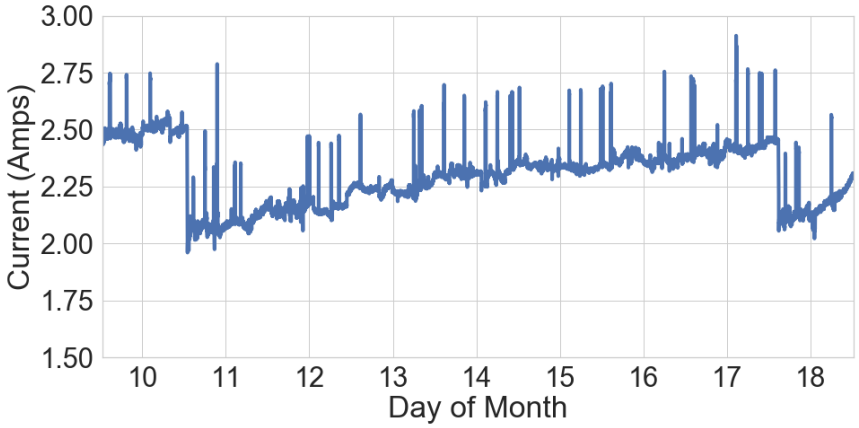

Figure 7.20: P1.DAR: current data over a 9-day period.

labelled reasonable, and erroneous otherwise. This resulted in 24 reasonable and 12 erroneous examples. The KNN and RFC models produced virtually the same decision boundaries (Figure 7.19). However, KNN's was marginally higher in accuracy at 95.7%. The KNN model created a horizontal boundary at ~500, which enabled the prediction as to whether a reasonable trend had, or was likely, to appear.

7.4.5 P1: Pump Filter Blocked

P1.DAR

This requirement was defined to prevent excessive energy consumption and water contamination by notifying staff to clean the filters on high current. The typical current trend (Figure 7.20) showed that, when it had risen by 0.2 after 3 days,

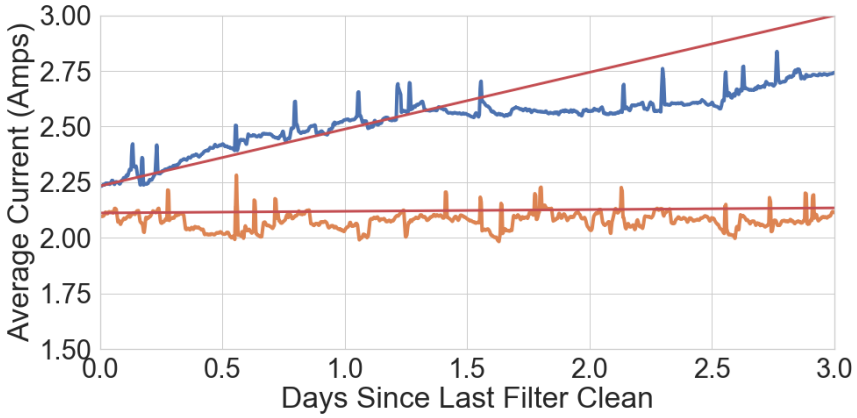

Figure 7.21: P1.PRE: rolling average of current data since last filter clean, showing reasonable (blue) and erroneous (orange) trends, and their average slopes (red lines).

system admins at IGS would clean the filters, which caused a current reduction. The trend-check NFA (Figure 5.2b) was used to check whether some data trend was unreasonable. To minimise current spikes (Figure 7.20), the trend check performed a rolling average of the first 20 current events since the filters were last cleaned, and the 20 most recent current events at least 3 days later[1], as follows:

$$(avg(e_{m-20}, ..., e_m) - avg(e_1, ..., e_{20}))/2 \geq 0.2 \tag{7.1}$$

Where $e_{m-20}, ..., e_m$ are the latest 20 events, which have occurred at least 3 days after the first 20 events $e_1, ..., e_{20}$. On high current detection, the default assessment assumed that the sustained current increase was caused by clogged filters, and the recovery was to notify system admins to clean them.

P1.PRE

Before admin notification, pump current data would gradually increase over at least 2 days with a slope ≥ 0.2. Failure to achieve this trend within 7 days suggested other failures (e.g. pump malfunction) because clogged filters were an inevitable natural system failure. Visualising these trends (Figure 7.21) showed the difference between reasonable (blue) and erroneous (orange) current behaviour over time.

A dataset was built containing 22 examples of pump current increase, with 152166

[1]The current was sampled every 30 seconds and so 20 events represent ~10 minutes of data.

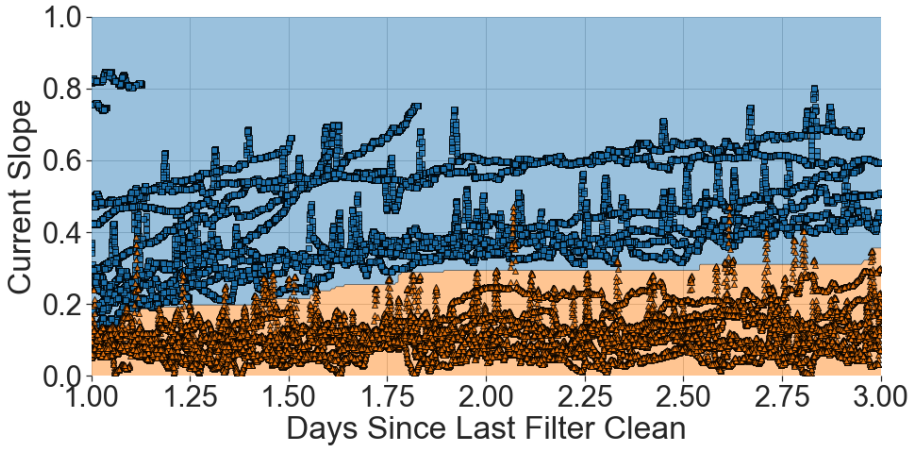

Figure 7.22: P1.PRE: RFC model to predict reasonable (blue) and erroneous (orange) data trends.

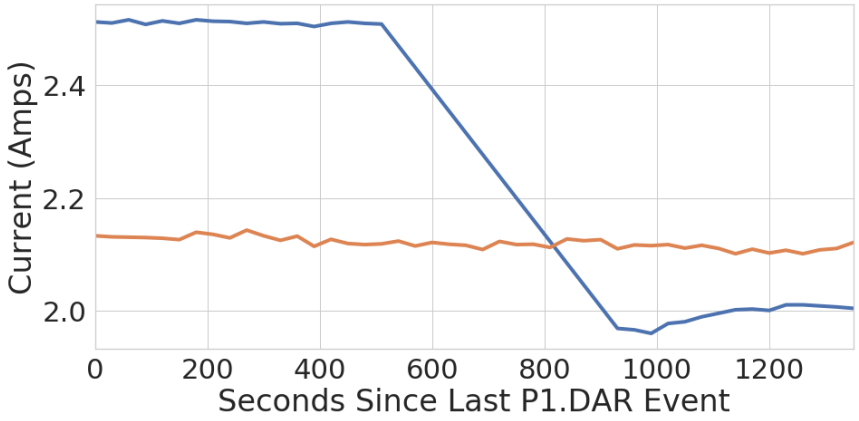

Figure 7.23: P1.POST: rolling average of current data since last filter clean, showing reasonable (blue) and erroneous (orange) data trends.

data points in total. Any examples that fulfilled the trend check predicate above were labelled reasonable, and erroneous otherwise: this led to 14 reasonable and 8 erroneous examples. The KNN model had achieved the highest accuracy and F1-Score (Table 7.4). However, its decision boundary was very jagged and had overfit the data. The RFC model (Figure 7.22) achieved comparable results with a boundary that better reflected the data trend.

P1.POST

After admin notification, there was a MTTR of ~700 seconds when filters were cleaned, causing the current to sharply decline to its original level. Conversely, no sudden current change in at least 1400 seconds (i.e. double the MTTR) might have

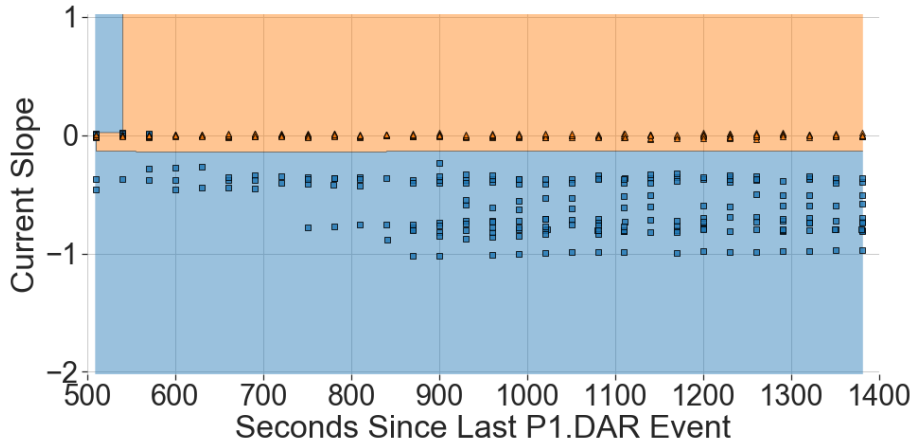

Figure 7.24: P1.POST: RFC model to predict reasonable (blue) and erroneous (orange) data trends.

suggested that IGS system admins failed to clean the filters in good time. Visualising these trends (Figure 7.23) showed a reasonable trend (blue) resulted in a sharp current drop by ~0.4 amps, and erroneous (orange) trends had no such change.

A dataset was built containing 22 examples of 14 reasonable and 8 erroneous data trends, with 891 data points in total. Erroneous trends were portions of the dataset where ≥ 7 days had passed without any significant current changes, which suggested the filter had not been cleaned. The RFC model (Figure 7.24) achieved the highest accuracy of 98.5% and was able to provide a horizontal boundary between a trend and lack of trend. This facilitated predictions whereby a lack of trend at *any* future time was always erroneous.

7.5 Experiments: BoboCEP

BoboCEP is designed to be a CEP system suitable for reliable edge computing, where partially completed complex events are distributed across k software instances via active replication. This makes it the ideal platform for reactive FT because FT support can be dependable with minimal service loss. Two such advantages of BoboCEP are covered in this section: (1) the ability to provide long-term event processing that can withstand BoboCEP instance failures; and (2) the ability to load balance incoming data across instances without affecting how complex events are generated. Furthermore, the performance of the Python implementation of BoboCEP is explored in

Table 7.4: Models trained in the experiments from Section 7.4, with the selected models in bold.

Requirement	Model	Accuracy	F1-Score	RMSE
W1.PRE	**KNN**	**0.911**	**0.911**	-
	SVM	0.512	0.677	-
	RFC	0.895	0.895	-
W1.POST	KNN	0.913	0.942	-
	SVM	0.741	0.851	-
	RFC	**0.879**	**0.913**	-
L1.PRE	ABR	-	-	307.77
	SVR	-	-	305.39
	RFR	-	-	**288.47**
L1.POST	**KNN**	**0.957**	**0.966**	-
	SVM	0.692	0.787	-
	RFC	0.956	0.965	-
P1.PRE	KNN	0.998	0.998	-
	SVM	0.717	0.678	-
	RFC	**0.994**	**0.994**	-
P1.POST	KNN	0.916	0.927	-
	SVM	0.382	0.552	-
	RFC	**0.985**	**0.988**	-

terms of its ability to instantiate runs and process large volumes of primitive events from the data stream.

As stated in Section 6.2.2, the medium-scale VFS used BoboCEP (v0.35), developed using Python (v3.7). Its Receiver consumed stream data via a Flask (v1) server that enabled microcontrollers to send data approximately every 3 seconds to three BoboCEP instances running on three Raspberry Pi (v2 Model B) edge devices: *Edge1*, *Edge2*, and *Edge3*. Shelf1 contained microcontrollers LL, LR, which, by default, routed their data to Edge1; ML, MR to Edge2; and RL, RR to Edge3. On an edge device failure, microcontrollers sending data to the failed device would randomly pick another edge device through which to reroute their sensor data. RabbitMQ (v3.7) was used for the BoboCEP message broker.

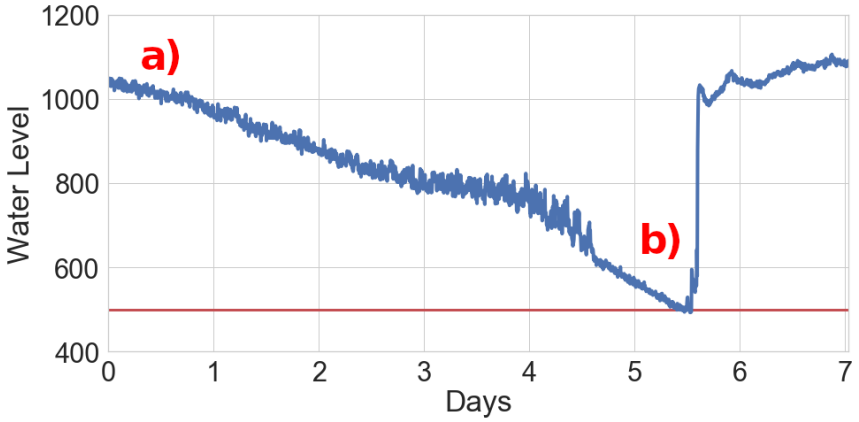

Figure 7.25: Water level data from a reservoir over a 7-day period.

7.5.1 FT Scenario

Long-Term Event Processing

To demonstrate long-term event processing, the trend-check NFA (Figure 5.2b) was used, which had two states: the first consumed an initial water-level data event e_1; the second consumed a water-level event e_2 such that e_2 was at least 3 days after e_1 and satisfied:

$$(e_2 - e_1)/2 \geq -200 \wedge e_2 \leq 500 \tag{7.2}$$

This ensured a significant negative trend since e_1, and a low water-level value overall (i.e. ≤ 500). Run fulfilment would cause pump activation for the appropriate shelf. Figure 7.25 shows water level data that was representative of the type of data the medium-scale VFS produced. It demonstrates a decline of water level within 1 week, and passes the 500 threshold at approximately 5.5 days, which would activate a water pump to replenish its reservoir that had run dry.

When a BoboCEP instance received the first water-level value e_1 (Figure 7.25a), which contained value 1039, it would trigger run instantiation for the trend-check NFA. The instance that created the run would then send a `CLONE` signal (Section 6.1.3) via the message broker, so that all other instances have a copy of the run also.

The second water-level value e_2 (Figure 7.25b), which contained value 499, would

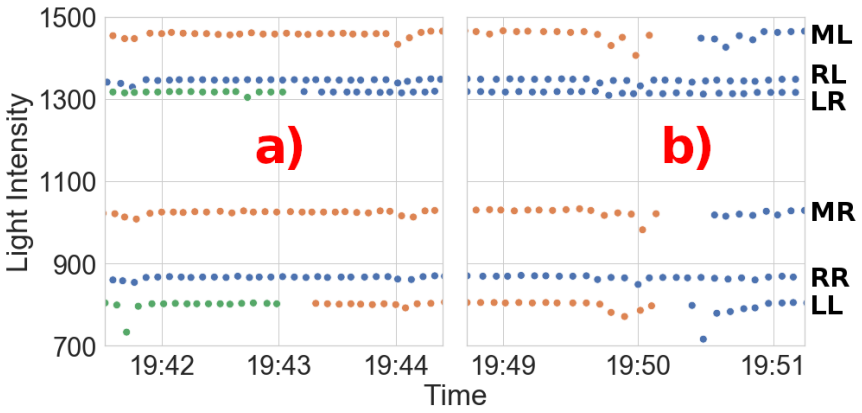

Figure 7.26: Light intensity data from Edge1 (green), Edge2 (orange), and Edge3 (blue).

fulfil the above predicate i.e. $(499-1039)/2 = -270$. The instance that first reached the run's final state would send a `FINAL` signal to all other instances, so that they would complete their copies of the run also. A pump activation would occur (Figure 7.25b) and lead to a sharp increase in water-level value, which would trigger an `ACTION` signal for the other instances to know that it was executed successfully.

Load Balancing

The microcontrollers for the medium-scale VFS were designed to send data to one edge device and, if this failed, would immediately select one of the other two edge devices at random through which to reroute future data; the microcontroller would simply try again with the next payload.

Microcontrollers LL, LR were connected to Edge1; ML, MR to Edge2; and RL, RR to Edge3, as specified in Section 6.2.2. Light intensity data from the microcontrollers were stable over a 10 minute interval. The BoboCEP instance on Edge1 was manually terminated (Figure 7.26a) and, after a brief data blackout, LL rerouted to Edge2, and LR to Edge3. Later, Edge2's instance was terminated (Figure 7.26b), which caused all of Edge2's microcontrollers to reroute Edge3 i.e. the last remaining device. Despite two edge device failures, all sensor data was still available to the FT-support system, and was balanced across any available instances.

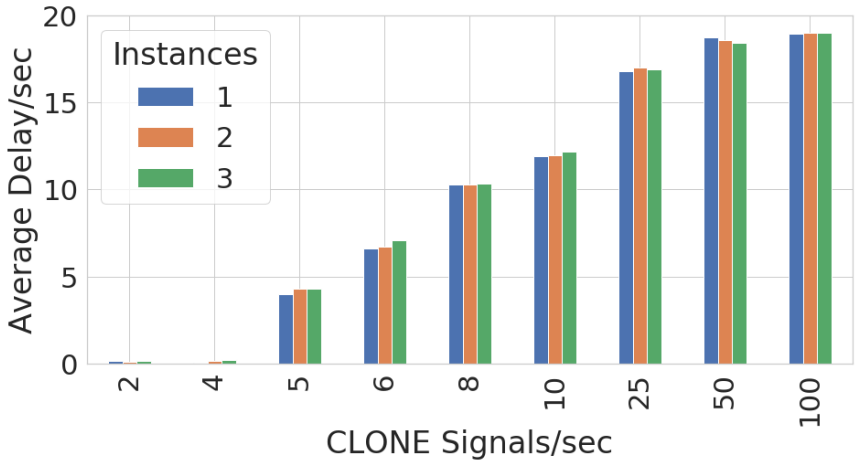

Figure 7.27: Data from the run instantiation experiment.

7.5.2 Performance

Run Instantiation

This experiment tested how instances coped with a large volume of `CLONE` signals. One, two, and three BoboCEP instances were set up, and 100 `CLONE` signals were passed to them at varying rates, which would eventually lead to 100 runs being cloned across all instances. The time to process 100 signals on each instance was averaged to calculate the average processing delay (Figure 7.27). Results showed that instances incurred progressively larger delays as the rate of `CLONE` signals increased. However, an increase in instances did not lead to a delay increase because the message broker serialised and broadcast `CLONE` signals to all instances equally.

Rule Throughput

This experiment tested how well an instance could handle a stream of data events relative to the number of NFA handlers it had. A BoboCEP instance was preloaded with k handlers that each had one incomplete run, so that a data event would be checked against both the handler, which determined whether a new run should be instantiated, and its run. None of the data events passed in this experiment were designed to cause a `CLONE` or `TRANSITION`. 1000 data events were passed to the instance in order to measure how long it took to process them all (Figure 7.28).

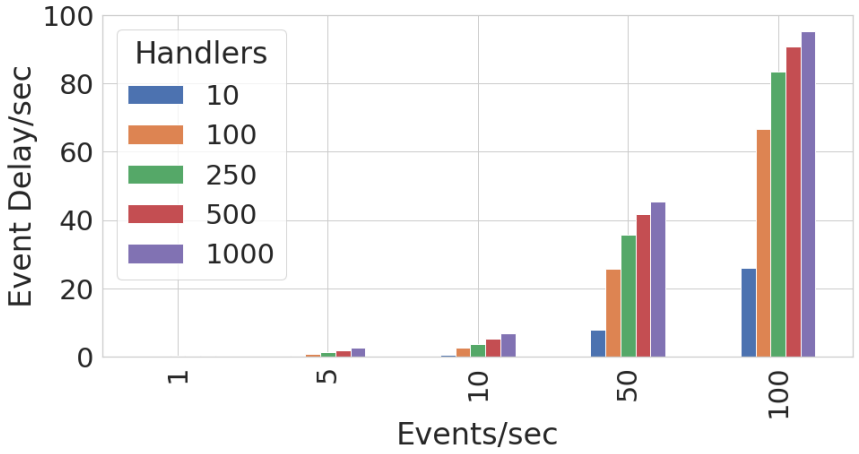

Figure 7.28: Data from the rule throughput experiment.

Results showed that the event-processing delay increased with the data rate, and larger delays occurred as the number of handlers increased.

7.6 Summary

This chapter evaluated the proposed FT framework to validate the key research objectives of the thesis, as outlined in Section 1.3. The first set of experiments in Section 7.2 provided an assessment of CPoF and its ability to provide effective, modular, reusable reactive-FT support via two failure scenarios: an empty water container, and data transmission degradation. The results of the experiments showed that a variety of NFAs could be used to detect them: trend, correlation, performance, and persistence checks.

The second and third set of experiments in Sections 7.3 & 7.4 applied both reactive and proactive FT on four failure scenarios. Specifically, these experiments helped to inform the development and integration of strategies and mechanisms for mitigating the faults into the framework, and resulted in four approaches for implementing effective proactive FT: (1) correlating system events with time (i.e. context-specific information); (2) correlating events with other detected error events; (3) identifying erroneous conditions that could hamper error detection; and (4) identifying erroneous conditions that might suggest error-recovery failure. The results of the proactive-FT

models generated (Tables 7.1 & 7.4) showed that it was possible to preempt a wide variety of erroneous system behaviour with favourably high accuracy.

Finally, the fourth set of experiments in Section 7.5 evaluated BoboCEP's Python implementation specifically, by demonstrating long-term error processing and load balancing with the medium-scale VFS testbed. A performance evaluation was also performed on BoboCEP which showed that the implementation coped well with reasonable loads but incurred delays under high-velocity data flows.

Chapter 8

Conclusion

This chapter concludes the work in this thesis by first reviewing the research objectives outlined in Section 1.3. The fulfilment of these objectives is manifest in the proposed concepts and the evaluation that implements them using the VFS use case. The chapter then reflects on the limitations of the work and the potential future directions to take in order to address them. Finally, the chapter concludes by summarising the findings of this thesis.

8.1 Objectives Revisited

This section revisits the research objectives described in Section 1.3 and discusses how the FT framework proposed in this thesis satisfies these objectives. The objectives were identified based on the challenges and limitations of currently proposed FT-support initiatives (Section 1.2). The objectives are compared against the results obtained, matching the key contributions to its related objective, as follows:

1. **To identify and classify faults events and fault patterns in IoT systems**. The first objective was to have a thorough review of the different faults which could occur within IoT systems and classify the faults so that patterns could be identified regarding what faults occurred, their severity, their effects on the system, and what was likely to cause them.

   The literature review (Section 3.1) was the first indication that a mechanism to classify fault events and patterns was needed, in order to provide a means of comparing the FT support present in different IoT systems. This became

the VFF framework (Section 4.2) that enabled system defect categorisation via the vulnerabilities, faults, and failures associated with the defects.

CPoF (Section 7.2) provided a language-agnostic, reusable means of error definition, to which VFF could be applied. System errors were detected using various NFAs and used a CEP system as the engine to implement the NFAs. CPoF was validated during the evaluation (Section 7.2) and it was determined that CPoF was able to reuse its error-checking NFAs to provide reactive-FT support for failure scenarios in the small-scale VFS testbed.

2. **To develop a service-oriented fault-tolerance framework for IoT systems that combines both reactive- and proactive-FT support**. The second objective was to build on the first objective by identifying how reactive- and proactive-FT support can work together to provide the FT framework with the ability to anticipate future errors via proactive FT given previous experience of the error.

   The correlation-check NFA (Section 5.2.2) was used as the means of passing important error-detection experience to the proactive FT aspect of the FT framework. When used with CvR (Section 5.3), correlation checks were able to provide information regarding whether an error was probably going to happen imminently or whether it was not. This was demonstrated in the experiments from Section 7.3 that showed time and error correlations being used to predict and react to imminent future errors.

3. **To incorporate strategies and mechanisms that facilitate effective fault mitigation**. The third objective was to perform an extensive review of strategies to resolve system errors that were reactively and proactively detected in the second objective. The experiments from Section 7.3 had demonstrated the need for fault mitigation by preempting an imminent battery depletion and switching service over to a redundant device that could provide equivalent data to the nearly-failed device.

   In Section 7.4, a formal approach to fault mitigation was proposed using the DARA framework that considered how data might influence erroneous error de-

tection. That is, if data were erroneous, then it might cause an error-detection event to be generated for an error that is not present (i.e. false positive), or it might cause an error to go undetected (i.e. false negative).

This led to the notion of Pre-Detect and Post-Recover requirements in the DARA process (Section 7.4.2) to formally identify the types of data errors that might prevent correct FT. By defining Pre-Detect and Post-Recover requirements, fault mitigation strategies could be planned to circumvent the data-error problem, for example, via temporal redundancy or re-executing tasks on redundant hardware (Table 7.3).

4. **To ensure that the framework can scale to more complex IoT scenarios**. This objective focussed on the importance of the the framework being able to work effectively at any system scale. The proposed FT-framework design (Chapter 4) delegates FT support to two microservices: one provides reactive FT and the other proactive FT. Microservices are modular and lightweight, which means that they have the flexibility to be placed anywhere in an IoT system's infrastructure and easily replicated for redundancy.

   The VFSs from Section 6.2 demonstrated that the FT framework could work effectively at both a small and medium scale. The small-scale VFS was used in the experiments from Sections 7.2 & 7.3, and the medium-scale VFS was used in the experiments from Sections 7.4 & 7.5. Furthermore, the FT framework was able to detect erroneous behaviour in a dataset from a large-scale, real-world VFS in the experiments from Section 7.4, which helps to further validate the FT framework's efficacy in providing FT support at scale.

   The BoboCEP engine (Section 6.1) was designed to provide CEP at scale for use in the medium-scale VFS by distributing event processing across k devices at the network edge, with experiments demonstrating its ability to perform long-term event processing despite hardware crashes, as well as load balancing data from multiple data sources (Section 7.5).

8.2 Reflection

8.2.1 Limitations

Many of the design and implementation choices in this thesis resulted in a solution that could provide the key desired qualities from Section 1.4, namely, an FT-support solution that is interoperable, reusable, dependable, and scalable. However, there are limitations to the proposed solution that affect its applicability in some circumstances, as follows.

- **IoT systems require a client-server architecture model to use the FT-support framework**. The key enabling technologies used in the proposed FT framework and its implementations from Chapter 6 are: (1) edge network devices, to run reactive-FT support; (2) the fog and cloud, to run proactive-FT support; and (3) microservices, to enable easy integration of FT support into an existing centralised IoT system. Unfortunately, the assumption that this infrastructure will be in use before the FT-support framework is incorporated will mean that some IoT systems will not be suitable; in particular, IoT systems that adopt a constrained, decentralised architecture like in [186]. The client-server architecture also means additional costs with regard to the cloud infrastructure that is required for data storage and ML predictive analytics, as well as an edge network infrastructure for deploying CEP microservices near an IoT system's sensor network. In very remote locations with limited Internet connectivity, it is unlikely that an edge computing platform could actually be deployed, or for there to be a reliable connection to a cloud platform, without excessive additional costs to provide a more reliable Internet connection.

- **CEP is only suitable for data-centric error detection**. The proposed error-checking NFAs are designed for IoT stream-data analytics to infer erroneous system behaviour via data. This approach to FT is advantageous in that it makes the FT framework pluggable: it can be attached to an existing IoT system and consume streaming data like any other process already in the system. However, many of the internal system states may not be exposed as

a data representation for the FT framework to consume and analyse. For example, the structural and diagnostic checks from [122] are designed to check the structural integrity of data and the correctness of system components via black-box checking of system interfaces, respectively. These two checks require a low-level FT implementation to properly assess whether internal interfaces and data representations are correct, which would be difficult to implement in the proposed FT framework.

- **Proactive-FT solution is not appropriate for critical applications**. A key design choice for proactive-FT support is that it would use data from reactive FT as ground truth, by learning how errors are detected and recovered from, and using this experience as data to train predictive models. This provides the advantage that proactive FT is an automatic learning process that does not require human intervention. However, it is trained on the assumption that data from reactive-FT support is always correct. Erroneous data, or unexpected changes to system context, would make the models susceptible to *concept drift*, which is where the relationship between input data and the target variable changes over time [81]. For critical applications, especially safety-critical IoT systems, a manual proactive-FT solution would be more appropriate.

8.2.2 Lessons Learned

During the early development of the FT framework, it was designed to be a generic service-oriented architecture, where all processes resided on a single software instance. This early design proved to be too restrictive and heavyweight, and did not reflect the rising trend of distributing cloud functionality to the edge/fog. As a consequence, a microservices architecture was adopted instead, and individual subprocesses were delegated to their own microservices (Section 4.1.1). This design decision led to a more IoT-appropriate solution that made developing each system service more manageable and scalable. Furthermore, it made software crashes less consequential on the system. For example, if the Edge microservice failed, it would not affect any data processing on the Real-Time FT microservice.

Another realisation was the need for a bespoke CEP system. Despite there being many CEP systems available to use, few were actually appropriate for the IoT domain. The FlinkCEP CEP system was initially used with the small-scale VFS (Section 6.2.1). However, FlinkCEP was primarily designed for use with the wider *Apache Flink* platform[1] that is designed for distributed stream- and batch-data processing. This led to many compatibility issues and, due to FlinkCEP's heavyweight design, memory issues that resulted in frequent software crashes on edge devices. For this reason, BoboCEP was developed with few software dependencies, making it lightweight enough to run on the network edge. Its IoT-specific design made it appropriate as an FT-support platform, e.g. the active replication of partially completed complex events.

Initial experiments for proactive-FT detection used the internal states of NFA runs as features for predictive models, and would attempt to predict whether a run would reach its accepting state or its halt state. However, with this approach, there were very few scenarios which were useful. Consequently, this led to more research into context awareness, and so subsequent experiments focussed on correlating composite events with contextual system data as well as other composite events.

8.3 Future Work

This section outlines future directions for the development of the FT framework, such as further improvements to its design and features that could be added to enhance its ability to provide effective FT for IoT systems.

- **Provide wider protocol and data representation support**. The key quality of the proposed FT framework is its ability to be interoperable with as many IoT systems as possible. As such, a primary design decision was for it to have platform, syntactic, and semantic interoperability (Section 4.1). The design led to a microservices approach where interfaces were discoverable via the OpenAPI Specification and data could be interpreted via the Web Thing API. However, there are numerous other ways in which data can be represented

[1] https://flink.apache.org

and transmitted throughout an IoT system, and for this framework to be truly interoperable, it must cater to these different standards and protocols.

Future iterations of the system design would incorporate more data formats than just JSON. In particular, legacy formats, such as XML that is less efficient but more human readable; and also more recent formats, such as *Binary JSON* (BSON)[2] that is more efficient but less human readable.

- **Support for Linked Data to provide more intelligent error detection**. Linked Data refers to machine-readable data that is linked to other external data and can in turn be linked to from external data, which results in a "Web of Data" that can more accurately describe things in the world [27]. By harnessing the power of Linked Data, the FT framework would have a richer understanding of the data it is analysing, in order to draw more context awareness into the error-detection process. The JSON format used by the FT framework has been extended to provide a means of linking JSON data, called *JSON for Linking Data* (JSON-LD)[3], which would be a useful extension to incorporate more semantic interoperability into the FT framework.

- **Declarative FT definitions for BoboCEP**. In BoboCEP (v0.35), the NFA checks (Section 5.2.2) were directly implemented in the software as automata, and error-recovery actions were subscribed to relevant NFAs such that, if a composite event were generated, the subscribed actions for the given NFA would be executed. This is a very explicit approach that enables a system designer to precisely define the detection and recovery procedures for BoboCEP instances to trigger.

  A future direction for the software would be an interface that abstracts the NFA error-definition process from the system designer. The interface would instead follow the process of the DARA framework (Section 7.4.1) by describing the failure semantics i.e. the likely ways in which the system could fail, and for BoboCEP to automatically generate the appropriate NFAs and actions to

[2]http://bsonspec.org
[3]https://json-ld.org

handle the errors commonly associated with those failures. This would provide a declarative, SQL-esque approach to error definition that focusses more on system requirements than implementation.

- **More error-detection automata**. Hanmer [91] provided an exhaustive classification of software error-detection patterns. For example, they include: (1) *complete parameter checking*, to check inputs and parameters in order to prevent erroneous task execution; (2) *routine audits*, to check data using routine background tasks in order to ensure correctness; and (3) *leaky bucket counter*, to maintain a counter that automatically increases and decreases with the number of errors ignored by the system, which helps to prevent too many errors being ignored that might accumulate and cause a large-scale error in the future.

  Many of Hanmer's proposed error-detection patterns overlap with the existing NFAs from Section 5.2.2, and some do not. In future work, more automata could be devised to match each of the error-detection patterns by Hanmer, which would ensure complete coverage of all possible error-detection approaches identified in previous FT literature.

8.4 Final Remarks

IoT has led to a surge in the number of connected Internet-connected devices and is growing at an exponential rate, which is contributing to ever-increasing, massive data volumes that require real-time analytics in order to extract business value [201]. For IoT systems to deliver service on this scale, they must be fault tolerant, in order to tolerate any faults remaining in the system after its development and to protect against the constant threat of service failures [163].

This thesis provided a representative review of current FT implementations in IoT literature which identified that they suffer many drawbacks. For example, they were designed for specific faults [96, 103], specific applications [86, 2, 208], and were dependent on decentralised networks [46, 186] as well as bespoke devices [96] and protocols [208]. The research in this thesis set out to provide a solution to

this heterogeneity and incompatibility by providing a means of expressing erroneous system behaviour as NFAs that allow for reusable, language-agnostic reactive FT via CEP. Furthermore, ML was used to facilitate proactive FT by learning from the errors encountered by its reactive-FT counterpart. Experiments were conducted to test the efficacy of the FT framework, and showed that CEP and ML were effective as complementary means of providing FT in various failure scenarios on VFSs.

While the evaluation of the FT solution was only performed with VFSs of varying design and scale, the solution is also applicable to use cases beyond VFSs. For example, smart manufacturing would benefit from the low-latency analysis of the condition of machinery using the inconsistency check NFA (Figure 5.5), which could check for uniformity between sensors that monitor system behaviour and identify any erroneous system behaviour if a particular sensor's data were to deviate from its replicas' data. Also, analysing functional correctness throughout the manufacturing process could be performed using the correlation check NFA (Figure 5.4) which could check that each stage in the manufacturing process had been successfully executed in the right order and with appropriate timeliness.

From the findings, recommendations are made to improve on the work by expanding beyond the client-service architecture style that is used throughout the thesis, as well as considering other means of detecting errors beyond the data-centric approach used by CEP systems. Future work in this area should investigate developing more NFAs for even more errors possible in IoT systems, and also to support Linked Data in order to enable more intelligent error detection.

References

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith and P. Steggles, 'Towards a better understanding of context and context-awareness,' in *International symposium on handheld and ubiquitous computing*, Springer, 1999, pp. 304–307 (cit. on p. 85).

[2] D. P. Abreu, K. Velasquez, M. Curado and E. Monteiro, 'A resilient internet of things architecture for smart cities,' *Annals of Telecommunications*, vol. 72, no. 1-2, pp. 19–30, 2017 (cit. on pp. 49, 58, 62, 155).

[3] H. Agarwal and A. Sharma, 'A comprehensive survey of fault tolerance techniques in cloud computing,' in *2015 International Conference on Computing and Network Communications (CoCoNet)*, IEEE, 2015, pp. 408–413 (cit. on pp. 32, 34).

[4] J. Agrawal, Y. Diao, D. Gyllstrom and N. Immerman, 'Efficient pattern matching over event streams,' in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 147–160 (cit. on pp. 100, 177).

[5] A. Akbulut and H. G. Perros, 'Performance analysis of microservice design patterns,' *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019 (cit. on p. 14).

[6] M. Akdere, U. Çetintemel and N. Tatbul, 'Plan-based complex event detection across distributed sources,' *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 66–77, 2008 (cit. on p. 83).

[7] C. Alcaraz and S. Zeadally, 'Critical infrastructure protection: Requirements and challenges for the 21st century,' *International journal of critical infrastructure protection*, vol. 8, pp. 53–66, 2015 (cit. on p. 31).

[8] S. R. Ali, 'Reliability testing for advanced networks,' in *Next Generation and Advanced Network Reliability Analysis*, Springer, 2019, pp. 277–304 (cit. on p. 15).

[9] Z. Ali, Z. H. Abbas and F. Y. Li, 'A stochastic routing algorithm for distributed iot with unreliable wireless links,' in *Vehicular Technology Conference (VTC Spring), 2016 IEEE 83rd*, IEEE, 2016, pp. 1–5 (cit. on p. 46).

[10] I. Álvarez, M. Barranco and J. Proenza, 'Towards a fault-tolerant architecture based on time sensitive networking,' in *2018 IEEE 23rd International Con-*

ference on Emerging Technologies and Factory Automation (ETFA), IEEE, vol. 1, 2018, pp. 1113–1116 (cit. on p. 31).

[11] D. Anicic, S. Rudolph, P. Fodor and N. Stojanovic, 'Stream reasoning and complex event processing in etalis,' *Semantic Web*, vol. 3, no. 4, pp. 397–407, 2012 (cit. on p. 89).

[12] G. Arunkumar and N. Venkataraman, 'A novel approach to address interoperability concern in cloud computing,' *Procedia Computer Science*, vol. 50, pp. 554–559, 2015 (cit. on p. 66).

[13] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher and P. Bose, 'Understanding the propagation of transient errors in hpc applications,' in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2015, pp. 1–12 (cit. on p. 19).

[14] R. A. Ashraf, O. Mouri, R. Jadaa and R. F. DeMara, 'Design-for-diversity for improved fault-tolerance of tmr systems on fpgas,' in *2011 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2011, pp. 99–104 (cit. on p. 25).

[15] A. Avižienis and J.-C. Laprie, 'Dependable computing: From concepts to design diversity,' *Proceedings of the IEEE*, vol. 74, no. 5, pp. 629–638, 1986 (cit. on p. 12).

[16] A. Avižienis, J.-C. Laprie, B. Randell and C. Landwehr, 'Basic concepts and taxonomy of dependable and secure computing,' *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004 (cit. on pp. 2, 12, 13, 15–17, 19, 20, 77, 89, 90).

[17] A. Avižienis, J.-C. Laprie and B. Randell, 'Fundamental concepts of computer system dependability,' in *Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*, Citeseer, 2001, pp. 1–16 (cit. on pp. 12, 13, 31).

[18] M. Awad and R. Khanna, 'Support vector regression,' in *Efficient Learning Machines*, Springer, 2015, pp. 67–80 (cit. on p. 117).

[19] A. Bala and I. Chana, 'Fault tolerance-challenges, techniques and implementation in cloud computing,' *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 1, p. 288, 2012 (cit. on p. 32).

[20] C. Barbero, P. Dal Zovo and B. Gobbi, 'A flexible context aware reasoning approach for iot applications,' in *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, IEEE, vol. 1, 2011, pp. 266–275 (cit. on p. 85).

[21] A. Bassi, M. Bauer, M. Fiedler, T. Kramp, R. van Kranenburg, S. Lange and S. Meissner, *Enabling Things to Talk: Designing IoT solutions with the*

IoT Architectural Reference Model. Springer Berlin Heidelberg, 2013, ISBN: 9783642404030 (cit. on pp. 64, 73).

[22] E. Bauer, *Design for Reliability: Information and Computer-Based Systems*. Wiley, 2011, ISBN: 9781118075081 (cit. on pp. 65, 72, 77).

[23] J. Bauer and N. Aschenbruck, 'Design and implementation of an agricultural monitoring system for smart farming,' in *IoT Vertical and Topical Summit on Agriculture-Tuscany (IOT Tuscany), 2018*, IEEE, 2018, pp. 1–6 (cit. on pp. 1, 104).

[24] K. Benke and B. Tomkins, 'Future food-production systems: Vertical farming and controlled-environment agriculture,' *Sustainability: Science, Practice and Policy*, vol. 13, no. 1, pp. 13–26, 2017 (cit. on p. 104).

[25] J. W. Bennett, G. J. Atkinson, B. C. Mecrow and D. J. Atkinson, 'Fault-tolerant design considerations and control strategies for aerospace drives,' *IEEE Transactions on Industrial Electronics*, vol. 59, no. 5, pp. 2049–2058, 2011 (cit. on p. 31).

[26] I. Beschastnikh, P. Wang, Y. Brun and M. D. Ernst, 'Debugging distributed systems,' *Queue*, vol. 14, no. 2, pp. 91–110, 2016 (cit. on p. 3).

[27] C. Bizer, T. Heath and T. Berners-Lee, 'Linked data: The story so far,' in *Semantic services, interoperability and web applications: emerging concepts*, IGI Global, 2011, pp. 205–227 (cit. on p. 154).

[28] G. Boloix and P. N. Robillard, 'A software system evaluation framework,' *Computer*, vol. 28, no. 12, pp. 17–26, 1995 (cit. on p. 112).

[29] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico and F. Grandoni, 'Discriminating fault rate and persistency to improve fault treatment,' in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, IEEE, 1997, pp. 354–362 (cit. on p. 34).

[30] A. Botta, W. De Donato, V. Persico and A. Pescapé, 'Integration of cloud computing and internet of things: A survey,' *Future generation computer systems*, vol. 56, pp. 684–700, 2016 (cit. on p. 3).

[31] L. Breiman, 'Random forests,' *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001 (cit. on pp. 116, 117).

[32] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte and W. White, 'Cayuga: A high-performance event processing engine,' in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 1100–1102 (cit. on p. 82).

[33] A. Bröring, S. Schmid, C.-K. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. Le Phuoc, J. Mitic, D. Anicic and E. Teniente, 'Enabling iot ecosystems

through platform interoperability,' *IEEE software*, vol. 34, no. 1, pp. 54–61, 2017 (cit. on p. 66).

[34] G. Brown, 'Ensemble learning,' in *Encyclopedia of Machine Learning*, C. Sammut and G. I. Webb, Eds. Boston, MA: Springer US, 2010, pp. 312–320, ISBN: 978-0-387-30164-8 (cit. on p. 116).

[35] D. Bruneo and S. Distefano, *Quantitative assessments of distributed systems: Methodologies and techniques*. John Wiley & Sons, 2015 (cit. on p. 33).

[36] A. Buchmann and B. Koldehofe, 'Complex event processing,' *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, vol. 51, no. 5, pp. 241–242, 2009 (cit. on pp. 2, 82).

[37] M. Butler, C. Jones, A. Romanovsky and E. Troubitsyna, *Rigorous Development of Complex Fault-Tolerant Systems*, ser. LNCS sublibrary: Programming and software engineering. Springer, 2006, ISBN: 9783540482659 (cit. on p. 91).

[38] B. Butzin, F. Golatowski and D. Timmermann, 'Microservices approach for the internet of things,' in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2016, pp. 1–6 (cit. on p. 71).

[39] R. Buyya and A. V. Dastjerdi, *Internet of Things: Principles and Paradigms*. Elsevier, 2016 (cit. on p. 67).

[40] C/S2ESC Software & Systems Engineering Standards Committee, 'IEEE 24765-2017 - ISO/IEC/IEEE International Standard - Systems and software engineering - Vocabulary,' IEEE, Tech. Rep., Sep. 2017 (cit. on pp. 13, 31).

[41] J. Cao, X. Wei, Y. Liu, D. Mao and Q. Cai, 'Logcep-complex event processing based on pushdown automaton,' *International Journal of Hybrid Information Technology*, vol. 7, no. 6, pp. 71–82, 2014 (cit. on p. 83).

[42] T. Chakraborty, A. U. Nambi, R. Chandra, R. Sharma, M. Swaminathan, Z. Kapetanovic and J. Appavoo, 'Fall-curve: A novel primitive for iot fault detection and isolation,' in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, ACM, 2018, pp. 95–107 (cit. on p. 121).

[43] M. Al-Chalabi, 'Vertical farming: Skyscraper sustainability?' *Sustainable Cities and Society*, vol. 18, pp. 74–77, 2015 (cit. on p. 104).

[44] P. P. W. Chan, M. R. Lyu and M. Malek, 'Making services fault tolerant,' in *International Service Availability Symposium*, Springer, 2006, pp. 43–61 (cit. on p. 31).

[45] S. Cherrier, Y. M. Ghamri-Doudane, S. Lohier and G. Roussel, 'D-lite: Distributed logic for internet of things services,' in *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, IEEE, 2011, pp. 16–24 (cit. on p. 38).

[46] ——, 'Fault-recovery and coherence in internet of things choreographies,' in *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, IEEE, 2014, pp. 532–537 (cit. on pp. 38, 61, 62, 155).

[47] S. Chetan, A. Ranganathan and R. Campbell, 'Towards fault tolerance pervasive computing,' *IEEE Technology and Society Magazine*, vol. 24, no. 1, pp. 38–44, 2005 (cit. on p. 4).

[48] C. Chilipirea, A. Ursache, D. O. Popa and F. Pop, 'Energy efficiency and robustness for iot: Building a smart home security system,' in *2016 IEEE 12th International Conference on Intelligent Computer Communication and Processing (ICCP)*, IEEE, 2016, pp. 43–48 (cit. on pp. 48, 61).

[49] J.-H. Cho, P. M. Hurley and S. Xu, 'Metrics and measurement of trustworthy systems,' in *MILCOM 2016-2016 IEEE Military Communications Conference*, IEEE, 2016, pp. 1237–1242 (cit. on p. 14).

[50] S. Choochotkaew, H. Yamaguchi, T. Higashino, M. Shibuya and T. Hasegawa, 'Edgecep: Fully-distributed complex event processing on iot edges,' in *2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, IEEE, 2017, pp. 121–129 (cit. on pp. 82, 97).

[51] P. K. Choubey, S. Pateria, A. Saxena, V. P. C. SB, K. K. Jha and S. B. PM, 'Power efficient, bandwidth optimized and fault tolerant sensor management for iot in smart home,' in *2015 IEEE International Advance Computing Conference (IACC)*, Bangalore: IEEE, 2015, pp. 366–370, ISBN: 9781479980475 (cit. on pp. 40, 41, 55, 61, 116).

[52] G. Choudhary and A. Jain, '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, 2016, pp. 1–8 (cit. on p. 1).

[53] S. Chouikhi, I. El Korbi, Y. Ghamri-Doudane and L. A. Saidane, 'A survey on fault tolerance in small and large scale wireless sensor networks,' *Computer Communications*, vol. 69, pp. 22–37, 2015 (cit. on p. 33).

[54] O. Çiço, Z. Dika and B. Cico, 'High reliability approaches in cloud applications for business-reliability as a service (raas) model,' *International Journal on Information Technologies and Security*, vol. 9, pp. 3–18, 2017 (cit. on p. 30).

[55] Cisco, 'Fog computing and the internet of things: Extend the cloud to where the things are,' Tech. Rep., 2015, Accessed: 2020-01-30. [Online]. Available: `https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf` (cit. on p. 65).

[56] D. Cotroneo, R. Natella, R. Pietrantuono and S. Russo, 'A survey of software aging and rejuvenation studies,' *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, p. 8, 2014 (cit. on pp. 33, 34).

[57] S. F. Crone, J. Guajardo and R. Weber, 'The impact of preprocessing on support vector regression and neural networks in time series prediction.,' in *DMIN*, 2006, pp. 37–44 (cit. on p. 116).

[58] G. Cugola and A. Margara, 'Complex event processing with t-rex,' *Journal of Systems and Software*, vol. 85, no. 8, pp. 1709–1728, 2012 (cit. on p. 82).

[59] ——, 'Processing flows of information: From data stream to complex event processing,' *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, pp. 1–62, 2012 (cit. on pp. 84, 86, 99, 100).

[60] ——, 'The complex event processing paradigm,' in *Data Management in Pervasive Systems*, F. Colace, M. De Santo, V. Moscato, A. Picariello, F. A. Schreiber and L. Tanca, Eds. Cham: Springer International Publishing, 2015, pp. 113–133, ISBN: 978-3-319-20062-0 (cit. on pp. 2, 69, 82, 83, 99).

[61] F. Daniel and L. Guida, 'A service-oriented perspective on blockchain smart contracts,' *IEEE Internet Computing*, vol. 23, no. 1, pp. 46–53, 2019 (cit. on p. 14).

[62] A. V. Dastjerdi and R. Buyya, 'Fog computing: Helping the internet of things realize its potential,' *Computer*, vol. 49, no. 8, pp. 112–116, 2016 (cit. on p. 65).

[63] K. A. Delic, 'On resilience of iot systems: The internet of things (ubiquity symposium),' *Ubiquity*, vol. 2016, no. February, pp. 1–7, 2016 (cit. on p. 2).

[64] B. Dorsemaine, J.-P. Gaulier, J.-P. Wary, N. Kheir and P. Urien, 'Internet of things: A definition & taxonomy,' in *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, IEEE, 2015, pp. 72–77 (cit. on p. 1).

[65] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, 'Microservices: Yesterday, today, and tomorrow,' in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67425-4 (cit. on p. 64).

[66] H. Drucker, 'Improving regressors using boosting techniques,' in *ICML*, vol. 97, 1997, pp. 107–115 (cit. on p. 117).

[67] H. Drucker, R. Schapire and P. Simard, 'Boosting performance in neural networks,' in *Advances in Pattern Recognition Systems using Neural Network Technologies*, World Scientific, 1993, pp. 61–75 (cit. on p. 117).

[68] M. Eckert, F. Bry, S. Brodt, O. Poppe and S. Hausmann, 'A cep babelfish: Languages for complex event processing and querying surveyed,' in *Reasoning in Event-Based Distributed Systems*, Springer, 2011, pp. 47–70 (cit. on p. 82).

[69] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow and M. N. Hindia, 'An overview of internet of things (iot) and data analytics in agriculture: Benefits

and challenges,' *IEEE Internet of Things Journal*, vol. 5, no. 5, pp. 3758–3773, 2018 (cit. on pp. 8, 104).

[70] C. Engelmann, G. R. Vallee, T. Naughton and S. L. Scott, 'Proactive fault tolerance using preemptive migration,' in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, IEEE, 2009, pp. 252–257 (cit. on p. 33).

[71] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen and M. Villari, 'Open issues in scheduling microservices in the cloud,' *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016 (cit. on pp. 8, 64).

[72] B. Fekade, T. Maksymyuk, M. Kyryk and M. Jo, 'Probabilistic recovery of incomplete sensed data in iot,' *IEEE Internet of Things Journal*, vol. 5, no. 4, pp. 2282–2292, 2017 (cit. on p. 116).

[73] C. Flaviu, 'Understanding fault-tolerant distributed systems,' *Communications of the ACM*, vol. 34, pp. 56–78, 1993 (cit. on pp. 3, 15, 16, 77, 129).

[74] A. Floris and L. Atzori, 'Quality of experience in the multimedia internet of things: Definition and practical use-cases,' in *2015 IEEE International Conference on Communication Workshop (ICCW)*, IEEE, 2015, pp. 1747–1752 (cit. on p. 1).

[75] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp and M. Mock, 'Issues in complex event processing: Status and prospects in the big data era,' *Journal of Systems and Software*, vol. 127, pp. 217–236, 2017 (cit. on pp. 8, 83).

[76] Ó. Fontenla-Romero, B. Guijarro-Berdiñas, D. Martinez-Rego, B. Pérez-Sánchez and D. Peteiro-Barral, 'Online machine learning,' *Efficiency and Scalability Methods for Computational Intellect*, vol. 27, 2013 (cit. on p. 70).

[77] Y. Freund and R. E. Schapire, 'A desicion-theoretic generalization of on-line learning and an application to boosting,' in *European conference on computational learning theory*, Springer, 1995, pp. 23–37 (cit. on p. 117).

[78] L. J. Fülöp, Á. Beszédes, G. Tóth, H. Demeter, L. Vidács and L. Farkas, 'Predictive complex event processing: A conceptual framework for combining complex event processing and predictive analytics,' in *Proceedings of the Fifth Balkan Conference in Informatics*, ACM, 2012, pp. 26–31 (cit. on p. 93).

[79] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, 'Internet of things: A survey on enabling technologies, protocols, and applications,' *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015 (cit. on p. 67).

[80] D. Galar and U. Kumar, 'Chapter 5 - diagnosis,' in *eMaintenance*, D. Galar and U. Kumar, Eds., Academic Press, 2017, pp. 235–310, ISBN: 978-0-12-811153-6 (cit. on pp. 19, 73).

[81] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy and A. Bouchachia, 'A survey on concept drift adaptation,' *ACM computing surveys (CSUR)*, vol. 46, no. 4, pp. 1–37, 2014 (cit. on p. 152).

[82] V. Garcia, E. Debreuve and M. Barlaud, 'Fast k nearest neighbor search using gpu,' in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, IEEE, 2008, pp. 1–6 (cit. on p. 115).

[83] I. Gashi, P. Popov and L. Strigini, 'Fault tolerance via diversity for off-the-shelf products: A study with sql database servers,' *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 280–294, 2007 (cit. on p. 28).

[84] G. Gediga, K.-C. Hamborg and I. Düntsch, 'Evaluation of software systems,' *Encyclopedia of computer science and technology*, vol. 45, no. supplement 30, pp. 127–53, 2002 (cit. on pp. 111, 112).

[85] A. Gepperth and B. Hammer, 'Incremental learning algorithms and applications,' in *European Symposium on Artificial Neural Networks (ESANN)*, Bruges, Belgium, 2016 (cit. on p. 71).

[86] T. N. Gia, A.-M. Rahmani, T. Westerlund, P. Liljeberg and H. Tenhunen, 'Fault tolerant and scalable iot-based architecture for health monitoring,' in *Sensors Applications Symposium (SAS), 2015 IEEE*, IEEE, 2015, pp. 1–6 (cit. on pp. 4, 41, 56, 58, 76, 155).

[87] S. S. Gokhale, M. R. Lyu and K. S. Trivedi, 'Reliability simulation of fault-tolerant software and systems,' in *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, IEEE, 1997, pp. 167–173 (cit. on p. 27).

[88] A. Gorbenko, V. Kharchenko and A. Romanovsky, 'Using inherent service redundancy and diversity to ensure web services dependability,' in *Methods, Models and Tools for Fault Tolerance*, Springer, 2009, pp. 324–341 (cit. on p. 31).

[89] R. Guerraoui and A. Schiper, 'Software-based replication for fault tolerance,' *Computer*, vol. 30, no. 4, pp. 68–74, 1997 (cit. on p. 24).

[90] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg and G. Anderson, 'Sase: Complex event processing over streams,' *arXiv preprint cs/0612128*, 2006 (cit. on p. 82).

[91] R. Hanmer, *Patterns for Fault Tolerant Software*, ser. Wiley Software Patterns Series. Wiley, 2013, ISBN: 9781118351543 (cit. on pp. 14, 17, 21, 30, 31, 34, 128, 155).

[92] S. Hasan, E. Curry, M. Banduk and S. O'Riain, 'Toward situation awareness for the semantic sensor web: Complex event processing with dynamic linked data enrichment,' in *Proceedings of the 4th International Conference on Se-*

mantic Sensor Networks-Volume 839, CEUR-WS. org, 2011, pp. 69–82 (cit. on p. 85).

[93] I. A. T. Hashem, V. Chang, N. B. Anuar, K. Adewole, I. Yaqoob, A. Gani, E. Ahmed and H. Chiroma, 'The role of big data in smart city,' *International Journal of Information Management*, vol. 36, no. 5, pp. 748–758, 2016 (cit. on p. 5).

[94] W. Hasselbring and G. Steinacker, 'Microservice architectures for scalability, agility and reliability in e-commerce,' in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, 2017, pp. 243–246 (cit. on p. 32).

[95] X. Hu, B. Wang and H. Ji, 'A wireless sensor network-based structural health monitoring system for highway bridges,' *Computer-Aided Civil and Infrastructure Engineering*, vol. 28, no. 3, pp. 193–209, 2013 (cit. on p. 30).

[96] Y.-L. Hu, Y.-Y. Cho, W.-B. Su, D. S. Wei, Y. Huang, J.-L. Chen, I.-Y. Chen and S.-Y. Kuo, 'A programming framework for implementing fault-tolerant mechanism in iot applications,' in *Algorithms and Architectures for Parallel Processing*, G. Wang, A. Zomaya, G. Martinez and K. Li, Eds., Zhangjiajie: Springer, 2015, pp. 771–784 (cit. on pp. 4, 42, 55, 58, 61, 76, 155).

[97] R. Isermann, *Fault-Diagnosis Applications: Model-Based Condition Monitoring: Actuators, Drives, Machinery, Plants, Sensors, and Fault-tolerant Systems*. Springer Berlin Heidelberg, 2011, ISBN: 9783642127670 (cit. on p. 19).

[98] R. Isermann, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Science & Business Media, 2006 (cit. on pp. 16, 77, 88).

[99] A. Javed, K. Heljanko, A. Buda and K. Främling, 'Cefiot: A fault-tolerant iot architecture for edge and cloud,' in *Internet of Things (WF-IoT), 2018 IEEE 4th World Forum on*, IEEE, 2018, pp. 813–818 (cit. on pp. 52, 58).

[100] R. Jhawar, V. Piuri and M. Santambrogio, 'Fault tolerance management in cloud computing: A system-level perspective,' *IEEE Systems Journal*, vol. 7, no. 2, pp. 288–297, 2012 (cit. on p. 14).

[101] F. Kalantari, O. M. Tahir, R. A. Joni and E. Fatemi, 'Opportunities and challenges in sustainability of vertical farming: A review,' *Journal of Landscape Ecology*, vol. 11, no. 1, pp. 35–60, 2018 (cit. on p. 105).

[102] K. Kanoun and M. Ortalo-Borrel, 'Fault-tolerant system dependability-explicit modeling of hardware and software component-interactions,' *IEEE Transactions on reliability*, vol. 49, no. 4, pp. 363–376, 2000 (cit. on p. 20).

[103] S. A. Karthikeya, J. K. Vijeth and C. S. R. Murthy, 'Leveraging solution-specific gateways for cost-effective and fault-tolerant iot networking,' in *2016*

IEEE Wireless Communications and Networking Conference (WCNC), Doha: IEEE, 2016, ISBN: 9781467398145 (cit. on pp. 4, 49, 76, 155).

[104] J. M. Keller, M. R. Gray and J. A. Givens, 'A fuzzy k-nearest neighbor algorithm,' *IEEE transactions on systems, man, and cybernetics*, no. 4, pp. 580–585, 1985 (cit. on p. 115).

[105] J. P. J. Kelly, T. I. McVittie and W. I. Yamamoto, 'Implementing design diversity to achieve fault tolerance,' *IEEE Software*, vol. 8, no. 4, pp. 61–71, Jul. 1991, ISSN: 1937-4194 (cit. on p. 22).

[106] G. N. Khan and G. Wei, 'Fault-tolerant wormhole routing using a variation of the distributed recovery block approach,' *IEE Proceedings-Computers and Digital Techniques*, vol. 147, no. 6, pp. 397–402, 2000 (cit. on p. 23).

[107] R. Khan, S. U. Khan, R. Zaheer and S. Khan, 'Future internet: The internet of things architecture, possible applications and key challenges,' in *2012 10th international conference on frontiers of information technology*, IEEE, 2012, pp. 257–260 (cit. on p. 65).

[108] J. Kiljander, A. D'elia, F. Morandi, P. Hyttinen, J. Takalo-Mattila, A. Ylisaukko-Oja, J.-P. Soininen and T. S. Cinotti, 'Semantic interoperability architecture for pervasive computing and internet of things,' *IEEE access*, vol. 2, pp. 856–873, 2014 (cit. on pp. 9, 67).

[109] J. Kivinen, A. J. Smola and R. C. Williamson, 'Online learning with kernels,' *IEEE Transactions on Signal Processing*, vol. 52, no. 8, pp. 2165–2176, 2004 (cit. on p. 70).

[110] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Elsevier, 2010 (cit. on pp. 17, 19, 89).

[111] T. Kozai, G. Niu and M. Takagaki, *Plant Factory: An Indoor Vertical Farming System for Efficient Quality Food Production*. Elsevier Science, 2019, ISBN: 9780128166925 (cit. on p. 104).

[112] B. Krawczyk, L. L. Minku, J. Gama, J. Stefanowski and M. Woźniak, 'Ensemble learning for data stream analysis: A survey,' *Information Fusion*, vol. 37, pp. 132–156, 2017 (cit. on p. 71).

[113] Kubo, 'The research of iot based on rfid technology,' in *2014 7th International Conference on Intelligent Computation Technology and Automation*, 2014, pp. 832–835 (cit. on p. 1).

[114] V. Kumar and R. Vidhyalakshmi, *Reliability Aspect of Cloud Computing Environment*. Springer Singapore, 2018, ISBN: 9789811330230 (cit. on p. 14).

[115] P. Kumari and P. Kaur, 'A survey of fault tolerance in cloud computing,' *Journal of King Saud University-Computer and Information Sciences*, 2018 (cit. on p. 2).

[116] A. La Marra, F. Martinelli, P. Mori and A. Saracino, 'Implementing usage control in internet of things: A smart home use case,' in *2017 IEEE Trustcom/BigDataSE/ICESS*, IEEE, 2017, pp. 1056–1063 (cit. on p. 4).

[117] ——, 'Implementing usage control in internet of things: A smart home use case,' in *2017 IEEE Trustcom/BigDataSE/ICESS*, IEEE, 2017, pp. 1056–1063 (cit. on p. 76).

[118] I. A. Lakhiar, J. Gao, T. N. Syed, F. A. Chandio and N. A. Buttar, 'Modern plant cultivation technologies in agriculture under controlled environment: A review on aeroponics,' *Journal of plant interactions*, vol. 13, no. 1, pp. 338–352, 2018 (cit. on p. 105).

[119] J.-C. Laprie, 'Dependability—its attributes, impairments and means,' in *Predictably Dependable Computing Systems*, Springer, 1995, pp. 3–18 (cit. on p. 33).

[120] N. Lathia, S. Hailes, L. Capra and X. Amatriain, 'Temporal diversity in recommender systems,' in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, 2010, pp. 210–217 (cit. on p. 28).

[121] I. Lee and K. Lee, 'The internet of things (iot): Applications, investments, and challenges for enterprises,' *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015 (cit. on p. 1).

[122] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, ser. Dependable Computing and Fault-Tolerant Systems. Springer Vienna, 2012, ISBN: 9783709189900 (cit. on pp. 2, 16, 20, 21, 25, 30, 31, 34, 73, 77, 87, 90, 91, 128, 152).

[123] R. de Lemos, P. A. de Castro Guerra and C. M. Rubira, 'A fault-tolerant architectural approach for dependable systems,' *IEEE software*, vol. 23, no. 2, pp. 80–87, 2006 (cit. on pp. 20, 31).

[124] P. R. Lewis, H. Goldingay and V. Nallur, 'It's good to be different: Diversity, heterogeneity, and dynamics in collective systems,' in *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, IEEE, 2014, pp. 84–89 (cit. on p. 28).

[125] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve and Y. Zhou, 'Understanding the propagation of hard errors to software and implications for resilient system design,' *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1, pp. 265–276, 2008 (cit. on p. 19).

[126] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zahl and K. Wehrle, 'Floating-point symbolic execution: A case study in n-version programming,' in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 601–612 (cit. on p. 25).

[127] H. Liu, I. I. Yusuf, H. W. Schmidt and T. Y. Chen, 'Metamorphic fault tolerance: An automated and systematic methodology for fault tolerance in the absence of test oracle,' in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 420–423 (cit. on p. 27).

[128] Y. Liu, Y. Yang, X. Lv and L. Wang, 'A self-learning sensor fault detection framework for industry monitoring iot,' *Mathematical problems in engineering*, vol. 2013, 2013 (cit. on pp. 36, 55, 87).

[129] R. Lu, K. Heung, A. H. Lashkari and A. A. Ghorbani, 'A lightweight privacy-preserving data aggregation scheme for fog computing-enhanced iot,' *IEEE Access*, vol. 5, pp. 3302–3312, 2017 (cit. on p. 51).

[130] D. C. Luckham and B. Frasca, 'Complex event processing in distributed systems,' *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford*, vol. 28, p. 16, 1998 (cit. on p. 82).

[131] M. Luthra and B. Koldehofe, 'Progcep: A programming model for complex event processing over fog infrastructure,' in *Proceedings of the 2nd International Workshop on Distributed Fog Services Design*, 2019, pp. 7–12 (cit. on p. 82).

[132] M. Ma, P. Wang and C.-H. Chu, 'Ltcep: Efficient long-term event processing for internet of things data streams,' in *2015 IEEE International Conference on Data Science and Data Intensive Systems*, IEEE, 2015, pp. 548–555 (cit. on p. 96).

[133] A. I. Maarala, X. Su and J. Riekki, 'Semantic reasoning for context-aware internet of things applications,' *IEEE Internet of Things Journal*, vol. 4, no. 2, pp. 461–473, 2017 (cit. on p. 86).

[134] F. Machida, D. S. Kim and K. S. Trivedi, 'Modeling and analysis of software rejuvenation in a server virtualized system with live vm migration,' *Performance Evaluation*, vol. 70, no. 3, pp. 212–230, 2013 (cit. on p. 33).

[135] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi and A. P. Sheth, 'Machine learning for internet of things data analysis: A survey,' *Digital Communications and Networks*, vol. 4, no. 3, pp. 161–175, 2018 (cit. on p. 3).

[136] R. Mahmud, R. Kotagiri and R. Buyya, 'Fog computing: A taxonomy, survey and future directions,' in *Internet of everything*, Springer, 2018, pp. 103–130 (cit. on p. 65).

[137] A. Malik, B. Aziz, M. Adda and C.-H. Ke, 'Smart routing: Towards proactive fault handling of software-defined networks,' *Computer Networks*, 2020 (cit. on p. 19).

[138] M. Marjani, F. Nasaruddin, A. Gani, A. Karim, I. A. T. Hashem, A. Siddiqa and I. Yaqoob, 'Big iot data analytics: Architecture, opportunities, and open research challenges,' *IEEE Access*, vol. 5, pp. 5247–5261, 2017 (cit. on p. 3).

[139] A. Mdhaffar, R. B. Halima, M. Jmaiel and B. Freisleben, 'Cep4cloud: Complex event processing for self-healing clouds,' in *2014 IEEE 23rd International WETICE Conference*, IEEE, 2014, pp. 62–67 (cit. on p. 82).

[140] A. Medvedev, A. Zaslavsky, M. Indrawan-Santiago, P. D. Haghighi and A. Hassani, 'Storing and indexing iot context for smart city applications,' in *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, Springer, 2016, pp. 115–128 (cit. on p. 4).

[141] Y. Mei and S. Madden, 'Zstream: A cost-based query processor for adaptively detecting composite events,' in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 193–206 (cit. on p. 83).

[142] H. Meling and J. L. Gilje, 'A distributed approach to autonomous fault treatment in spread,' in *2008 Seventh European Dependable Computing Conference*, IEEE, 2008, pp. 46–55 (cit. on p. 34).

[143] Mozilla, *Web thing api*, Accessed: 2018-03-01, 2018. [Online]. Available: `https://iot.mozilla.org/wot` (cit. on p. 67).

[144] M. A. Mukwevho and T. Celik, 'Toward a smart cloud: A review of fault-tolerance methods in cloud systems,' *IEEE Transactions on Services Computing*, 2018 (cit. on pp. 2, 33).

[145] D. Nallaperuma, D. De Silva, D. Alahakoon and X. Yu, 'A cognitive data stream mining technique for context-aware iot systems,' in *Industrial Electronics Society, IECON 2017-43rd Annual Conference of the IEEE*, IEEE, 2017, pp. 4777–4782 (cit. on p. 85).

[146] A. S. Nascimento, C. M. Rubira, R. Burrows, F. Castor and P. H. Brito, 'Designing fault-tolerant soa based on design diversity,' *Journal of Software Engineering Research and Development*, vol. 2, no. 1, p. 13, Nov. 2014 (cit. on p. 23).

[147] R. Natella, D. Cotroneo, J. A. Duraes and H. S. Madeira, 'On fault representativeness of software fault injection,' *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2012 (cit. on p. 19).

[148] R. Natella, D. Cotroneo and H. S. Madeira, 'Assessing dependability with software fault injection: A survey,' *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 44, 2016 (cit. on p. 56).

[149] F. Nawaz, N. K. Janjua and O. K. Hussain, 'Perceptus: Predictive complex event processing and reasoning for iot-enabled supply chain,' *Knowledge-Based Systems*, vol. 180, pp. 133–146, 2019 (cit. on p. 82).

[150] V. P. Nelson, 'Fault-tolerant computing: Fundamental concepts,' *Computer*, vol. 23, no. 7, pp. 19–25, 1990 (cit. on pp. 14, 17, 19).

[151] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox and J. W. Davidson, 'Security through redundant data diversity,' in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, IEEE, 2008, pp. 187–196 (cit. on p. 27).

[152] M. Noura, M. Atiquzzaman and M. Gaedke, 'Interoperability in internet of things: Taxonomies and open challenges,' *Mobile Networks and Applications*, pp. 1–14, 2018 (cit. on pp. 66, 67).

[153] OpenAPI Initiative, *OpenAPI Specification 3.0.2*, Accessed: 2019-03-26, 2017. [Online]. Available: `https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md` (cit. on p. 67).

[154] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun and U. Ramachandran, 'Mcep: A mobility-aware complex event processing system,' *ACM Transactions on Internet Technology (TOIT)*, vol. 14, no. 1, pp. 1–24, 2014 (cit. on p. 82).

[155] K. K. Patel, S. M. Patel *et al.*, 'Internet of things-iot: Definition, characteristics, architecture, enabling technologies, application & future challenges,' *International journal of engineering science and computing*, vol. 6, no. 5, 2016 (cit. on p. 4).

[156] P. K. Patra, H. Singh and G. Singh, 'Fault tolerance techniques and comparative implementation in cloud computing,' *International Journal of Computer Applications*, vol. 64, no. 14, 2013 (cit. on p. 32).

[157] C. Perera, A. Zaslavsky, P. Christen and D. Georgakopoulos, 'Context aware computing for the internet of things: A survey,' *IEEE communications surveys & tutorials*, vol. 16, no. 1, pp. 414–454, 2014 (cit. on p. 85).

[158] A. Polze, P. Troger and F. Salfner, 'Timely virtual machine migration for pro-active fault tolerance,' in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, IEEE, 2011, pp. 234–243 (cit. on p. 33).

[159] D. Powell, J. Arlat, Y. Deswarte and K. Kanoun, 'Tolerance of design faults,' in *Dependable and Historic Computing*, Springer, 2011, pp. 428–452 (cit. on p. 23).

[160] A. Power and G. Kotonya, '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, 2018, pp. 588–599 (cit. on pp. iv, 7).

[161] ——, 'Complex patterns of failure: Fault tolerance via complex event processing for iot systems,' in *2019 International Conference on Internet of*

Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, 2019, pp. 986–993 (cit. on pp. iv, 7, 8).

[162] ——, 'Providing fault tolerance via complex event processing and machine learning for iot systems,' in *Proceedings of the 9th International Conference on the Internet of Things*, Bilbao, Spain: ACM, 2019, 1:1–1:7 (cit. on pp. iv, 7, 8).

[163] L. Pullum, *Software Fault Tolerance Techniques and Implementation*, ser. Artech House computing library. Artech House, 2001, ISBN: 9781580531375 (cit. on pp. 2, 12, 16, 21, 23, 27, 28, 30, 73, 77, 89, 155).

[164] M. O. Rabin and D. Scott, 'Finite automata and their decision problems,' *IBM journal of research and development*, vol. 3, no. 2, pp. 114–125, 1959 (cit. on p. 83).

[165] B. Randell, 'Recovery blocks,' *Encyclopedia of Software Engineering*, 2002 (cit. on p. 23).

[166] L. Rashid, K. Pattabiraman and S. Gopalakrishnan, 'Characterizing the impact of intermittent hardware faults on programs,' *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 297–310, 2014 (cit. on p. 19).

[167] D. Ratasich, F. Khalid, F. Geissler, R. Grosu, M. Shafique and E. Bartocci, 'A roadmap toward the resilient internet of things for cyber-physical systems,' *IEEE Access*, vol. 7, pp. 13 260–13 283, 2019 (cit. on p. 1).

[168] K. F. Rauscher, R. E. Krock and J. P. Runyon, 'Eight ingredients of communications infrastructure: A systematic and comprehensive framework for enhancing network reliability and security,' *Bell Labs Technical Journal*, vol. 11, no. 3, pp. 73–81, 2006 (cit. on pp. 18, 76).

[169] M. A. Razzaque, M. Milojevic-Jevric, A. Palade and S. Clarke, 'Middleware for internet of things: A survey,' *IEEE Internet of things journal*, vol. 3, no. 1, pp. 70–95, 2015 (cit. on p. 1).

[170] Z. Ren, W. Wang, G. Wu, C. Gao, W. Chen, J. Wei and T. Huang, 'Migrating web applications from monolithic structure to microservices architecture,' in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, 2018, pp. 1–10 (cit. on p. 14).

[171] D. Richter, M. Konrad, K. Utecht and A. Polze, 'Highly-available applications on unreliable infrastructure: Microservice architectures in practice,' in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, 2017, pp. 130–137 (cit. on p. 32).

[172] M. Rizwan, A. Nadeem and M. B. Khan, 'An evaluation of software fault tolerance techniques for optimality,' in *2015 International Conference on Emerging Technologies (ICET)*, IEEE, 2015, pp. 1–6 (cit. on p. 28).

[173] R. Salvador, A. Otero, J. Mora, E. de la Torre, L. Sekanina and T. Riesgo, 'Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems,' in *2011 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2011, pp. 164–169 (cit. on p. 32).

[174] 'Random forests,' in *Encyclopedia of Machine Learning*, C. Sammut and G. I. Webb, Eds. Boston, MA: Springer US, 2010, pp. 828–828, ISBN: 978-0-387-30164-8 (cit. on p. 116).

[175] A. M. Sampaio and J. G. Barbosa, 'A comparative cost analysis of fault-tolerance mechanisms for availability on the cloud,' *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 315–323, 2018 (cit. on p. 14).

[176] J. D. dos Santos, A. L. L. da Silva, J. da Luz Costa, G. N. Scheidt, A. C. Novak, E. B. Sydney and C. R. Soccol, 'Development of a vinasse nutritive solution for hydroponics,' *Journal of environmental management*, vol. 114, pp. 8–12, 2013 (cit. on p. 105).

[177] M. Satyanarayanan, 'The emergence of edge computing,' *Computer*, vol. 50, no. 1, pp. 30–39, 2017 (cit. on p. 65).

[178] B. Schroeder and G. Gibson, 'A large-scale study of failures in high-performance computing systems,' *IEEE transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2009 (cit. on p. 14).

[179] N. P. Schultz-Møller, M. Migliavacca and P. Pietzuch, 'Distributed complex event processing with query rewriting,' in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ACM, 2009, p. 4 (cit. on p. 86).

[180] J. Semião, D. Saraiva, C. Leong, A. Romão, M. B. Santos, I. C. Teixeira and J. P. Teixeira, 'Performance sensor for tolerance and predictive detection of delay-faults,' in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, 2014, pp. 110–115 (cit. on p. 32).

[181] O. B. Sezer, E. Dogdu and A. M. Ozbayoglu, 'Context-aware computing, learning, and big data in internet of things: A survey,' *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 1–27, 2017 (cit. on pp. 1, 2).

[182] U. S. Shanthamallu, A. Spanias, C. Tepedelenlioglu and M. Stanley, '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, 2017, pp. 1–8 (cit. on p. 116).

[183] B. Spinnewyn, B. Braem and S. Latre, 'Fault-tolerant application placement in heterogeneous cloud environments,' in *Network and Service Management (CNSM), 2015 11th International Conference on*, IEEE, 2015, pp. 192–200 (cit. on pp. 43, 56, 61).

[184] J. P. Sterbenz, D. Hutchison, E. K. Çetinkaya, A. Jabbar, J. P. Rohrer, M. Schöller and P. Smith, 'Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines,' *Computer Networks*, vol. 54, no. 8, pp. 1245–1265, 2010 (cit. on pp. 2, 14).

[185] L. Strigini, 'Fault tolerance and resilience: Meanings, measures and assessment,' in *Resilience assessment and evaluation of computing systems*, Springer, 2012, pp. 3–24 (cit. on pp. 21, 22).

[186] P. H. Su, C.-S. Shih, J. Y.-J. Hsu, K.-J. Lin and Y.-C. Wang, 'Decentralized fault tolerance mechanism for intelligent iot/m2m middleware,' in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul: IEEE, 2014, pp. 45–50, ISBN: 978-1-4799-3459-1 (cit. on pp. 4, 39, 55, 58, 61, 62, 76, 151, 155).

[187] A. Y. Sun, Z. Zhong, H. Jeong and Q. Yang, 'Building complex event processing capability for intelligent environmental monitoring,' *Environmental modelling & software*, vol. 116, pp. 1–6, 2019 (cit. on p. 82).

[188] L. Sun, Y. Li and R. A. Memon, 'An open iot framework based on microservices architecture,' *China Communications*, vol. 14, no. 2, pp. 154–162, 2017 (cit. on p. 64).

[189] W. Tan, Y. Fan, A. Ghoneim, M. A. Hossain and S. Dustdar, 'From the service-oriented architecture to the web api economy,' *IEEE Internet Computing*, vol. 20, no. 4, pp. 64–68, 2016 (cit. on p. 67).

[190] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007, ISBN: 9780132392273 (cit. on pp. 16, 77).

[191] K. Taylor and L. Leidinger, 'Ontology-driven complex event processing in heterogeneous sensor networks,' in *Extended Semantic Web Conference*, Springer, 2011, pp. 285–299 (cit. on p. 82).

[192] J. Thönes, 'Microservices,' *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015 (cit. on pp. 8, 64).

[193] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel and G. Saake, 'Applying design by contract to feature-oriented programming,' in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2012, pp. 255–269 (cit. on p. 127).

[194] D. Tilman, C. Balzer, J. Hill and B. L. Befort, 'Global food demand and the sustainable intensification of agriculture,' *Proceedings of the national academy of sciences*, vol. 108, no. 50, pp. 20 260–20 264, 2011 (cit. on p. 104).

[195] D. Touliatos, I. C. Dodd and M. McAinsh, 'Vertical farming increases lettuce yield per unit area compared to conventional horizontal hydroponics,' *Food and energy security*, vol. 5, no. 3, pp. 184–191, 2016 (cit. on p. 105).

[196] A. M. Tyrrell, 'Recovery blocks and algorithm-based fault tolerance,' in *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies*, IEEE, 1996, pp. 292–299 (cit. on p. 23).

[197] I. S. Udoh and G. Kotonya, 'Developing iot applications: Challenges and frameworks,' *IET Cyber-Physical Systems: Theory & Applications*, vol. 3, no. 2, pp. 65–72, 2018 (cit. on p. 3).

[198] I. Umesh and G. N. Srinivasan, '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, 2017, pp. 779–787 (cit. on p. 33).

[199] J. C. Vazquez, V. Champac, A. Ziesemer, R. Reis, J. Semião, I. C. Teixeira, M. B. Santos and J. P. Teixeira, 'Predictive error detection by on-line aging monitoring,' in *2010 IEEE 16th International On-Line Testing Symposium*, IEEE, 2010, pp. 9–14 (cit. on p. 32).

[200] C. Verdouw, H. Sundmaeker, B. Tekinerdogan, D. Conzon and T. Montanaro, 'Architecture framework of iot-based food and farm systems: A multiple case study,' *Computers and Electronics in Agriculture*, vol. 165, p. 104 939, 2019 (cit. on p. 104).

[201] S. Verma, Y. Kawamoto, Z. M. Fadlullah, H. Nishiyama and N. Kato, 'A survey on network methodologies for real-time analytics of massive iot data and open research issues,' *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1457–1477, 2017 (cit. on pp. 1, 155).

[202] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano *et al.*, 'Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures,' in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE, 2016, pp. 179–182 (cit. on p. 14).

[203] U. Voges, *Software Diversity in Computerized Control Systems*, ser. Dependable Computing and Fault-Tolerant Systems. Springer Vienna, 2012, ISBN: 9783709189320 (cit. on p. 91).

[204] S. Wasserkrug, A. Gal, O. Etzion and Y. Turchin, 'Efficient processing of uncertain events in rule-based systems,' *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 1, pp. 45–58, 2010 (cit. on p. 83).

[205] Y. Wen, Z. Zhu, Y. Dai, J. Hu and C. Chen, 'The research of intelligent fault diagnosis system based on internet of things,' in *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing*

(CPSCom) and IEEE Smart Data (SmartData), IEEE, 2019, pp. 1120–1124 (cit. on p. 19).

[206] D. Wettschereck, D. W. Aha and T. Mohri, 'A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms,' *Artificial Intelligence Review*, vol. 11, no. 1-5, pp. 273–314, 1997 (cit. on p. 116).

[207] G. White, V. Nallur and S. Clarke, 'Quality of service approaches in iot: A systematic mapping,' *Journal of Systems and Software*, vol. 132, pp. 186–203, 2017 (cit. on p. 2).

[208] M. W. Woo, J. Lee and K. Park, 'A reliable iot system for personal healthcare devices,' *Future Generation Computer Systems*, vol. 78, pp. 626–640, 2018 (cit. on pp. 4, 53–55, 61, 76, 155).

[209] D. Wu, D. I. Arkhipov, E. Asmare, Z. Qin and J. A. McCann, 'Ubiflow: Mobility management in urban-scale software defined iot,' in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, IEEE, 2015, pp. 208–216 (cit. on p. 44).

[210] M. Wu, T.-J. Lu, F.-Y. Ling, J. Sun and H.-Y. Du, 'Research on the architecture of internet of things,' in *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, IEEE, vol. 5, 2010, pp. V5–484 (cit. on p. 65).

[211] Y. Xie, 'Escep: A cep based on event sharing in internet of things,' in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, IEEE, 2017, pp. 187–190 (cit. on p. 97).

[212] J. Xu and B. Randell, 'Software fault tolerance: T/(n-1)-variant programming,' *IEEE Transactions on Reliability*, vol. 46, no. 1, pp. 60–68, 1997 (cit. on p. 25).

[213] X. Xu and M.-L. Li, 'Understanding soft error propagation using efficient vulnerability-driven fault injection,' in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, IEEE, 2012, pp. 1–12 (cit. on p. 19).

[214] I. Yaqoob, S. A. Hussain, S. Mamoon, N. Naseer, J. Akram and A. ur Rehman, 'Penetration testing and vulnerability assessment,' *Journal of Network Communications and Emerging Technologies (JNCET) www. jncet. org*, vol. 7, no. 8, 2017 (cit. on p. 17).

[215] H. Zhang, Y. Diao and N. Immerman, 'On complexity and optimization of expensive queries in complex event processing,' in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 217–228 (cit. on p. 84).

[216] W. Zhang, D. Xu, P. N. Enjeti, H. Li, J. T. Hawke and H. S. Krishnamoorthy, 'Survey on fault-tolerant techniques for power electronic converters,' *IEEE*

Transactions on Power Electronics, vol. 29, no. 12, pp. 6319–6331, 2014 (cit. on p. 31).

[217] Z. Zhang, 'Microsoft kinect sensor and its effect,' *IEEE MultiMedia*, vol. 19, pp. 4–12, Apr. 2012 (cit. on p. 46).

[218] J. Zhao, Y. Wang, G. Ning, K. S. Trivedi, R. Matias Jr and K.-Y. Cai, 'A comprehensive approach to optimal software rejuvenation,' *Performance Evaluation*, vol. 70, no. 11, pp. 917–933, 2013 (cit. on p. 34).

[219] Z. Zheng, M. R. T. Lyu and H. Wang, 'Service fault tolerance for highly reliable service-oriented systems: An overview,' *Science China Information Sciences*, vol. 58, no. 5, pp. 1–12, 2015 (cit. on p. 128).

[220] S. Zhou, K.-J. Lin, J. Na, C.-C. Chuang and C.-S. Shih, 'Supporting service adaptation in fault tolerant internet of things,' in *2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE, 2015, pp. 65–72 (cit. on pp. 45, 48, 56, 61, 62).

Appendix A
Code Listings

In this appendix are code listings of the Python code that was used in the BoboCEP implementation, as described in Section 6.1.

A.1 Shared Versioned Match Buffer

The *Shared Versioned Match Buffer* (SVMB) was proposed by Agrawal et al. [4] to ensure only one version of an event is stored, even if it is used by multiple runs. The BoboCEP implementation of SVMB closely resembles the original algorithm, which is described in more detail in Section 6.1.2. The key methods from the Python implementation of SVMB are shown in Listing A.1, with comments in the code to aid the understanding of the SVMB process.

Listing A.1: The BoboCEP SVMB implementation in Python.

```
1  class SharedVersionedMatchBuffer:
2      """
3      The buffer stores events for partially completed runs.
4      A BoboEvent instance is stored within a MatchEvent instance
5      that provides a means of linking BoboEvents that can be used
6      by one or more runs.
7      The next event in a MatchEvent link points to an earlier
8      event accepted by a run.
9      Traversing the next links will identify all events for
10     a given run.
11     """
12
13     def __init__(self) -> None:
14         super().__init__()
15
16         # BoboNFA Name -> BoboState Label ->
17         # BoboEvent ID -> MatchEvent
18         self._eve = {}
19
20         # BoboNFA Name -> BoboRun ID ->
21         # RunVersion -> Last MatchEvent
22         self._ver = {}
23
24     @staticmethod
25     def _get_or_create_subdict(d: dict, key: str) -> dict:
26         """
27         Gets a nested dict inside an existing dict, or creates
```

```
28          one there if one does not exist.
29
30          :param d: The dict in which to create a dict.
31
32          :param key: The key that will point to the new dict.
33
34          :return: The nested dict.
35          """
36          if key not in d:
37              d[key] = {}
38
39          return d.get(key)
40
41      def get_event(self,
42                    nfa_name: str,
43                    state_label: str,
44                    event_id: str,
45                    default=None) -> BoboEvent:
46          """
47          Get a BoboEvent instance from the buffer.
48
49          :param nfa_name: The NFA name.
50
51          :param state_label: The state label.
52
53          :param event_id: The event ID.
54
55          :param default: The default value, defaults to None.
56
57          :return: The BoboEvent instance,
58                   or default value if an event is not found.
59          """
60
61          try:
62              return self._eve[nfa_name][state_label][event_id].event
63
64          except KeyError:
65              return default
66
67      def put_event(self,
68                    nfa_name: str,
69                    run_id: str,
70                    version: str,
71                    state_label: str,
72                    event: BoboEvent,
73                    new_run_id: str = None,
74                    new_version: str = None) -> MatchEvent:
75          """
76          Puts a BoboEvent instance into the buffer.
77
78          :param nfa_name: The BoboNFA instance name.
79
80          :param run_id: The run ID with which to associate
81                         the event.
82
83          :param version: The run version of the run with which
```

```
84                          to associate the event.
85
86          :param state_label: The label of the state.
87
88          :param event: The event to add to the buffer.
89
90          :param new_run_id: The new run ID with which to associate
91                             the event, so that the other run ID
92                             will point to this one, defaults to None.
93
94          :param new_version: The new run version with which to
95                              associate the event, so that the other
96                              version will point to this one,
97                              defaults to None.
98
99          :return: A MatchEvent instance containing the BoboEvent
100                  instance.
101         """
102
103         # get match events for this nfa
104         # keyed by the bobo event they represent
105         nfa_labels = self._get_or_create_subdict(
106             self._eve, nfa_name)
107         nfa_events = self._get_or_create_subdict(
108             nfa_labels, state_label)
109
110         # look for bobo event in buffer
111         # or add a new match event for it
112         # if a match event is not found
113         if event.event_id not in nfa_events:
114             nfa_events[event.event_id] = MatchEvent(
115                 nfa_name=nfa_name,
116                 label=state_label,
117                 event=event)
118
119         new_match_event = nfa_events[event.event_id]
120
121         # get last event under the original run ID and version
122         # before (maybe) updating run ID and version
123         last_match_event = self.get_last_event(
124             nfa_name=nfa_name,
125             run_id=run_id,
126             version=version)
127
128         if new_run_id is not None:
129             run_id = new_run_id
130
131         if new_version is not None:
132             version = new_version
133
134         # point the (maybe new) run ID and version
135         # to the next match event
136         nfa_runs = self._get_or_create_subdict(self._ver, nfa_name)
137         run_versions = self._get_or_create_subdict(nfa_runs, run_id)
138         run_versions[version] = new_match_event
139
```

```
140             # point new match event to the last match event,
141             # if it exists
142             if last_match_event is not None:
143                 # points to the last match event
144                 new_match_event.add_pointer_next(
145                     version=version,
146                     label=last_match_event.label,
147                     event_id=last_match_event.event.event_id)
148 
149                 # points backwards to the new match event
150                 last_match_event.add_pointer_previous(
151                     version=version,
152                     label=new_match_event.label,
153                     event_id=new_match_event.event.event_id)
154             else:
155                 # adds pointer to nothing, so run is still
156                 # linked to event
157                 new_match_event.add_pointer_next(version=version)
158 
159             return new_match_event
160 
161         def remove_version(self, nfa_name: str, version: str) -> None:
162             """
163             Removes a run version from all of the match events
164             in the buffer.
165 
166             :param nfa_name: The name of the BoboNFA instance.
167 
168             :param version: The run version.
169             """
170 
171             # remove version from runs
172             nfa_runs = self._ver.get(nfa_name)
173 
174             if nfa_runs is not None:
175                 for nfa_run in tuple(nfa_runs.values()):
176                     for run_version in tuple(nfa_run.keys()):
177                         if version == run_version:
178                             nfa_runs.pop(version, None)
179 
180             # remove pointers in events, and maybe event itself
181             nfa_labels = self._eve.get(nfa_name)
182 
183             if nfa_labels is not None:
184                 for match_events in tuple(nfa_labels.values()):
185                     for match_event in tuple(match_events.values()):
186                         match_event.remove_all_pointers(version)
187 
188                         # drop more pointers, drop event
189                         if not match_event.has_pointers():
190                             self._eve \
191                                 .get(nfa_name) \
192                                 .get(match_event.label) \
193                                 .pop(match_event.event.event_id, None)
194 
195         def get_last_event(self,
```

```
196                         nfa_name: str,
197                         run_id: str,
198                         version: str,
199                         default=None) -> MatchEvent:
200         """
201         Gets the last match event.
202 
203         :param nfa_name: The BoboNFA instance name.
204 
205         :param run_id: The run ID.
206 
207         :param version: The run version.
208 
209         :param default: A value to return if no match event
210                         is found, defaults to None.
211 
212         :return: The last match event,
213                  or a default value if one does not exist.
214         """
215 
216         try:
217             return self._ver[nfa_name][run_id][version]
218 
219         except KeyError:
220             return default
221 
222     def get_all_events(self,
223                        nfa_name: str,
224                        run_id: str,
225                        version: RunVersion) -> BoboHistory:
226         """
227         Gets all events associated with a run and compiles
228         them into a BoboHistory instance.
229 
230         :param nfa_name: The BoboNFA instance name.
231 
232         :param run_id: The run ID.
233 
234         :param version: The run version.
235 
236         :return: A BoboHistory instance with all of the
237                  events in it.
238         """
239 
240         all_events = {}
241         current_level = 0
242         current_incr = 0
243         current_version = version.get_version_as_str()
244 
245         # start with the latest match event
246         current_event = self.get_last_event(
247             nfa_name=nfa_name,
248             run_id=run_id,
249             version=current_version)
250 
251         while True:
```

```
252             if current_event is not None:
253                 # add event to dict, keyed under the label name
254                 if current_event.label not in all_events:
255                     all_events[current_event.label] = []
256                 all_events[current_event.label].insert(
257                     0, current_event.event)
258 
259                 # get next match event using current version
260                 next_event = self._get_next_event(
261                     event=current_event,
262                     nfa_name=nfa_name,
263                     version_str=current_version)
264 
265                 # no event found under current version
266                 if next_event is None:
267                     # get previous version by decreasing increment
268                     current_incr += 1
269                     current_version = \
270                         version.get_previous_version_as_str(
271                             decrease_level=current_level,
272                             decrease_incr=current_incr)
273 
274                     # get previous version by decreasing level
275                     if current_version is None:
276                         current_level += 1
277                         current_incr = 0
278                         current_version = \
279                             version.get_previous_version_as_str(
280                                 decrease_level=current_level,
281                                 decrease_incr=current_incr)
282 
283                         # no previous version, stop search
284                         if current_version is None:
285                             break
286 
287                     # attempt to find next event with new version
288                     next_event = self._get_next_event(
289                         event=current_event,
290                         nfa_name=nfa_name,
291                         version_str=current_version)
292 
293                     if next_event is None:
294                         break
295 
296                 current_event = next_event
297             else:
298                 break
299 
300         return BoboHistory(events=all_events)
301 
302     def _get_next_event(self,
303                         event: MatchEvent,
304                         nfa_name: str,
305                         version_str: str):
306         try:
307             next_tuple = event.next_ids.get(version_str)
```

```
308             if next_tuple is not None:
309                 next_event = self._eve[
310                     nfa_name][next_tuple[0]][next_tuple[1]]
311             else:
312                 next_event = None
313         except KeyError:
314             next_event = None
315 
316         return next_event
```

A.2 Match Event

In order for SVMB to efficiently store and link its events, they are each individually stored in a *match event* that provides additional pointers to other events in the buffer. Listing A.2 shows the Python implementation of match events in BoboCEP.

Listing A.2: The BoboCEP match event implementation in Python.

```
1  class MatchEvent:
2      """
3      A BoboEvent instance that has been selected as a match for some
4      state criteria by one or more runs.
5  
6      :param nfa_name: The name of the BoboNFA instance.
7  
8      :param label: The state label with which the event is
9                    associated.
10 
11     :param event: The event selected as being a match for some
12                   state.
13 
14     :param next_ids: Matches from older states, to which this event
15                      links, defaults to an empty dict.
16 
17     :param prev_ids: Matches from newer states, to which this
18                      event links, default to an empty dict.
19     """
20 
21     def __init__(
22         self,
23         nfa_name: str,
24         label: str,
25         event: BoboEvent,
26         next_ids: Dict[str, Tuple[str, str]] = None,
27         prev_ids: Dict[str, Tuple[str, str]] = None) -> None:
28         super().__init__()
29 
30         self.nfa_name = nfa_name
31         self.label = label
32         self.event = event
33         self.next_ids = {} if next_ids is None else next_ids
34         self.prev_ids = {} if prev_ids is None else prev_ids
35 
36     def add_pointer_next(
37         self,
```

```
        version: str,
        label: str = "",
        event_id: str = "") -> None:
        """
        Points match event to the next match event ID using a
        run version.

        :param version: The run version.

        :param label: The state label, defaults to an empty string.

        :param event_id: The event ID to point to, defaults to an
                         empty string.
        """

        # Cannot point match event to itself.
        if label == self.label and event_id == self.event.event_id:
            raise RuntimeError()

        self.next_ids[version] = (label, event_id)

    def add_pointer_previous(
        self,
        version: str,
        label: str = "",
        event_id: str = "") -> None:
        """
        Points match event to the previous match event ID using
        a run version.

        :param version: The run version.

        :param label: The state label, defaults to an empty string.

        :param event_id: The event ID to point to, defaults to an
                         empty string.
        """

        # Cannot point match event to itself.
        if label == self.label and event_id == self.event.event_id:
            raise RuntimeError()

        self.prev_ids[version] = (label, event_id)

    def remove_all_pointers(self, version: str) -> None:
        """
        Removes all pointers to a match event with a given
        run version.

        :param version: The run version.
        """

        self.next_ids.pop(version, None)
        self.prev_ids.pop(version, None)

    def has_pointers(self) -> bool:
```

```
94          """
95          :return: True if the match event points to any other
96                   match events, False otherwise.
97          """
98
99          return len(self.next_ids) > 0 or len(self.prev_ids) > 0
```

A.3 Run Version

As demonstrated in Section 6.1.2, each run has a unique *run version* number so that its events can be identified within a buffer. A match event uses run versions as keys in order to link a run's events together. Listing A.3 shows the Python implementation of this in BoboCEP.

Listing A.3: The BoboCEP run version implementation in Python.

```
1  class RunVersion:
2      """
3      The current version of a run.
4
5      :param parent_version: The parent version. If provided,
6                             the new version will copy the parent
7                             version's levels. Defaults to None.
8      """
9
10     def __init__(self, parent_version: 'RunVersion' = None) -> None:
11         super().__init__()
12
13         self._levels = [] if parent_version is None \
14             else copy(parent_version._levels)
15
16     @staticmethod
17     def list_to_version(version_list: List[str]) -> 'RunVersion':
18         """
19         Converts a list of strings into a RunVersion instance,
20         where each string of the list becomes a level in the
21         RunVersion instance, in list order.
22
23         :param version_list: A list of strings.
24
25         :return: A new RunVersion instance.
26         """
27
28         run = RunVersion()
29         run._levels = version_list
30         return run
31
32     @staticmethod
33     def list_to_version_str(str_list: List[str]) -> str:
34         """
35         Converts a list of strings into a valid version string
36         by joining each string in the list by the delimeter ".".
37
38         :param str_list: A list of strings.
```

```
39
40          :return: A valid version string.
41          """
42
43          return RunVersion._DELIMITER.join(str_list)
44
45      @staticmethod
46      def str_to_version(version_str: str) -> 'RunVersion':
47          """
48          Converts a version string into a RunVersion instance
49          by splitting the string by the delimiter "." and making
50          each split a level in the new instance.
51
52          :param version_str: The version string.
53
54          :return: A new RunVersion instance.
55          """
56
57          run = RunVersion()
58          run._levels = version_str.split(RunVersion._DELIMITER)
59          return run
60
61      def add_level(self, level: str) -> None:
62          """
63          Adds a new level to the version.
64
65          :param level: The level name.
66          """
67
68          self._levels.append([level])
69
70      def increment_level(self, increment: str) -> None:
71          """
72          Increments the current level.
73
74          :param increment: The increment name.
75
76          :raises RuntimeError: Attempting to increment when
77                                there are no levels.
78          """
79
80          if len(self._levels) == 0:
81              raise RuntimeError()
82
83          self._levels[-1].append(increment)
84
85      def size(self) -> int:
86          """
87          :return: The number of levels.
88          """
89
90          return len(self._levels)
91
92      def size_level(self, index: int = -1):
93          """
94          :param index: The index of the level, defaults to the
```

```
95                          latest level.
96
97          :return: The number of increments in at a given level.
98          """
99
100         return len(self._levels[index])
101
102     def remove_all_levels(self) -> None:
103         """
104         Removes all levels from the version.
105         """
106
107         self._levels = []
108
109     def get_version_as_list(self) -> List[str]:
110         """
111         :return: The latest version as a list of strings, where
112                  each string is the latest increment of the
113                  given level.
114         """
115
116         return [level[-1] for level in self._levels]
117
118     def get_version_as_str(self) -> str:
119         """
120         :return: The latest version as a string, where each level
121                  is the latest increment of the given level,
122                  separated with the delimeter ".", e.g. "a.b.c".
123                  If there are no levels, it returns "".
124         """
125
126         if len(self._levels) == 0:
127             return ""
128
129         return RunVersion._DELIMITER.join(
130             [level[-1] for level in self._levels])
131
132     def get_previous_version_as_list(
133         self,
134         decrease_level: int,
135         decrease_incr: int) -> List[str]:
136         """
137         Gets a previous version of the run version as a list
138         of strings.
139
140         :param decrease_level: How many levels to decrease by.
141                                For example, if there are 5 levels,
142                                a decrease_level of 2 will go back
143                                to level 3.
144
145         :param decrease_incr: How many increments of a level to
146                               decrease by.
147
148         :raises RuntimeError: Decreasing beyond the number
149                               of levels.
150         :raises RuntimeError: Decreasing beyond the number
```

```
151                                 of increments in the chosen level.
152
153             :return: The previous version as a list.
154             """
155
156             index_level = len(self._levels) - decrease_level - 1
157
158             if not (0 <= index_level < len(self._levels)):
159                 raise RuntimeError()
160
161             index_incr = len(self._levels[
162                 index_level]) - decrease_incr - 1
163
164             if not (0 <= index_incr < len(self._levels[index_level])):
165                 raise RuntimeError()
166
167             version = [level[-1] for level in self._levels[
168                 :index_level]]
169             version.append(self._levels[index_level][index_incr])
170
171             return version
172
173         def get_previous_version_as_str(self,
174                                         decrease_level: int,
175                                         decrease_incr: int,
176                                         default=None) -> str:
177             """
178             Gets a previous version of the run version as a string,
179             delimited by ".".
180
181             :param decrease_level: How many levels to decrease by.
182                                    For example, if there are 5 levels,
183                                    a decrease_level of 2 will go back
184                                    to level 3.
185
186             :param decrease_incr: How many increments of a level to
187                                   decrease by.
188
189             :param default: The default value to return if a version
190                             string fails to be generated,
191                             defaults to None.
192
193             :return: The previous version as a string delimited by ".".
194                      If there are no levels, it returns an empty string.
195             """
196
197             if len(self._levels) == 0:
198                 return ""
199
200             try:
201                 return RunVersion._DELIMITER.join(
202                     self.get_previous_version_as_list(decrease_level,
203                                                       decrease_incr))
204             except RuntimeError:
205                 return default
```

Appendix B
Testbed Design

This appendix describes the design of the medium-scale VFS from Section 6.2.2 in more depth. It explores how the smart plugs were modified and interacted with via BoboCEP, the setup of the water pumps and grow lights, as well as a more detailed insight into the design of the microcontrollers which generated the sensor data for the VFS. The medium-scale testbed was an upgrade of the small-scale VFS from Section 6.2.1, which was of a comparable design.

B.1 Smart Plugs

B.1.1 KanKun KK-SP3

The KanKun KK-SP3 was a smart plug that featured an Atheros AR9330 (400MHz) processor, 4MB flash memory, 32 MB RAM, and ran on an OpenWRT[1] Linux distribution (Figure B.1a). The plugs had very weak security and the operating system could be accessed directly via SSH, which meant that the official app used to control the plugs could be circumvented completely. This meant that custom *Common Gateway Interface* (CGI) scripts could be installed onto the plugs to enable direct control of them via HTTP.

Four scripts were written for the plugs: (1) to turn the plug `ON`; (2) to turn the plug

[1]https://openwrt.org

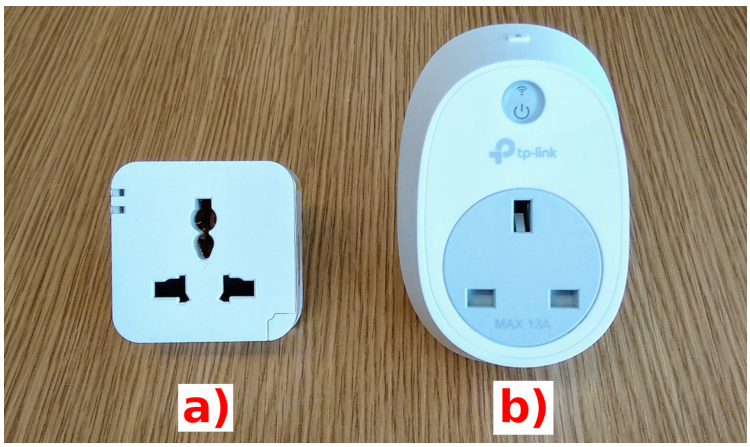

Figure B.1: The (a) KanKun KK-SP3 and (b) TP-Link HS100 smart plugs.

OFF; (3) to get the current state of the plug; and (4) a timer to turn the plug ON for t seconds and then OFF, which is shown in Listing B.1. The timer function was an important script that was used to control the water pumps specifically. The script guaranteed that the water pumps turned off after t seconds, rather than relying on BoboCEP instances to send an OFF signal after an ON signal, which might have failed to send and might have caused a water pump to flood its connected water reservoir.

Listing B.1: The CGI script for the KanKun KK-SP3 timer feature.

```
#!/bin/sh

echo "Content-Type: text/plain"
echo "Cache-Control: no-cache, must-revalidate"
echo "Expires: Sat, 26 Jul 1997 05:00:00 GMT"
echo

RELAY_CTRL=/sys/class/leds/tp-link:blue:relay/brightness

case "$QUERY_STRING" in
        seconds=*)
        SECONDS=${QUERY_STRING:8}

        case $SECONDS in
            ''|*[!0-9]*)
                echo ERR
            ;;
            *)
                (echo 1 > $RELAY_CTRL &&
                sleep $SECONDS &&
                echo 0 > $RELAY_CTRL) &
                echo OK
            ;;
        esac
        ;;
        *)
        echo ERR
esac
```

B.1.2 TP-Link HS100

The TP-Link HS100 was a smart plug from TP-Link's smart home range (Figure B.1b). They were purchased when the KanKun KK-SP3 smart plugs had become obsolete, however more smart plugs were needed during the upgrade from the small- to medium-scale VFS. When setting up these plugs, they were first registered to a TP-Link personal account and were delegated unique device IDs after registration.

BoboCEP instances were able to authenticate with the TP-Link servers by sending custom JSON data with the account credentials by using an account token that provided access to the smart plugs registered to the account. The BoboCEP instances running alongside the VFS were loaded with a class that enabled access to the smart plugs using this approach, shown in Listing B.2. This code generated a new token every time it attempted to change the plugs to either their ON or OFF states.

Listing B.2: A simplified version of how BoboCEP performed actions using the TP-Link HS100 smart plugs.

```
class TPLinkHS100Action(BoboAction):
    ADDR_WAP = "https://wap.tplinkcloud.com"
    ADDR_EU_WAP = "https://eu-wap.tplinkcloud.com"
    ERROR_CODE = "error_code"
    RESULT = "result"
    TOKEN = "token"

    def __init__(self,
                 name: str,
                 email: str,
                 password: str,
                 device_id: str,
                 on: bool) -> None:
        super().__init__(name=name)

        # Account credentials to access the plug.
        self.email = email
        self.password = password

        # Which plug to control.
        self.device_id = device_id

        # Whether to turn the plug ON or OFF.
        self.on = on

    def _perform_action(self, event: BoboEvent) -> bool:
        # Fetch a new token before performing the desired action.
        token = TPLinkHS100Action.fetch_token(
            self.email, self.password)

        # If a valid token was returned.
        if len(token) > 0:
            # Build a path with the token included.
            url = "{}?token={}".format(
                TPLinkHS100Action.ADDR_EU_WAP, token)

            req = urllib.request.Request(url)
            req.add_header('Content-Type', 'application/json')

            # Send ON/OFF request in JSON format.
            json_body_bytes = json.dumps({
                'method': 'passthrough',
                'params': {
                    'deviceId': self.device_id,
                    'requestData':
                        '{{\"system\":{{\"set_relay_state\":'
                        '{{\"state\":{'
                        '}}}}}}}'.format('1' if self.on else '0')
                }
            }).encode('utf-8')

            req.add_header('Content-Length', len(json_body_bytes))
            response = json.loads(
                urllib.request.urlopen(req, json_body_bytes).read())
```

```
55
56              return int(response['error_code']) == 0
57          return False
58
59      @staticmethod
60      def fetch_token(email: str, password: str) -> str:
61          req = Request(TPLinkHS100Action.ADDR_WAP)
62
63          data = {
64              "method": "login",
65              "params": {
66                  "appType": "Kasa_Android",
67                  "cloudUserName": email,
68                  "cloudPassword": password,
69                  "terminalUUID": str(uuid4())
70              }
71          }
72
73          json_bytes = dumps(data).encode('utf-8')
74
75          req.add_header('Content-Type', 'application/json')
76          req.add_header('Content-Length', len(json_bytes))
77
78          res = urlopen(req, json_bytes)
79          code = res.getcode()
80
81          if code == 200:
82              json_res = loads(res.read().decode())
83
84              if json_res[TPLinkHS100Action.ERROR_CODE] == 0:
85                  return json_res[TPLinkHS100Action.RESULT][
86                      TPLinkHS100Action.TOKEN]
87
88          return ""
```

B.2 Actuators

The two actuators that enabled the VFS to provide service were the water pumps and the grow lights. Both of these devices could not be controlled directly, and their `ON` and `OFF` states were instead controlled by the smart plugs to which they connected. The water pumps were connected to the KanKun KK-SP3 plugs so that they could use the timer function from Listing B.1, which ensured that they switched `OFF` when activated. The grow lights were connected to the TP-Link HS100 plugs.

The grow lights themselves could be programmed to switch `OFF` after t hours. However, BoboCEP provided this functionality instead, and was a more adaptive solution because it used light intensity data to determine the optimal time to activate and deactivate the grow lights. Figure B.2 (left) shows the grow lights activating as natural light began to deplete, which ensured constant photosynthesis for the plants.

Similarly, the water pumps were activated when the reservoir began to deplete its supply of water. Each water pump would pump water through its connected plastic

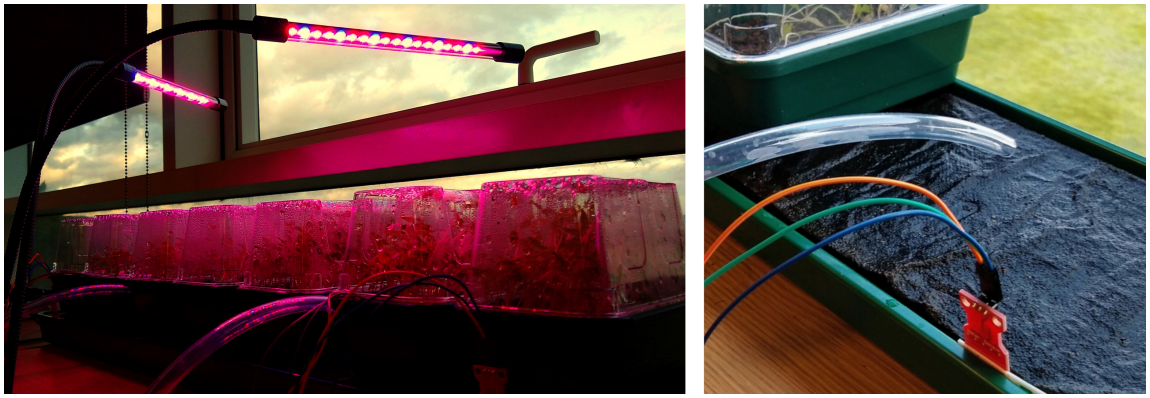

Figure B.2: *Left*: grow lights activated when natural light decreased in the evening. *Right*: plastic tubing connecting a water pump to its reservoir.

tubing, which poured water directly onto the capillary matting and trickled down into the reservoir (Figure B.2, right). By pumping water onto the matting, it ensured that the matting did not dry out because, if it did, it would hamper the ability for it to draw water from the reservoir below.

B.3 Microcontrollers

The microcontrollers were the means by which the medium-scale VFS could be monitored and decisions could be made as to whether the system was behaving erroneously. In each water reservoir were two water-level sensors (Figure B.3ai,ii). These were held into place using double-sided adhesive tape that ensured the bottom of the sensor touched the bottom of the reservoir, and therefore a low reading represented nearly-depleted water supplies. The water-level sensors would then influence the water pumps associated with the reservoir being monitored, and would prompt BoboCEP to activate the primary pump, or the redundant pump if the primary pump failed to activate.

The LDRs were positioned on the shelves such that they were able to detect natural light in the day, but were also close enough to the grow lights that they could detect whether the lights were `ON` or `OFF` (Figure B.3b). This was important data that influenced the experiment in Section 7.4.4 which ensured correct grow light functionality. The temperature/humidity sensors were placed into plant pots to sample two pots per shelf (Figure B.3c). The pots had plastic lids on them, which helped the plants to retain water and reduce the amount of water that the system needed to consume. The data from these sensors were important in providing another means of inferring the health of the produce, because a higher humidity suggested that the plants were well watered.

Each microcontroller was configured as shown in Figure B.4. This meant that, for each shelf, there were two water-level sensors per reservoir, two LDRs per shelf, and two temperature/humidity sensors in two plant pots on each shelf. This level of redundancy provided more certainty that the data being collected by each sensor were

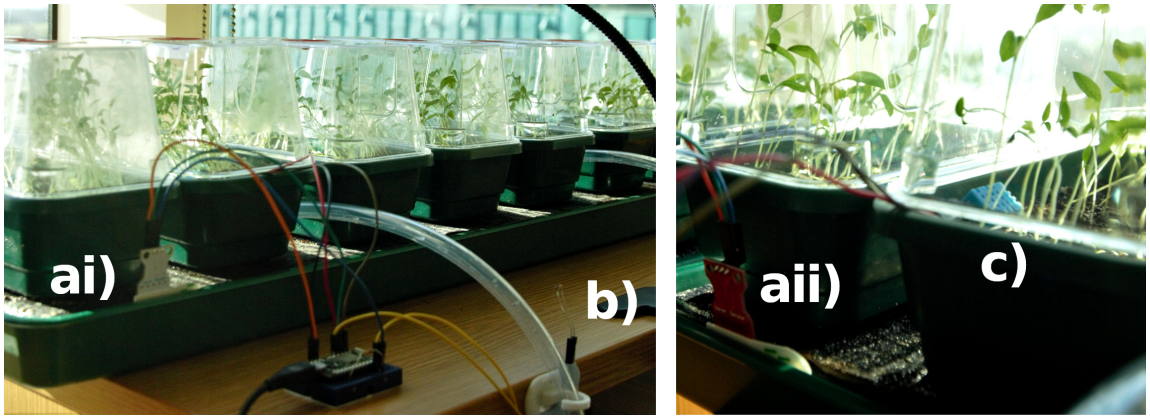

Figure B.3: A microcontroller connected to a reservoir, plant pot, and shelf.

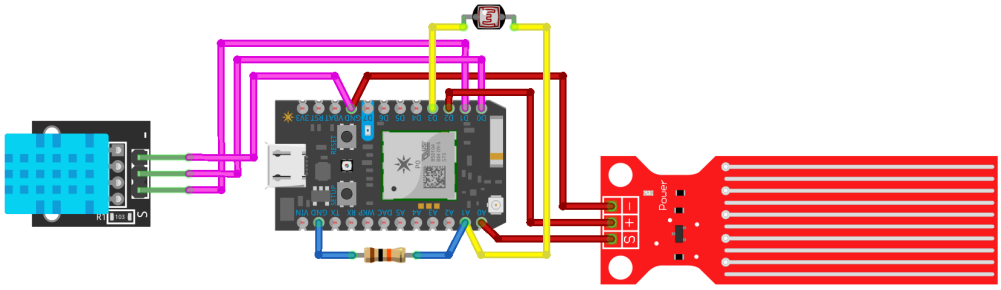

Figure B.4: The configuration of the microcontrollers.

reasonable, because a direct comparison between the data of two sensors collecting the same data from the same shelf/reservoir could be performed.

A key feature of BoboCEP was its ability to provide load balancing due to the activate replication of its state, as discussed in Sections 6.1 & 7.5.1. The microcontrollers helped to demonstrate this feature with a simple protocol whereby, if a microcontroller failed to send data to one edge device, it would randomly pick another edge device through which to attempt to send future data. This implementation is shown in Listing B.3. The three entry points represent the three edge devices used during the evaluation (Section 7.5), each of which sent data to BoboCEP using the **/properties** interface described in Section 4.1.

Listing B.3: A simplified version of the C++ code on the microcontrollers.

```
1  class EntryPoint {
2    public:
3      String addr;
4      int port;
5      String path;
6
7      EntryPoint(String, int, String);
8  };
9
10 EntryPoint::EntryPoint (String _addr, int _port, String _path) {
11     addr = _addr;
12     port = _port;
```

```
13      path = _path;
14  }
15
16  const EntryPoint ENTRY_POINTS[] = {
17      EntryPoint("192.168.1.123", 5050, "/properties"),
18      EntryPoint("192.168.1.124", 5050, "/properties"),
19      EntryPoint("192.168.1.125", 5050, "/properties")
20  };
21
22  int currentEntryPoint;
23
24  void setup() {
25      currentEntryPoint = 0;
26  }
27
28  void loop() {
29      if(!send_data(get_sensor_data_as_json(), currentEntryPoint))
30          set_new_entry_point();
31
32      delay(3000); // 3 seconds
33  }
```