BoboCEP: Distributed Complex Event Processing for Resilient Fault-Tolerance Support in IoT

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

***Abstract*—Providing effective fault-tolerance (FT) support for Internet of Things (IoT) systems is hampered by the many ad hoc ways that it is implemented. We propose BoboCEP, a Complex Event Processing (CEP) system that provides resilient FT support for IoT systems, where errors are defined as nondeterministic finite automata. BoboCEP is designed to be distributed at the network edge, which facilitates resilient event processing and load balancing due to the active replication of FT support across the edge. We evaluated BoboCEP on a vertical farming use case to demonstrate long-term FT support and load balancing, and stress tested it under scenarios with high data throughput.**

***Index Terms*—internet of things, complex event processing, fault tolerance, distributed systems, edge computing**

I. Introduction

The *Internet of Things* (IoT) has caused a surge in the number of interconnected devices and is growing at an exponential rate. This growth has, in turn, contributed to an explosion in data that requires real-time analytics in order to extract business value [1]. However, IoT system *dependability* is threatened by faults and errors that can cause failures and data loss[1]. *Fault tolerance* (FT) offers a way to address this.

Current FT solutions in IoT are designed in an ad hoc and application-specific manner that can only handle a subset of failure scenarios, and are often situated on non-resilient platforms. For example, they are designed to work with specific hardware and protocols [3], bespoke architectures [4], and FT support itself is often a *single point of failure* (SPoF) [5]. What is required is 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.

We propose BoboCEP, a *Complex Event Processing* (CEP) system that can detect, assess, and recover from complex, erroneous system behaviors (i.e. *composite events*) that are detected via patterns in stream data (i.e. *primitive events*). Error definitions are expressed as *nondeterministic finite automata* (NFAs), and composite events represent detected errors in an IoT system. BoboCEP is designed to be distributed across the network edge on k instances of the software. Each instance maintains the current state of partially completed NFAs via *active replication*, so that $(k-1)$ instances can fail without complete FT-support service loss.

[1]We use the error, fault, and failure definitions by Avižienis et al. [2].

The key contributions of BoboCEP are: (1) generic error definition and FT implementation, by defining errors as NFAs; (2) fault-tolerant FT support at the network edge, which facilitates long-term error detection without interruption; and (3) load balancing, due to the active replication of FT support across the edge. The rest of the paper is as follows. Section II discusses relevant background work. Section III explores BoboCEP and its design. Section IV evaluates BoboCEP with a vertical farming use case. Section V concludes our paper.

II. Background

In our previous work [6], [7], we explored how CEP systems could be used as an effective means of providing data-centric error detection, assessment, and recovery. Errors were defined as NFAs that are completed when applicable steam data fulfilled the states of the automaton; a completion represented the detection of an error. We adopted CEP as an FT-support framework, where all errors were defined as NFAs and would execute appropriate recovery actions on error detection. However, few CEP systems have been designed specifically for the IoT domain, and to provide a distributed, resilient approach to CEP-based FT support at the edge.

Ma et al. [8] proposed a *long-term CEP* (LTCEP) approach that enabled long-term events to be efficiently detected without excessive overhead. This was accomplished by splitting the detection processing into two phases: (1) online detection, which performed real-time detection within a limited time period before buffering any intermediate result it had achieved; and (2) offline detection, which generated a complex event using intermediate result linking from both online detection results and the event buffer. This approach was shown to significantly reduce redundancy in intermediate state and data.

Xie [9] proposed an *event sharing CEP* (ESCEP) system for the design of high event-overlap rate (i.e. repeatedly detecting the same event), to reduce the wasted energy consumption and increase processing efficiency. It used a hashing algorithm to decompose complex events into several intermediate events, which enabled them to be shared more easily.

Choochotkaew et al. [10] proposed EdgeCEP, a fully distributed CEP engine for the collaboration of devices at the edge network. Distributed task assignment and delivery is accomplished using tabu search and a heuristic flow-based greedy move algorithm. Their large-scale, simulated nursing

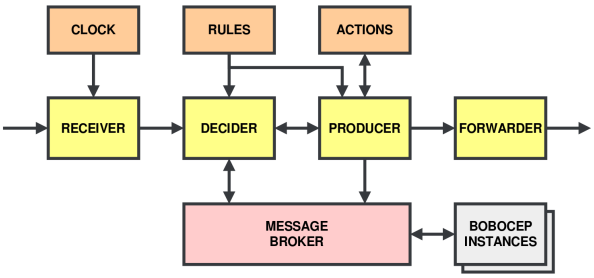

Fig. 1. The BoboCEP architecture.

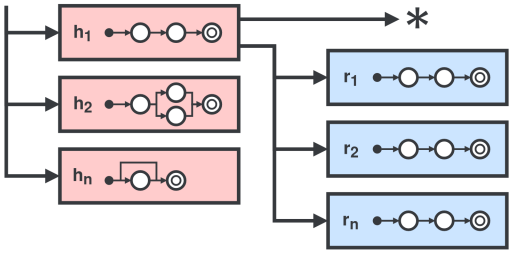

Fig. 2. Handlers (red) and runs (blue) in Decider. Handler h_1 currently contains runs $r_1, ..., r_n$ and can create more runs on-the-fly (asterisk).

home scenario showed that EdgeCEP approached 6.6 times fewer total packets than a centralized CEP approach.

EdgeCEP is the closest system to BoboCEP in terms of design and functionality, however our contribution is to provide *resilient* event processing in a distributed CEP environment. To the best of our knowledge, there does not exist a CEP system for IoT that is designed to provide a resilient FT support framework. We are able to provide long-term event processing, as with [8], by actively replicating system state, to protect against arbitrary edge device failures.

III. BoboCEP

A. Architecture

BoboCEP's architecture is inspired by the *information flow processing* (IFP) functional architecture proposed by Cugola et al. [11] that describes the main functional components that are common to all IFP systems, of which CEP systems are a subset. We have adopted the key components of the IFP architecture in the design of BoboCEP, as follows (Figure 1):

1) Receiver: Provides an entry point for data sources to push primitive events into BoboCEP. It uses an internal *Clock* to order and serialize events using timestamps to create an event stream.

2) Decider: The decision-making component of BoboCEP that determines whether composite events are generated or not (Figure 2). It generates a *handler* for each NFA in the set of all NFAs to be implemented (i.e. the *Rules*). A handler is a container for the NFA and all instances of the NFA that are created at runtime, which are called *runs*. Each handler contains a *buffer* that efficiently links events together that are shared by all of the runs for a given handler.

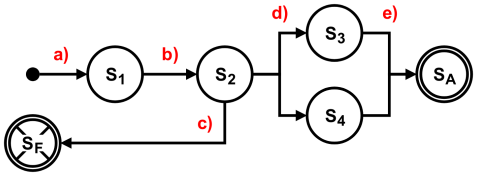

Fig. 3. An example NFA, with different transition types: (a,d) `CLONE`; (b) `TRANSITION`; (c) `HALT`; and (e) `FINAL`.

Our buffer is an implementation of the *Shared Versioned Match Buffer* proposed by Agrawal et al. [12] which ensures that only one version of an event is stored, even if it in use by multiple runs, in order to provide a more memory-efficient event processing approach. It has been successfully implemented in existing CEP systems, such as FlinkCEP[2].

Events from Receiver and Producer are passed to the handlers, which might trigger a run to be instantiated, or cause state changes (Section III-B) in existing runs. If a run reaches the accepting state S_A of its NFA (Figure 3), it will trigger a composite event to be generated by Producer.

3) Producer: Producer receives notification from Decider of a complex phenomenon (i.e. an error) being detected, and receives all of the events that infers the phenomenon (i.e. the *pattern*). The pattern, the time of detection, and the NFA, are used to generate a composite event representing the error. Producer then triggers the appropriate *Actions* to be executed, as determined by the Rules. That is: after error detection, Producer begins an assessment into the probable root cause of the error, and then executes an appropriate recovery strategy. The composite event is recursively sent back to Decider for potential use in the detection of future errors.

4) Forwarder: Delivers composite events from Producer to subscribed data sinks e.g. databases, external systems.

B. Message Broker

BoboCEP uses a message broker to replicate a state change in one instance to all other running instances, 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 3):

- `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

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

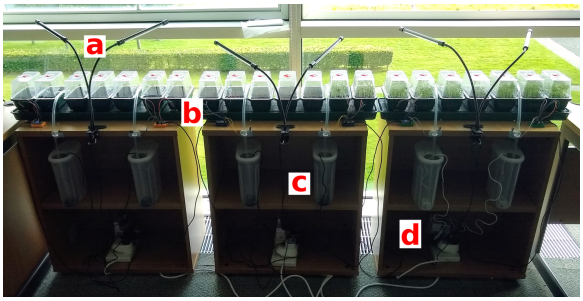

Fig. 4. Our indoor vertical farming testbed.

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 of this, and whether it was successfully executed or not.
- `SYNC`. When a BoboCEP instance first starts, it synchronizes with any other online instances that are already synchronized to retrieve the current state of FT support before starting error processing.

IV. Evaluation

This evaluation focused on how BoboCEP can provide our key contributions (Section I) as well as a software performance analysis. Our testbed was an indoor 'vertical farming' system i.e. growing produce indoors where environmental factors are tightly controlled [13]. Our system had three shelves (Figure 4), each with two water containers (c) that pumped water to the reservoir of its self-watering propagator. Grow lights (a) turned on when the room was dark, and off when bright.

Each shelf had two microcontrollers on the left and right of its shelf (b) that both monitored plant temperature and humidity and the light intensity around the shelf. The leftmost sensor on the left shelf is *LL*, rightmost is *LR*; *ML*, *MR* for middle shelf; and *RL*, *RR* for right shelf. Pumps and lights were connected to smart plugs (d) that enabled edge devices to control them.

We used BoboCEP (v0.35)[3], developed using Python (v3.7). Its Receiver consumed stream data via a Flask (v1)[4] 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*. We used RabbitMQ (v3.7)[5] for our message broker.

A. FT Scenario

1) Long-Term Event Processing: For our FT scenario, we considered the *trend check* NFA from our previous work [6], [7]. This NFA 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:

$$(e_2 - e_1)/2 \geq -200 \wedge e_2 \leq 500$$

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 5 (top) presents water level data that is representative of the type of data our system might produce. 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.

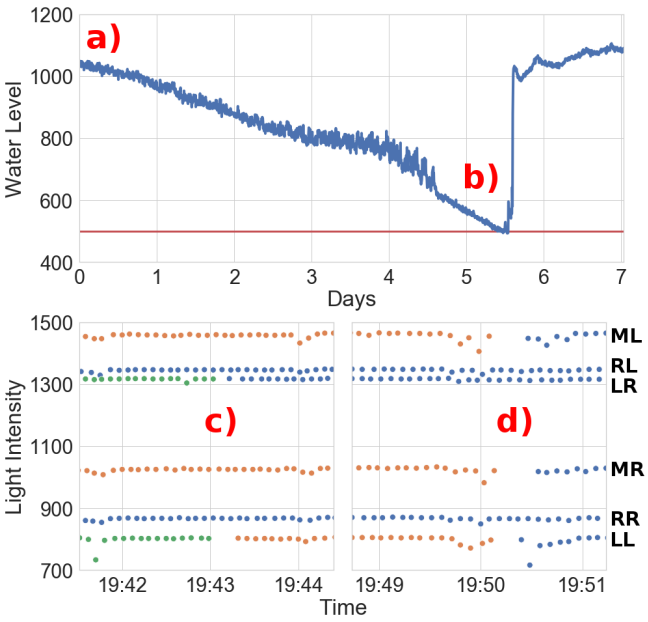

Fig. 5. *Top*: Water level data. *Bottom*: Light intensity data from Edge1 (green), Edge2 (orange), and Edge3 (blue).

When a BoboCEP instance receives the first water level value e_1 (Figure 5a), which contains value 1039, it triggers the instantiation of a run for the trend check NFA. The instance that created the run then sends a `CLONE` signal (Figure 3a) via the message broker, so that all other instances have a copy of the run also.

The second water level value e_2 (Figure 5b), which contains value 499, fulfils the above predicate i.e. $(499 - 1039)/2 = -270$. The instance that first reaches the run's the final state sends a `FINAL` signal (Figure 3e) to all other instances, so that they will complete their copies of the run also. A pump activation would occur (Figure 5b), leading to a sharp increase in water level value. This triggers an `ACTION` signal for the other instances to know this Action was executed successfully.

2) Load Balancing: Our microcontrollers 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 and simply try again with the next payload.

[3]https://github.com/r3w0p/bobocep
[4]https://palletsprojects.com/p/flask
[5]https://www.rabbitmq.com

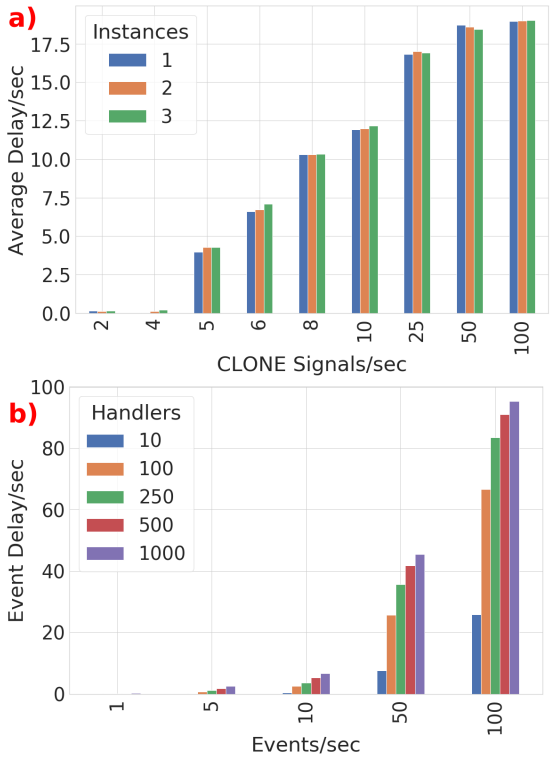

Fig. 6. Data from (a) run instantiation and (b) rule throughput experiments.

We connected LL, LR to Edge1; ML, MR to Edge2; and RL, RR to Edge3 (Figure 5, bottom). The light intensity data from the microcontrollers was relatively stable over a 10 minute interval. We manually terminated Edge1 (Figure 5c) and, after a brief data blackout, LL rerouted to Edge2, and LR to Edge3. Later, we then terminated Edge2 (Figure 5d), which caused all of Edge2's microcontrollers to reroute Edge3 i.e. the last remaining device. Despite 2 edge device failures, all sensor data was still available to the FT support system, and was balanced across any available instances at the time.

B. Performance

1) Run Instantiation: We wanted to test how instances coped with a large volume of `CLONE` signals. We set up 1, 2, and 3 instances, and passed 100 `CLONE` signals 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 6a). 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 serialized and broadcast `CLONE` signals to all instances equally.

2) Rule Throughput: We wanted to test how well an instance could handle a stream of data events relative to the number of NFA handlers it had. We preloaded an instance with k handlers that each had one incomplete run, so that a data event would be checked against both the handler (to determine 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`. We passed 1000 data events to the instance and measured how long it took to process them all (Figure 6b). Results showed that the event processing delay increased with the data rate, and incurred even larger delays when the number of handlers was greater.

V. Conclusion and Future Work

We proposed BoboCEP, a CEP system to provide generic, resilient FT support that is designed to be distributed at the network edge. Our evaluation demonstrated long-term error processing and edge device failures with a vertical farming testbed. We stress-tested BoboCEP and identified that it coped well with reasonable loads but incurred delays under high-velocity data flows. In future work, we will improve the efficiency of our BoboCEP implementation to achieve greater performance under heavy loads and high data throughput.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] ——, "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*. ACM, 2019, pp. 1:1–1:7.

[8] 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.

[9] 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.

[10] 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.

[11] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[12] 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*. ACM, 2008, pp. 147–160.

[13] 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.