Performance and Dependability Evaluation of Distributed Event-based Systems: A Dynamic Code-injection Approach

[Work-in-Progress Paper]

Saleh Mohamed, Matthew Forshaw, Nigel Thomas
School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
{s.mohamed,matthew.forshaw,nigel.thomas}@ncl.ac.uk

Andrew Dinn
Red Hat UK Ltd, UK
a.dinn@redhat.com

ABSTRACT

Distributed stream processing and event-based systems are an increasingly critical component in contemporary large-scale data processing applications, and are often subject to strict latency and reliability requirements. However, to achieve scalability demands, they are often deployed on distributed clusters of heterogeneous nodes, causing unpredictable runtime performance and complex fault characteristics.

The behaviour of these systems is poorly understood, and existing performance and dependability evaluation techniques are ill-equipped to handle the challenges introduced by the complex and distributed nature of event-based systems.

We develop a dynamic code-injection approach to evaluate the performance and dependability of stream processing and event-based systems. Our approach supports fine-grained instrumentation of applications and their runtime infrastructure, and the dynamic injection of code mutations and faults into a production system at runtime. We demonstrate the proposed approach by performing instrumentation and code injection on a distributed Apache Spark cluster.

Keywords

Event-based systems; performance; dependability

1. INTRODUCTION

Event-based systems and complex event processing (CEP) engines are an increasingly critical component in modern large-scale software deployments, e.g., Internet of Things (IoT). In order to make optimal deployment and resource management decisions, it is necessary to gain an understanding of the performance of systems. However, the performance and dependability characteristics of these systems are not well understood [6]. Furthermore, there are compelling scenarios to motivate autonomic operation and fault recovery of event-based systems [13, 3], but there is a reluctance – particularly within industrial applications – to add complexity to fault resolution scenarios. There is the belief that software will be buggy, especially under error cases and code which is executed infrequently.

Existing approaches to evaluate event-based systems have focused on the instrumentation of applications or infrastructure, but few have the ability to capture the interactions between the software deployment and its runtime environment [16]. Benchmarks for stream processing systems are emerging, but these generally only consider application-level metrics [7], and not infrastructure issues such as resource utilisation and energy consumption. Preliminary efforts are emerging to apply code injection techniques within stream processing systems [17], though these focus on instrumentation only, rather than code and fault injection.

Our work was motivated by the limitations of existing fault injection approaches, limiting their usefulness to evaluate the dependability of distributed event-based systems. Firstly, many approaches require the re-compilation of application code. Secondly, there is limited flexibility in when faults can be injected in the application lifecycle. Finally, coordinating complex test scenarios enacted across multiple nodes in distributed clusters is an open challenge.

We present an approach which addresses these key challenges to evaluate the performance and dependability, using our non-invasive dynamic code injection tool, Byteman [4]. Our approach allows a practitioner to instrument an application, and develop code injection rules with only the knowledge of the public interface of the application, and without the need to adapt or re-compile the application. We can *'attach'* our tool to a deployment at runtime, and dynamically load and unload our rules during the execution of the system under evaluation. Our system facilitates greater test coverage. Finally, our approach allows programmatic specification of complex fault scenarios, across distributed nodes.

The remainder of this paper is organised as follows. In Section 2 we present our architecture, and describe how code and fault injection rules are specified in the Byteman language. In Section 3 we demonstrate the applicability of our approach in performance evaluation and dependability as-

ICPE'17, April 22 - 26, 2017, L'Aquila, Italy
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4404-3/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3030207.3030245

sessment of an Apache Spark cluster. We present related work in Section 4. Finally, in Section 5 we provide concluding remarks and present future challenges and research directions.

2. ARCHITECTURE

We present our system architecture in Figure 1. This shows a typical deployment of our approach to add dynamic code injection to an Apache Spark cluster distributed across remotely running Java Virtual Machines (JVMs). We evaluate this representative use case in Section 3.

Fault Load: We adopt a fault load which describes the types of faults experienced on the system under test [9] and runtime environment [16]. We can also use metrics obtained from production deployments to inform our fault load through workload characterisation.

Byteman Rules: The defined 'Fault Load' acts as the basis when developing a Byteman ruleset which captures these failure types, targeting components of the runtime system to recreate these faults.

Test Scenario: One or more rules may be then composed, alongside timing information and other required metadata, to construct a test scenario.

Test Coordinator: Our tool is capable of ingesting a test scenario definition, automatically deploying the test infrastructure [?] and enacting code injection through the Thermostat Client, which communicates over a Command Channel with Thermostat agents on each node.

On each compute node we instrument, we run two *'agents'*, the *'Thermostat agent'* and *'Byteman agent'*.

Byteman Agent: Byteman [4] is an open-source dynamic Java bytecode manipulation tool, facilitating code injection into running JVM processes.

Thermostat Agent: Thermostat [2] is an open-source instrumentation and monitoring tool, facilitating performance monitoring covering various aspects of the operating system and fine-grained JVM behaviour. Thermostat is responsible for instructing Byteman to perform code injection into a JVM, specified by a *'rule'* (Section 2.1).

Storage Layer: Logging data and metrics – whether emitted by Byteman rules or obtained from OS/JVM processes – are aggregated via the Thermostat agent and saved to a MongoDB persistent store.

2.1 Code injection with Byteman Rules

Byteman rules are expressed as Event-Condition-Action rules (ECA-rules) [5], describing ***where*** during application execution a side-effect should occur, ***whether*** the side-effect should happen or not, and ***what*** the side effect should be.

Events: A rule specifies the `CLASS` and `METHOD` the rule targets, as well as the stage in the method call lifecyle the rule applies (e.g. `ON EXIT, AT INVOKE`). A location specifier (e.g. `AT ENTRY`, `AT EXIT` or `AT LINE`) specifies the trigger point within the target method call.

Condition: Rules can be enacted conditionally, based on boolean rule expressions.

Action: POJO code and/or Byteman built-in calls supporting complex thread coordination, exception raising.

We explore each aspect of an ECA-rule, with respect to a representative example of a Byteman rule in Listing 1. This rule instruments Spark batches. The rule is enacted `AT EXIT` for the `onBatchCompleted` method within the `spark.JobListener` class, and uses `BIND` to access a number of instance variables within the class. Each time the rule is triggered, the rule dispatches these bound values – in JSON format to the MongoDB persistent store – using the `send(...)` method of the Thermostat `HELPER` class.

Listing 1: Exemplar Byteman Rule

```
RULE Instrumentation of Spark job batches
CLASS spark.JobListener
METHOD onBatchCompleted
AT EXIT
HELPER org.jboss.byteman.thermostat.helper.ThermostatHelper
BIND timestamp = $time,
      input_size = $inputSize,
      processing_time = $processingTime,
      scheduling_delay = $schedulingDelay,
      total_delay = $totalDelay
IF TRUE
DO
    debug("Sending batch metrics to thermostat");
    send("map", new Object[] { "timestamp", timestamp,
                               "batch_size", input_size,
                               "processing_time", processing_time,
                               "scheduling_delay", scheduling_delay,
                               "total delay", total_delay});
ENDRULE
```

2.2 Rule classification

Byteman rules may be written to target components at different levels of the applications and infrastructure.

1) Operating system (OS) and environment issues can be targeted, e.g. node crash, network connectivity, resource contention, launching of external processes leading to interference, etc.

2) We can explore the impact of JVM-specific issues e.g. Stop-the-World (STW) garbage collection, memory leaks, thread deadlocks, etc.

3) Issues arising related to the particular event-based system under test, e.g. Spark-specific exceptions and faults.

4) Finally, we consider domain-specific faults concerning the user's application, e.g. performance degradation. At each level, we consider two broad classes of *rule*:.

Instrumentation: Collection of metrics not otherwise exposed by the system. We can optionally share state between Byteman rules at runtime, such that the enactment of a rule be conditional on global system state; e.g. a rule could be triggered when a tuple arrives to a worker node with high CPU load.

Fault Injection: A class of rule to bring about a failure of a component, representative of a real-world fault. For example, one may trigger node crashes, deadlocks, or impose probabilistic delays to processing (based on models derived from empirical evaluation of production systems, e.g. [11]).

2.3 Coordinated failure scenarios

Many previous works focus on *'independent'* failures, namely those affecting an individual node. In the context of distributed event-based systems, we must consider scenarios exhibiting inter-cluster fault propagation, where a fault on a particular node has a knock-on impact across the cluster.

A significant contribution of our approach is to offer programmatic specification of distributed test scenarios. These scenarios control the automatic provisioning of runtime infrastructure, automated code injection allowing multiple Byteman rules to be injected across nodes in a cluster. This allows us to enact complex test scenarios to evaluate emergent

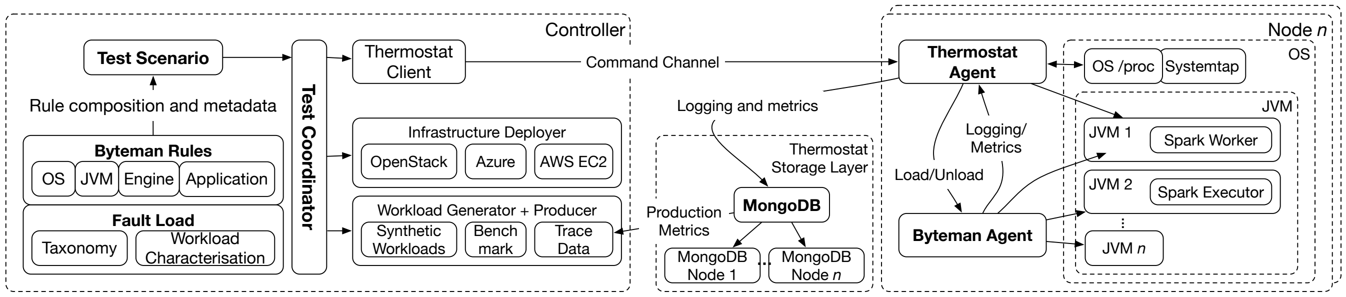

Figure 1: System architecture to support code-injection of event-based and stream processing systems.

behaviours across distributed clusters, e.g. fault propagation. This work is closely aligned with our related research concerning the automated deployment of distributed event-based infrastructure to support IoT applications [14].

3. EVALUATION

Fault-injection approaches are often evaluated in the literature according to the tradeoff between three key requirements; *representativeness*, *usability* and *efficiency* [15]. Here we demonstrate our approach satisfies each of these, and is a suitable option for evaluating event-based systems.

Representativeness: The representativeness of a fault injection approach describes the realism of the tests, in relation to the types of faults encountered by the system and its environment. Our ability to compose rules together allows us to explore sophisticated test scenarios representative of those specified in *fault loads* for stream based systems.

Usability: Our developed tool is portable, and can be used with any stream processing system with little modification. Our simple rule specification, as highlighted in Section 2.1, allows rules to be written for any application whose Java public interface is known. In Section 3.2 we show our tool to be minimally intrusive, with the loading and unloading of our rulesets having negligible impact on the target workload and its underlying runtime infrastructure.

Efficiency: We lower experimental effort by offering automated test infrastructure deployment and tooling support for runtime injection, and we offer highly targeted fault injection through expressive rule language (Section 2.1).

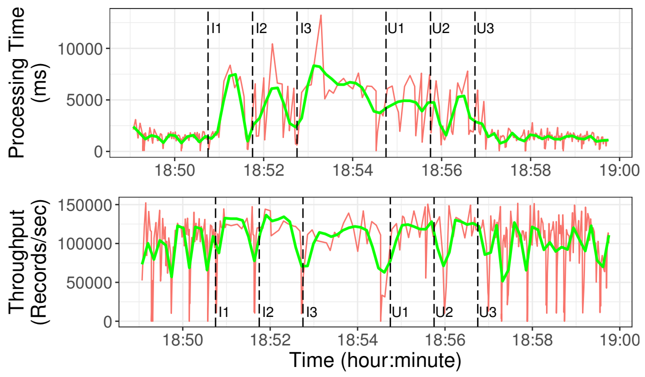

Figure 2: Code injection of probabilistic processing delays in a distributed Apache Spark cluster.

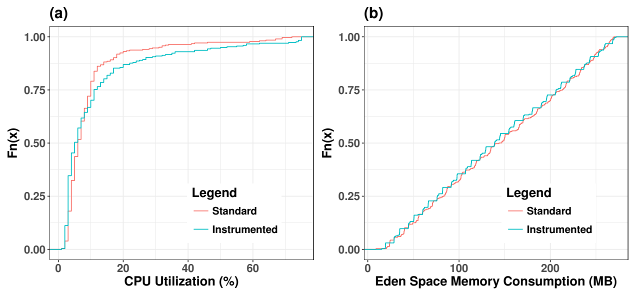

Figure 3: Injection overhead (a) CPU, (b) Memory

3.1 Example scenario

We demonstrate the effectiveness of our approach in experiments using Apache Spark [1]. We use a cluster of Microsoft Azure `DS1_V2 standard` virtual machines (1 core, 3.5GB disk space and 7GB of RAM) running Ubuntu 14.04. The cluster consists of one single-node Kafka server, one Spark master, and three Spark workers.

The workload is a Spark streaming word count application where a Kafka producer publishes lines of text into a Kafka server. The streaming application consumes the lines of text using the Spark-Kafka connector and performs a series of Spark built-in RDD transformations (`map`, `mapToPair` and `reduceBykey`) to generate the word counts.

Due to limiting space, we evaluate a single use case; injection and removal of tuple processing delays on our three Spark workers. Figure 2 shows the results of our experimentation on application throughput and processing time, where the red lines are the actual time series and the green lines represent the moving averages. The annotation lines mark the time when delaying faults were introduced to the system. At points *I1*, *I2* and *I3* we inject a 10ms processing delay on every 1000th message processed, for worker node one, two and three respectively. We see this leads to increased processing time, and degraded throughput. At points *U1*, *U2* and *U3* we unload the rules from worker node one, two and three respectively. It can be seen that processing time and throughput recover to their original level.

3.2 Performance Evaluation

In order for performance and dependability measurements to be meaningful, we must demonstrate that the presence of our instrumentation does not perturb the normal operation of the system under evaluation.

Here we evaluate the impact of the dynamic loading and unloading of Byteman rules into a production Apache Spark cluster. We collect two ten-minute performance traces from a node in a Spark cluster, with and without code injection. Figure 3(a) shows the empirical CDF plot of CPU load, while Figure 3(b) shows JVM Eden Space Memory Consumption. In both cases, we observe that our code-injection approach has negligible impact on host resource utilisation, giving us confidence our approach is non-intrusive.

4. RELATED WORK

Lopez *et al.* [12] explore the performance of Apache Storm, Apache Flink, and Apache Spark Streaming, with respect to message processing performance in the presence of node failures. The authors provide experimental results from a testbed comprising eight virtual machines, comprising one master node and eight workers. To emulate node failures, one virtual machine is turned off. Meanwhile, Heorhiadi *et al.* [8] propose Gremlin, an approach to evaluating fault-tolerance of microservice architectures, through network-level manipulation of inter-service messages.

Vögler *et al.* [17] demonstrate the use of the AspectJ aspect-oriented programming (AOP) framework to instrument and collect performance measurements from an Apache Spark and Apache Storm cluster. This research focuses on the instrumentation of production stream processing systems, while our research furthers the application of fault injection in event-based systems, by supporting dynamic injection of faults and automated management of the testing lifecycle, including infrastructure provisioning.

Hummer *et al.* [9] present a taxonomy of classes of faults encountered in event-based systems such as, event stream processing (ESP) and complex event processing (CEP) systems. Pietrantuono *et al.* [16] present a characterisation of software faults arising from the runtime environment. Both of these efforts are complementary to ours. Their findings can inform our *'Fault load'* and our derived Byteman rules.

Gupta *et al.* [7] present BFT-Bench for evaluating Byzantine Fault Tolerance algorithms using fault injection. Jacques-Silva *et al.* [10] consider a fault-injection approach to evaluate the viability of Partial Fault Tolerance (PFT) for a financial application running within IBM System S. Their approach involves developing a fault injection operator (FIOP) which emulates a particular faulty operator behaviour. The application is then recompiled, with these faulty operators placed directly infront of the *'target'* operator.

5. CONCLUSION

In this paper we have shown how a dynamic code injection approach can be used to instrument distributed event-based systems for performance and dependability evaluation. Our approach provides practitioners with a usable set of tools which address many common issues inhibiting automated and holistic performance and dependability evaluation of event-based systems. We contribute our source code to the community as open-source through Github[1].

Our future work will focus on a full-scale evaluation of our approach, developing further tooling support, enabling practitioners to model more sophisticated failure scenarios.

[1]https://github.com/ncl-IoT/Fault-Injection

6. ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council, Centre for Doctoral Training in Cloud Computing for Big Data [grant number EP/L015358/1].

7. REFERENCES

[1] Apache Spark. http://spark.apache.org/.

[2] Thermostat. http://icedtea.classpath.org/thermostat/.

[3] T. Cooper. Proactive Scaling of Distributed Stream Processing Work Flows Using Workload Modelling: Doctoral Symposium. In *DEBS'16*, pages 410–413. ACM, 2016.

[4] A. E. Dinn. Flexible, Dynamic Injection of Structured Advice Using Byteman. In *Proceedings of the Tenth International Conference on Aspect-Oriented software Development companion*, pages 41–50. ACM, 2011.

[5] K. R. Dittrich, S. Gatziu, and A. Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In *RIDS*. Springer, 1995.

[6] M. Forshaw, N. Thomas, and A. S. McGough. The Case for Energy-Aware Simulation and Modelling of Internet of Things (IoT). In *ENERGY-SIM*, 2016.

[7] D. Gupta, L. Perronne, and S. Bouchenak. BFT-Bench: A Framework to Evaluate BFT Protocols. In *ACM/SPEC ICPE '16*, 2016.

[8] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic Resilience Testing of Microservices. In *IEEE ICDCS*, 2016.

[9] W. Hummer, C. Inzinger, P. Leitner, B. Satzger, and S. Dustdar. Deriving a Unified Fault Taxonomy for Event-Based Systems. In *ACM DEBS*, 2012.

[10] G. Jacques-Silva, B. Gedik, H. Andrade, K.-L. Wu, and R. K. Iyer. Fault Injection-Based Assessment of Partial Fault Tolerance in Stream Processing Applications. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based system*, pages 231–242. ACM, 2011.

[11] A. Khoshkbarforoushha and R. Ranjan. Resource and Performance Distribution Prediction for Large Scale Analytics Queries. In *ACM/SPEC ICPE*. ACM, 2016.

[12] M. A. Lopez, A. Lobato, and O. Duarte. A Performance Comparison of Open-Source Stream Processing Platforms. In *IEEE Globecom*, 2016.

[13] P. Michalák, S. Heaps, M. Trenell, and P. Watson. Doctoral Symposium: Automating Computational Placement in IoT Environments. In *DEBS'16*, 2016.

[14] S. Mohamed, M. Forshaw, and N. Thomas. Automatic Generation of Distributed Run-time Infrastructure for Internet of Things (IoT). In *ICSAW'17*, 2017.

[15] R. Natella, D. Cotroneo, and H. S. Madeira. Assessing Dependability With Software Fault Injection: A Survey. *ACM CSUR*, 48(3):44, 2016.

[16] R. Pietrantuono, S. Russo, and K. Trivedi. Emulating Environment-Dependent Software Faults: Position Paper. In *COUFLESS'15*, pages 34–40. IEEE Press, 2015.

[17] M. Vögler, J. M. Schleicher, C. Inzinger, B. Nickel, and S. Dustdar. Non-Intrusive Monitoring of Stream Processing Applications. In *2016 IEEE SOSE'16*, pages 162–171, March 2016.