Effectively Testing System Configurations of Critical IoT Analytics Pipelines

Morgan K. Geldenhuys*, Lauritz Thamsen*, Kain Kordian Gontarska†, Felix Lorenz* and Odej Kao*
*Technische Universität Berlin, Germany, {firstname.lastname}@tu-berlin.de
†Hasso Plattner Institute, University of Potsdam, Germany, kordian.gontarska@hpi.de

***Abstract*—The emergence of the Internet of Things has seen the introduction of numerous connected devices used for the monitoring and control of even Critical Infrastructures. Distributed stream processing has become key to analyzing data generated by these connected devices and improving our ability to make decisions. However, optimizing these systems towards specific Quality of Service targets is a difficult and time-consuming task, due to the large-scale distributed systems involved, the existence of so many configuration parameters, and the inability to easily determine the impact of tuning these parameters.**

In this paper we present an approach for the effective testing of system configurations for critical IoT analytics pipelines. We demonstrate our approach with a prototype that we called *Timon* which is integrated with Kubernetes. This tool allows pipelines to be easily replicated in parallel and evaluated to determine the optimal configuration for specific applications. We demonstrate the usefulness of our approach by investigating different configurations of an exemplary geographically-based traffic monitoring application implemented in Apache Flink.

***Index Terms*—Distributed Stream Processing, Internet of Things, Configuration Testing, Quality of Service.**

I. Introduction

The Internet of Things (IoT) is an important emerging technological paradigm whereby billions of ubiquitous sensor and actuator devices are connected to enable the development of applications across a wide number of domains. An increasing number of these applications are expected to perform in a capacity where the services they provide must meet certain minimum Quality of Service (QoS) requirements. This is especially relevant for applications used in the real-time monitoring and control of Critical Infrastructures, such as: human health-care, transportation systems, electrical generation, natural disaster prediction, and telecommunications, to name but a few [1]–[4].

As the number of Internet-connected devices increases year-on-year, so does the volume of data being produced. In order to process these large data streams, Distributed Stream Processing Frameworks (DSPF) such as Storm [5], and Flink [7] allow for the deployment of analytics pipelines which utilize the processing power of a cluster of commodity nodes. Therefore, these frameworks are being utilized increasingly for the processing of IoT data streams [8]–[11]. Applications developed within these systems are, in principle, required to operate indefinitely on an unbounded stream of continuous data in an environment where partial failures are to be expected as these applications scale. Consequently, DSPFs feature high availability modes, implement fault tolerance mechanisms by default, and expose a rich set of continually evolving features. The end result being that the way in which these systems are composed has a high level of complexity and number of configuration options. A quick scan of the official documentation reveals that Flink has over 300 options across 28 categories[1], and Spark [6] closer to 400 across 26 categories[2].

System configuration has an impact on performance and reliability. Yet, with the vast number of options available for tuning, i.e. framework settings, job parameters, resource selections, etc., the effects of which are not always well understood or straightforward to determine. That is, finding the best combination of resource selections and system configurations is difficult to estimate upfront both by experts and automatically by optimization tools as it is highly dependent on a number of key factors: *the analytics application* which exhibits its own unique operational characteristics; *the cluster environment* which is often not known before deployment and may vary over time, i.e. network topologies and physical hardware; and *the data* which is variable based on the characteristics of the input data, loads from other applications, and ingestion rates. This is especially true in environments consisting of multiple connected distributed systems making up larger application architectures, such as: resource managers, messaging queues, distributed file systems, scalable databases, etc. At the same time, critical IoT applications typically have defined QoS requirements with regards to performance, reliability, etc, which a configuration should meet [16].

Currently, the most common way of tuning configuration parameters is for it to be done manually by performance engineers, usually requiring several hours of investigation and testing [17]. These engineers require detailed knowledge of the specific DSPF itself and the cluster environment in order to find a system configuration that falls inline with the aforementioned QoS constraints. Approaches have been proposed at finding more precise and less time-consuming methods for the automatic tuning of DSPF parameters [18]–[22]. These typically focus on only a limited number of settings, while there are numerous points of configurations in practice with many dependencies between them. A solution is needed which is complementary to these existing performance modelling approaches, which provides an approach for

[1]Flink Configuration. URL: https://ci.apache.org/projects/flink/flink-docs-stable/ops/config.html

[2]Spark Configuration. URL: https://spark.apache.org/docs/latest/configuration

gathering analytics data through testing and monitoring.

For this purpose, we propose an approach for the effective testing of system configurations for critical IoT analytics pipelines in realistic conditions. We implemented our approach using a prototype called *Timon* which allows for the testing of multiple different versions of system configurations in parallel within an environment that behaves like production using real streaming data. In this way, operators can safely and efficiently experiment with potential system configurations to understand what impact these will have when used in production.

The remainder of the paper is structured as follows: Section II discusses the related work with regards to configuring DSPFs, Section III presents a typical architecture for critical IoT analytics pipelines, Section IV presents our approach to configuration testing, Section V describes our evaluation where we present our experiments and findings, and Section VI discusses our findings with conclusions.

II. Related Work

There exists a large body of work which addresses the problem of system configuration. A number of these approaches focus specifically on the tuning of parameters for DSPFs. This typically involves learning from analyzing actual executions or historical data, model specific aspects of the systems, and then adapt to actual conditions based on user requirements. We see our work as being orthogonal and complementary to these contributions in that Timon provides a testing and metric gathering environment within which these approaches could function. These approaches can be categorized as follows:

Rule-based: A gray-box heuristic approach where domain experts work with users to establish a rule-set which is used to recommend suitable configurations. Bilal et al. [20] present an approach where users provide a parameter ranking in accordance with a priority level and specify whether an increase in parameter value has an overall positive impact on latency and throughput. This approach favors quickly finding a suitable configuration at the expense of optimality.

Model-based: An approach which is concerned with conducting experiments on a chosen set of configurations to observe their performance. The results are used to train a statistical model for finding good configurations. Fischer et al. [18] and Trotter et al. [21] present an auto-tuning algorithm using Bayesian Optimization (BO) [23] to achieve high throughput. Jamshidi et al. [19] likewise proposes BO, however, it optimizes latency and leverages Gaussian Processes [24] to continuously estimate the mean and confidence interval of a response variable at yet-to-be explored configurations.

Search-based: For this approach, an initial configuration is selected after which experiments are conducted sequentially. Each iteration uses the results of the previous to fit a statistical model which is used to select the next configuration. Evolutionary Search Algorithms are typically adopted for automatic parameter tuning. Trotter et al. [21] proposes a method using genetic algorithms (GA) to optimize throughput. Additionally, in a later paper Trotter et al. [22] use GA to optimize throughput using SVM classifiers to further refine its search of the configuration space. Bilal et al. [20] proposes a hill-climbing algorithm based on Latin Hypercube Sampling [25] while taking both latency and throughput metrics into account.

Learning-based: An approach which use online learning techniques such as reinforcement learning to find the optimal configuration by reacting to feedback, i.e. metrics, from the DSPF at runtime [26]. This approach can be combined with offline learning techniques to speed up convergence [27].

To the best of our knowledge, no approach exists which focuses on parameter tuning for critical IoT analytics pipelines executing in the production environment. For these applications it is essential to consider the time-dependant nature of IoT data streams when optimizing the performance of DSPFs.

III. IoT Stream Processing Architecture

In this section we assume a typical architecture for the processing of IoT data streams, as depicted in Fig. 1. Here we see a number of systems which, when combined, provide a typical way of composing critical IoT analytics architectures.

The *distributed streaming platform* is where raw IoT data flows into the system from sensor devices and is stored in a messaging queue to await processing. This component is an implementation of the publisher/subscriber messaging pattern with Apache Kafka [13] being a distributed example of this.

Subscribers such as the *IoT analytics pipeline* register themselves with the distributed streaming platform and consume messages from targeted messaging queues when they becomes available. Additionally, messages that have already been processed and produce alarms or notifications, for instance, can be written back to a separate messaging queue for further consumption. The IoT analytics pipeline in turn consists of a number of inter-dependent systems. These systems include:

- *Distributed Stream Processing Framework*: responsible for executing the *IoT analytics application* with the current *configuration set* in order to process messages read from the distributed streaming platform. Once processed, outputs are written back to the distributed streaming platform if necessary and the *monitoring & analytics data store* for archival. E.g. Apache Flink.
- *Monitoring & Analytics Data Store*: this scalable database warehouses all sanitized messages and outputs of the IoT analytics application. The archived data can be used for analysis over a longer period of time to detect trends and other anomalies. E.g. Apache Cassandra [15].
- *Metrics*: it is important to monitor the health of the IoT analytics application being executed. For this purpose, most DSPFs like Flink and Spark generally have a metric system that allows for the gathering and export of internal metrics to external systems. These measurements should be stored in a time series database to be accessed from outside the cluster. E.g. InfluxDB[3].
- *Chaos Daemon*: an optional component, however, if deployed could provide the facility for using Chaos

[3]InfluxDB. URL: https://github.com/influxdata/influxdb

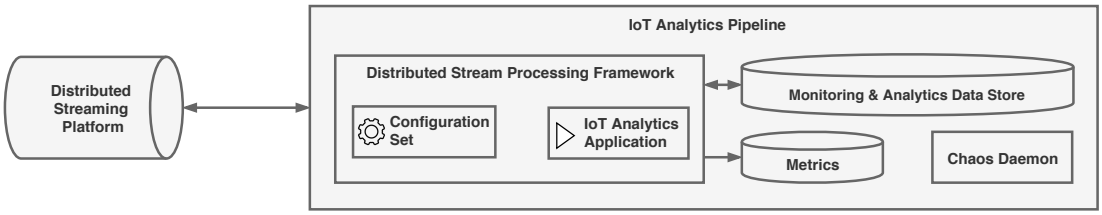

Fig. 1. Typical IoT stream processing architecture.

Engineering techniques to promote the development of resilient services [28]. E.g. PowerfulSeal[4].

In such a setup, the configuration set and/or IoT analytics application could be designed to a standardized specification allowing them to be traded out for alternate versions without causing too much of a disruption to the overall environment.

IV. Approach

In distributed stream processing, system configuration has a direct impact on performance and reliability. Yet, quantifying exactly how much of an impact is hard to ascertain. This is complicated by the existence of so many configuration parameters and the fact that no two stream processing applications with even a minor difference in operational characteristics is likely to share the same optimal setup. Additionally, if a more performant version of the configuration were to be found, migrating to this version in the production environment could cause significant disruptions. It is the goal of Timon to find solutions to these problems.

From a high level perspective, determining the best system configuration for any particular stream processing application can be found by comparing it to: the same application executing in the same environment while ingesting the same data but using alternate variations of the configuration set. For such a test, variants should be executed in parallel, metrics recorded over a specific time interval, and on conclusion, results compared to determine the best performer(s). This is the general idea behind Timon, to provide a testing system for the efficient comparison of alternative configurations in the production environment, i.e. testing with actually deployed systems, at scale, and with actual live data streams.

One of the key requirements of such a testing system would be the ability of an environment where alternate deployments can be quickly replicated. Moreover, these deployments would need to be isolated from each other in order to eliminate interference and provided with access to external services. For this purpose we make use of two key enabling technologies, i.e. OS-level virtualization and container orchestration. These technologies, when combined with Infrastructure-as-Code (IaC) processes, provide a mechanism for efficiently instantiating entire pipelines in parallel, assuming enough resources are available in the cluster to do so.

Fig. 2 provides an architectural overview of Timon and its dependencies. In this diagram we can see the virtual cluster environment where both the *production pipeline* and shorter-lived *configuration testing pipelines* exist and are managed by the container orchestrator. The flow of data is from left to right, from source, i.e. *distributed streaming platform*, to sink, i.e. *client gateway*. Each configuration testing pipeline is composed in the same way as the production pipeline and would be executing the same *IoT analytics application*. Importantly, all configuration testing pipelines will also process the same input data as the production pipeline. When notifications and alarms produced by a configuration testing pipelines needs to be written back to the distributed streaming platform, they will each have their own unique messaging queues.

As part of the assumed *IoT stream processing architecture* described in the previous section, each IoT analytics pipeline records metrics in a time series database. After all testing rounds have concluded, these metrics are collected, aggregated, and subsequently analyzed to determine if statistically significant effects were observed. If a better performing testing pipeline is found than the current production pipeline, then a strategy can be followed to replace it. This strategy first involves migrating all data not stored in the production pipeline to the new candidate pipeline, i.e. message queue and data store. Next, user traffic needs to be redirected towards the new data sources via the client gateway. In this way the client gateway can be thought of as a load balancer. Lastly, all redundant pipelines can then be safely decommissioned and resources recovered. It is important to note that the copying of archived data over the network can be an expensive operation both in terms of time and network resources. It is therefore prudent to follow a strategy which will minimize this impact. Container orchestrators such as Kubernetes [12] offer a number of mechanisms for working with persisted data[5].

Apart from performance, reliability testing is also important for understanding the behavior of distributed systems. The amount of things that can go wrong while a distributed system is running is enormous. This is mainly due to the distributed nature of all the components (which interact exclusively through direct message parsing). It is virtually impossible to predict every possible failure mode and then engineer solutions for all the edge cases. Instead, a more realistic approach would be to identify the weaknesses which cause these failures before they are triggered. This is where Chaos Engineering practices can be used to complement traditional testing approaches.

[4]PowerfulSeal. URL: https://github.com/bloomberg/powerfulseal

[5]Persistent Volumes. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes

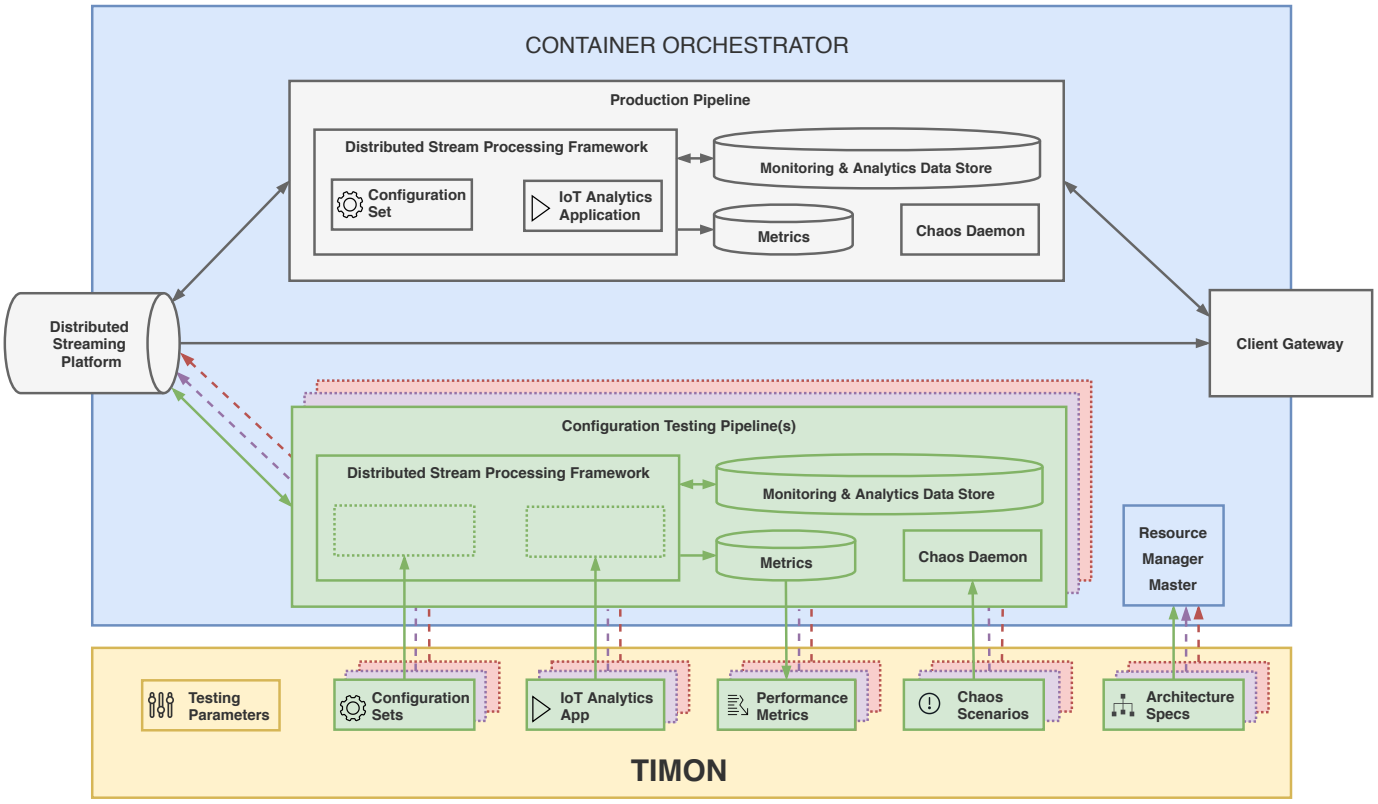

Fig. 2. Overview of Timon and system dependencies.

Therefore, Timon provides the ability to optionally define *failure scenarios* whereby failures are injected into the testing environment so that their impacts can be studied.

V. Evaluation

Now we demonstrate that using Timon is both practical and beneficial for distributed stream processing by presenting an experiment conducted to evaluate the impact of using different checkpoint intervals on the overall performance of the system. Ensuring that DSPFs are fault tolerant while running in the production environment is imperative, however, quantifying this impact is important when considering QoS.

A. Prototype Implementation

Timon is essentially a software client which interfaces directly with a container orchestrator to automatically manage the instantiation / destruction of container groups. These container groups compose the inter-dependent systems which in turn make up the individual analytics pipelines. For implementation, we use Docker[6] and Kubernetes. These technologies were selected because they provide an IaC approach for the management and provisioning of pipelines through machine-readable definition files. Additionally, Kubernetes provides a mechanism for isolating pipelines through the use of namespaces, thereby minimizing the possibility of interference.

[6]Docker. URL: https://docker.com

B. IoT Data Stream & Analytics Application

For the purposes of this experiment, we created a simulation which mapped the streets and intersections of an area with one kilometer radius of central Berlin, Germany. In this area we generated a number of vehicles which travelled along various routes while providing an update message every 1 second. This update contained the: vehicle ID, vehicle type, current location, speed, and direction. We use a sinusoidal function to model traffic behaviors where the number of simultaneous vehicles is varied from a minimum of 25,000 to a maximum of 75,000 as a function of the time of day (t in seconds). This was done to more closely resemble real traffic behaviors rather than a linear gradient, i.e. the number of vehicles gradually increases until a peak point (rush hour), before gradually decreasing again. Messages are submitted to an Apache Kafka cluster to await processing by the IoT analytics pipeline.

The live stream of traffic messages stored in Apache Kafka are consumed and analyzed using a DSPF. We use Apache Flink for our experiments as it has native support for fault-tolerant stream processing and is known for high performance and low latency [7]. A Flink cluster implements a master-slave architecture which consists of two processes: the JobManager and the TaskManager. We developed an analytics application for the purpose of analysis and define the following task: Determine the total number of different vehicle types within the simulation area accumulated over a 5 minute window period. Results were outputted to an Apache Cassandra database. This

task uses ”group by” transformations where the stream was logically partitioned into disjointed partitions. All messages with the same key, therefore, were assigned to the same partition which allowed for a high level of parallelism. The long windowing period of 5 minutes results in a larger accumulating of state across the parallel tasks and therefore is a good fit for testing fault tolerance behaviors.

C. Experimental Setup

Our experimental setup consists of a 3 node Apache Kafka cluster and a 30 node Kubernetes cluster with HDFS [14]. Node specifications are shown in Table I.

TABLE I
CLUSTER SPECIFICATIONS

Resource	Details
OS	Ubuntu 18.04.3
CPU	Quadcore Intel Xeon CPU E3-1230 V2 3.30GHz
Memory	16 GB RAM
Storage	3TB RAID0 (3x1TB disks, linux software RAID)
Network	1 GBit Ethernet NIC
Software	Java v1.8, Apache Flink v1.9.0, Apache Kafka v2.3.0, Apache ZooKeeper v3.5.5, Docker v18.06, Kubernetes v1.15.3, Apache HDFS V2.8.3, Apache Cassandra v3.11.4, InfluxDB v1.6.4

We limit our configuration sets to vary a single variable, i.e. checkpoint interval, and choose 3 different variations representing short (*1000ms*), medium (*20,000ms*), and long (*120,000ms*) intervals. Using Timon, we created 3 corresponding configuration testing pipelines in Kubernetes, each composed of:

- an Apache Flink (High Availability) cluster of 11 instances (1 JobManagers and 10 TaskManagers);
- an Apache ZooKeeper cluster of 3 instances for distributed coordination;
- an Apache Cassandra cluster of 3 instances for archival of processed data; and
- a single InfluxDB time series database instance for collection of performance measurements.

We define four key indicators to measure the performance of each configuration set. These are: *end-to-end latency*, in DSPFs, is the time difference between the moment a message is produced at the source task and the moment the tuple is produced at the output; *input throughput*, measured in messages / sec, is the cumulative frequency at which messages enter the source tasks of the dataflow, and; *CPU utilization* and *heap memory utilization* as a percentage.

The total time to provision pipelines for each experimental testing round averaged 330 seconds. There were 5 rounds of testing conducted where metrics were recorded over 6 hours with an increasing input throughput of 25,000 to 75,000 messages per second.

D. Experimental Results

In the experiments, during the user-defined time interval, metrics were recorded and saved to a time-series database. After all testing rounds were concluded, Timon automatically retrieved these metrics, aggregated them, and analysis was performed. Parallelism for the dataflow job was set to eight. This resulted in one sink operator executing for each of the eight active TaskManagers. Latency, therefore, is recorded at each sink operator separately. In order to address the observed variance in the performance metrics, the median latency value from all sink operators was chosen for each timestamp. The same was applied to the TaskManagers for CPU and memory utilization. Furthermore, the experiment was run for a total of five testing rounds over the same time interval, i.e. time of day. Again, the median values for each time step were chosen to be the expected values. To further remove noise from the diagrams, exponential weighted moving average windows with a span of 1000 seconds were applied to the averaged metrics.

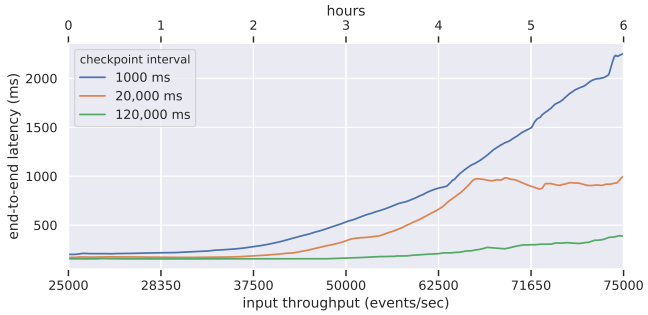

Fig. 3. End-to-end latencies.

Fig. 3 shows the average latencies as measured at the sink operator across all TaskManagers. Here we can clearly see how, as input throughput increases, performance deteriorates across all configurations. Additionally, latencies decrease more drastically the shorter the checkpoint interval as input throughput increases. It is visible there is a trade-off between QoS requirements, i.e. the maximum amount of time before a message should be processed, and the recovery time should a failure occur, i.e. the time for the system to go to a state where all messages are processed up until the failure.

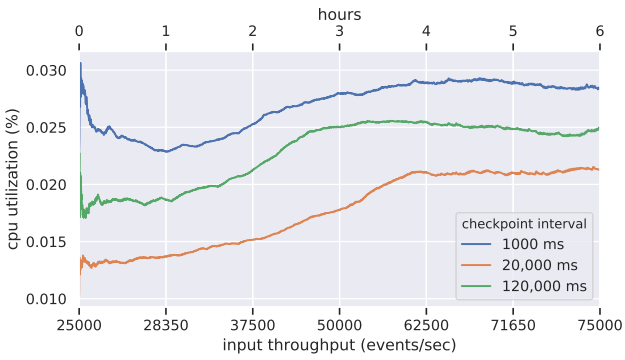

Fig. 4. TaskManager CPU utilization.

Fig. 4 and 5 show the average resource utilization for CPU and memory across all TaskManagers. Here we can see

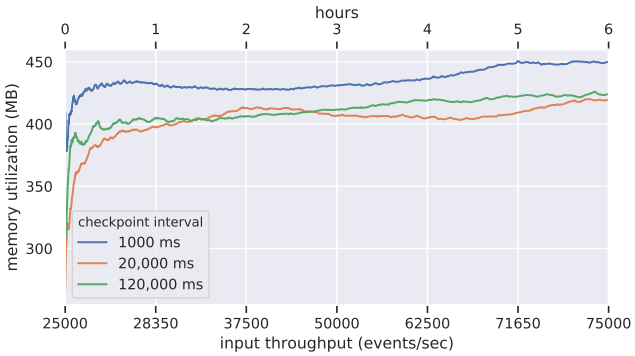

Fig. 5. TaskManager memory utilization.

how CPU and memory usage increases as input throughput increases, however, in both cases utilization is low and there is no need to provision more resources through configuration. The default setting for memory assigned to each TaskManager is 1 GB. Analysis of resource utilization for the JobManagers was likewise performed. As is to be expected, CPU and memory utilization was lower than the TaskManagers while following the same trend of the shorter the checkpoint interval, the more resources were consumed and this increases as input throughput increases.

VI. Conclusion

This paper presented an approach which allows for the effective testing of system configuration of critical IoT analytics pipelines in realistic conditions. For this, we assume a typical distributed architecture for critical IoT analytics pipelines and utilize containerization as well as container-orchestration in order to replicate instances of this architecture in parallel, each with their own configuration set. We showed how using such a testing approach in the production environment can capture the runtime behaviors of stream processing applications in order to investigate the individual performance of each configuration set. This was done by aggregating chosen metrics recorded over a defined number of testing rounds and then comparing them. Ultimately, the choice of which configuration set is the best performer should always consider pre-defined QoS requirements.

In the future, we would like to expand upon our approach in two ways. Firstly, we want to conducting experiments with failure scenarios and including critical IoT analytics applications from different domains in addition to smart city. Secondly, we want to research flexible methods for automatic parameter tuning and selection of optimal performing configurations. Nevertheless, this approach has already proven to be a helpful testing method and a usable tool.

Acknowledgments

This work has been supported through grants by the German Ministry for Education and Research (BMBF) as Berlin Big Data Center BBDC2 (funding mark 01IS18025A).

References

[1] Georgakopoulos, D., Jayaraman, P., Fazia, M., Villari, M., Ranjan, R., “Internet of Things and Edge Cloud Computing Roadmap for Manufacturing,“ IEEE Cloud Computing, 2016.
[2] Jin, J., Gubbi, J. Marusic, S., Palaniswami, M., “An Information Framework for Creating a Smart City Through Internet of Things“ IEEE Internet of Things Journal, 2014.
[3] Cheng, B., Longo, S., Cirillo, F., Bauer, M., Kovacs, E., “Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander,“ IEEE Big Data, 2015.
[4] Lom, M., Pribyl, O., Svitek, M., “Industry 4.0 as a Part of Smart Cities“, SCSP, IEEE, 2016.
[5] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy D., “Storm @Twitter”, ACM SIGMOD, 2014.
[6] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I., “Spark: Cluster Computing with Working Sets”, USENIX HotCloud, 2010.
[7] Carbone, P., Katsifodimos, A. Ewen, S. Markl, V. Haridi, S., Tzoumas, K., “Apache Flink: Stream and Batch Processing in a Single Engine”, IEEE Data Eng. Bull, 2015.
[8] De Francisci Morales, G., Bifet, A., Khan, L., Gama, J., Fan, W., “IoT Big Data Stream Mining“, SIGKDD, ACM, 2016.
[9] Shukla, A., Chaturvedi, S., Simmhan, Y., “RIoTBench: An IoT Benchmark for Distributed Stream Processing Systems“, CCPE, 2017.
[10] Janßen, G., Verbitskiy, I., Renner, T., Thamsen, L., “Scheduling Stream Processing Tasks on Geo-Distributed Heterogeneous Resources“, IEEE Big Data, 2018.
[11] Amini, S., Gerostathopoulos, I., Prehofer, C., “Big Data Analytics Architecture for Real-Time Traffic Control“, MT-ITS, IEEE, 2017.
[12] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J., “Large-scale cluster management at Google with Borg”, EuroSys, 2015.
[13] Kreps, J., Narkhede, N., Rao, J., and others, “Kafka: A distributed messaging system for log processing”, NetDB, 2011.
[14] Shvachko, K., Kuang, H., Radia, S., Chansler, R., “The Hadoop Distributed File System”, MSST, 2010.
[15] Lakshman, A., Malik, P., “Cassandra: a decentralized structured storage system”, SIGOPS, ACM, 2010.
[16] White, G., Nallur, V., Clarke, S., “Quality of Service approaches in IoT: Systemic Mapping“, The Journal of Systems and Software, 2017.
[17] Allen, S., Jankowski, M., Pathirana. P., “Storm Applied: Strategies for Real-time Event Processing“, Manning Publications Co., 2015.
[18] Fischer, L., Gao, S., Bernstein, A., “Machines Tuning Machines: Configuring Distributed Stream Processors with Bayesian Optimization“, IEEE Cluster, 2015.
[19] Jamshidi, P., Casale, G., “An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems“, arXiv:1606.06543, 2016.
[20] Bilal, M, Canini, M., “Towards automatic parameter tuning of stream processing systems“, ACM SoCC, 2017.
[21] Trotter, M., Liu, G., Wood, T., “Into the Storm: Descrying Optimal Configurations Using Genetic Algorithms and Bayesian Optimization“, IEEE FAS*W, 2017.
[22] Trotter, M., Wood, T., Hwang, J., “Forecasting a Storm: Divining Optimal Configurations using Genetic Algorithms and Supervised Learning“, IEEE ICAC, 2019.
[23] Shahriari, B., Swersky, K., Wang, Z., Adams, R., de Freitas, N., “Taking the human out of the loop: a review of bayesian optimization“, Technical report, 2015.
[24] Rasmussen, C., Nickisch. H., “Gaussian processes for machine learning (gpml) toolbox“, JMLR, 2010.
[25] McKay, M., Beckman, R., Conover. W., “A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code“, Technometrics, 1979.
[26] Vaquero, L., Cuadrado, F., “Auto-tuning Distributed Stream Processing Systems using Reinforcement Learning“, arXiv:1809.05495. 2018.
[27] Bu, X., Rao, R., Xu. C., “A reinforcement learning approach to online Web systems auto-configuration“, ICDCS, 2009.
[28] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C., “Chaos Engineering“, IEEE Software, 2016.