Self-Adaptation Based on Big Data Analytics: A Model Problem and Tool

Sanny Schmid[1], Ilias Gerostathopoulos[1], Christian Prehofer[1,2], Tomas Bures[3]

[1]Fakultät für Informatik
Technische Universität München
Munich, Germany
{schmidsa,gerostat,prehofer}@in.tum.de

[2]fortiss GmbH
Munich, Germany
prehofer@fortiss.org

[3]Charles University in Prague
Faculty of Mathematics and Physics
Prague, Czech Republic
bures@d3s.mff.cuni.cz

Abstract—**In this paper, we focus on self-adaptation in large-scale software-intensive distributed systems. The main problem in making such systems self-adaptive is that their adaptation needs to consider the current situation in the whole system. However, developing a complete and accurate model of such systems at design time is very challenging. To address this, we present a novel approach where the system model consists only of the essential input and output parameters. Furthermore, Big Data analytics is used to guide self-adaptation based on a continuous stream of operational data. We provide a concrete model problem and a reference implementation of it that can be used as a case study for evaluating different self-adaptation techniques pertinent to complex large-scale distributed systems. We also provide an extensible tool for endorsing an arbitrary system with self-adaptation based on analysis of operational data coming from the system. To illustrate the tool, we apply it on the model problem.**

Keywords-self-adaptation; Big Data analytics; model problem

I. Introduction

In this paper, we focus on self-adaptation in large-scale software-intensive distributed systems. As an example, consider a city traffic management system, which collects data from many sources like cars or traffic lights and then optimizes the traffic guidance. The main problem in making such systems self-adaptive is that their adaptation needs to consider the current situation (and possibly predictions on future situations) in the whole system. Due to the complexity of such systems, it is very difficult to explicitly plan for all eventual situations and model them accurately upfront. This means that most current approaches for adaptive systems which build an explicit model of the system [1]–[4] are not very suitable in our case.

The main, novel approach in this paper is to use the data generated by the system at runtime (operational data) for adaptation. Recent advances in Big Data technology permit the efficient storage and analysis of large amount of data both as they come (stream mode) and in batch mode [5]. In our approach, the stream of data coming from the distributed system is processed and used for adaptation.

Instead of creating an explicit system model, which is typically an abstraction to capture all essential states, we only use a simple model with the essential input and output parameters which is easy to construct at design time. Based on the operational data, we aim to optimize the system by different adaptation steps. For this purpose, we use evaluation functions to analyze the data and extract the suitable actions.

We provide the following two artifacts to substantiate our approach:

1. A model problem for adaptation in large-scale complex distributed systems, relying on Big Data analytics.
2. A tool for performing adaptation of a system based on analysis of large scale operational data at run time. The tool is based on established Big Data tools like Kafka and Spark. We employ a simple model based on input and output parameters and focus mainly on system optimizations based on optimization functions.

The main novelty of our artifacts is the seamless integration of self-adaptation with latest Big Data technology. This permits to evaluate large scale data sets at run time to effectively optimize systems based on adaptation.

As a sample application area for our model problem, we focus on a traffic application. We consider a smart navigation system that is deployed in a number of cars in a city, collects data from the cars' sensors (e.g. position, speed), and navigates the cars based on the current estimation of traffic in the streets. In the context of this running example, the main challenges connected with adaptation are as follows: How to design the system to minimize data collection from the cars so that the traffic predictions are still useful? How to design the system to use a different routing algorithm that scales better but has a bigger setup cost, based on the number of drivers that are present?

In this context, we provide a Big Data-based tool that makes it possible to specify how to experiment with different settings in the system and based on this specification, it drives the experiments and adapts the system.

The paper is structured as follows. Section II provides a brief overview of the Big Data tools relevant to our work. Section III describes the model problem, while Section IV provides the tool for performing Big Data-enabled adaptation. Section V describes challenges we faced and possible extensions of our work. Section VI briefly details on the structure of the provided artifacts (detailed instructions are packaged together with the software artifacts). Section VII discusses related work while Section VIII provides an overview of contributions.

II. Background on Big Data Tools

A. Kafka

Kafka is a distributed messaging system originally developed by LinkedIn, now an open-source project of the Apache foundation [6], [7]. It is intended for building real-time data pipelines with high throughout and low latencies.

In Kafka, a stream of messages of a particular type is defined by a topic. Producers publish messages to topics; the published messages are stored at a set of servers called brokers. Consumers subscribe to topics and pull messages from the brokers. Kafka supports both point-to-point delivery, where multiple consumers consume a single copy of all messages in a topic, and publish-subscribe, where each consumer receives its own copy of messages in a topic. Topics in Kafka are divided into multiple partitions; each broker (server in the cluster) stores one or more partitions, represented by a logical log (set of typically 1-GB-long files). Each message is addressed by its logical offset in the log. As a design decision for simplicity and low overhead, Kafka brokers are stateless: it is the consumers that need to keep track of the offset and send it to the broker when pulling new data. This has the interesting feature of a consumer being able to rewind to an old offset and re-consume data.

Kafka can scale horizontally by adding more brokers to a cluster. There is no central master broker; instead, consumers coordinate in a decentralized way using a distributed consensus service called Zookeeper [8].

B. Spark

Spark is a fault-tolerant computing framework for large clusters [9]. It was originally developed to support (i) interactive data exploration and analytics (a shortcoming of traditional map-reduce-based computations due to their high latency), and (ii) iterative jobs, where a function is repeatedly applied to a dataset—a common case in many multi-pass machine learning computations. Spark deals with both issues by keeping data in memory at each cluster node and avoiding the reloading of data from disk as much as possible.

To use Spark, developers write a *driver* program that connects to a cluster of workers. Driver programs can be written in Scala, Java, Python, or R. Spark comes with a number of libraries to support real-time SQL querying (Spark SQL), graph processing (GraphX), machine learning (MLlib) and stream analytics based on micro-batches (Spark Streaming [10]). In a Spark Streaming engine, data is typically injected from sources like Kafka or TCP sockets, processed using high-level functions over a pre-defined window (e.g. 1 sec) and outputted to filesystems, databases, or brokers.

III. Model Problem: Crowdsourced Navigation System

The reference problem embodied in the Crowdsourced Navigation system (CrowdNav), a hypothetical system inspired by the Waze app (https://www.waze.com). CrowdNav consists of a number of cars traveling in a city following the itineraries of their drivers and of a centralized navigation service. The city consists of a number of streets and intersections. Each car relies on the navigation service to receive a route, i.e. a series of streets to travel through, from its position to a destination. The navigation service employs a Dijkstra algorithm to calculate the shortest path to a destination in the map. The cost of the streets (edges) in the algorithm are calculated based on static information, i.e. the length and maximum speed of each street, and on dynamic information, i.e. the traffic on each street. The latter is estimated based on data being transmitted to the service by the cars via cellular technologies. Each car transmits the time it takes to navigate a street upon reaching its end (an intersection).

The efficacy of navigation based on real-time information in CrowdNav highly depends on the freshness of traffic information available to the navigation service. For example, in case of a traffic jam in certain streets, the service may be able to navigate some cars through longer but faster streets, if it has up-to-date information indicating low traffic in these streets. To obtain fresh data about some streets, the service may choose to route certain cars via different—possibly worse—routes than the ones calculated by the regular version of the Dijkstra algorithm in order to explore the traffic level and flow in these streets. We call the cars that are routed via exploration routes to collect up-to-date traffic information "explorers". Employing explorers may increase the efficacy of navigation (and therefore overall improve the system) at the expense of some potentially dissatisfied drivers.

As with any complex, real-life system with intricate dependencies and trade-offs, there is a need to tune the parameters of CrowdNav to maximize its value. Doing so at design time (e.g. via extensive simulations) is not a viable option, since there is high complexity and uncertainty regarding the operating conditions of the system, including the behavior of humans in the loop. What if some of the streets are blocked (due to an accident or traffic jam), making routing based on static data irrelevant? What if drivers do not follow the routes calculated by the navigation service? What if explorers get dissatisfied and stop using the service if they get highly suboptimal routes?

Ideally, the system should be aware of its goals (Section III.A), monitor its environment, and tune its parameters/knobs (Section III.B) at runtime, making CrowdNav an excellent case for self-adaptation.

A. Self-Adaptation Goals

The main goal of self-adaptation in CrowdNav relates to the non-functional requirement of optimizing user satisfaction. We provide here a number of metrics that can be used to quantify this abstract requirement.

Average of trip durations. A trip is defined as traveling along a set of streets from a start position to a destination. Each car makes several trips and measures their durations. Although different statistic measures can be evaluated on the set of durations we consider the *average of trip durations* as a representative quantity to be minimized in the system.

Average of trip overheads. A trip overhead is defined as the actual duration of a trip versus the theoretical case where routing is performed based only on static map data—length and maximum speed of each street. Ideally, this should be 1.

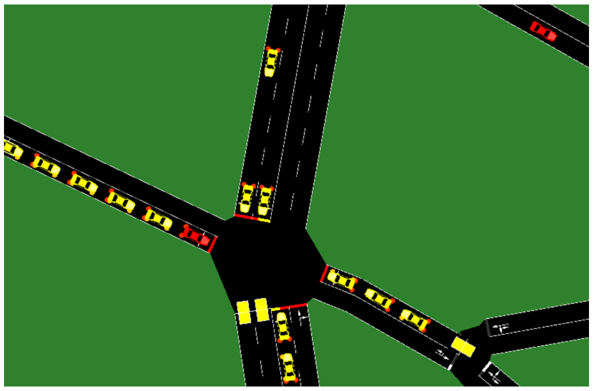

Figure 1. A snapshot of SUMO visualization. SUMO-controlled cars are depicted in yellow; CrowdNav-controlled cars in red.

In practice, it is always higher, as vehicles slow down to stop and accelerate again before reaching their maximum speed, which they might not be able to do in case of high traffic. Similar to average durations, we consider the *average of trip overheads* as representative metric; we have also used it in our experiments (Section IV). The advantage of using overheads over durations is that the first do not depend directly on street lengths, thus are directly comparable and can be better summarized by a single value (here, average).

Driver complaints rate. When a driver gets dissatisfied by distorted routes (the effect of being an explorer) they can file a complaint. The rate of complaints has to be minimized.

Driver drop-out events rate. Some drivers might never provide direct feedback in the form of a complaint in case they are dissatisfied, but may instead choose not to use CrowdNav's routing service anymore. Similar to driver complaints, the rate of such events should be minimized. Contrary to driver complaints, however, such events cannot be traced back to a system action, e.g. treating a car as explorer, in a straightforward way: a driver's impatience may grow over time or a driver may drop-out for completely different reasons.

***Acceptance rate for recommended routes*.** Having drivers not following the routes provided by the navigation service is an indication they are dissatisfied. The acceptance rate should of course be maximized.

B. *Self-Adaptation Knobs*

In CrowdNav, the main component that can be configured to optimize the system is the navigation service and specifically its routing algorithm, which is based on the following per-edge cost function:

$$cost = cost_{static} + cost_{dynamic} - benefit, \text{ where}$$

$$cost_{static} = w_{static} * (1 - freshness) * \frac{length}{maxSpeed}$$

$$cost_{dynamic} = w_{dynamic} * freshness * avgDuration$$

TABLE 1. ADAPTATION KNOBS IN CROWDNAV.

Name	Range	Description
$\boldsymbol{w_{static}}$	1..5	Importance of the static information is for routing
$\boldsymbol{w_{dynamic}}$	1..5	Importance of the dynamic information is for routing
$\boldsymbol{w_{benefit}}$	10..20	Importance of the exploration benefit is for routing
exploration percentage	0..30%	Importance of alternative (exploration) routes

$$benefit = w_{benefit} * (1 - freshness) * isExplorer$$

In the above equations, $freshness$ denotes the time since the street was last visited by any car; $avgDuration$ is average duration to traverse the street calculated from the durations transmitted by each car to the service. The $length$ and $maxSpeed$ are static information of the map.

In essence, when the service has up-to-date-enough information, routing relies more on the dynamic information on a street ($avgDuration$), otherwise it relies more on static information ($length$ and $maxSpeed$). If a car is an explorer, routing through streets not recently visited is prioritized.

The algorithm employs several tunable parameters or adaptation knobs (Table 1):

- w_{static} and $w_{dynamic}$ can be tuned to make the routing rely more on the static or the dynamic information, respectively.
- $w_{benefit}$ can be used to tune the degree of distorting the route of an explorer.
- *exploration percentage* can be used to control the number of routes which will be distorted for exploration reasons (when calculating these routes, the $isExplorer$ is set to 1 in the above equation).

C. *Relevance to Big Data*

CrowdNav can involve thousands of cars and drivers, each of them feeding the adaptation layer with data of their trip durations, overheads, complaints, and drop-out events at a potentially high rate. The adaptation layer should be able to cope with the high volume and velocity of incoming data in order to evaluate the metrics pertaining to adaptation goals in real-time and adapt the system according to the evaluation results. As such, CrowdNav lends itself as a self-adaptive system where Big Data analytics tools can be conveniently used in performing real-time goal evaluation at scale.

D. *Reference Implementaiton*

We have implemented CrowdNav using SUMO [11], the most comprehensive open-source solution for simulating urban mobility, and TraCI, a Python library that acts as interface to SUMO and allows for high-level control of traffic simulations.

In our experimental setup, shipped in this artifact, we are using a map of the medium-size German city of Eichstädt with

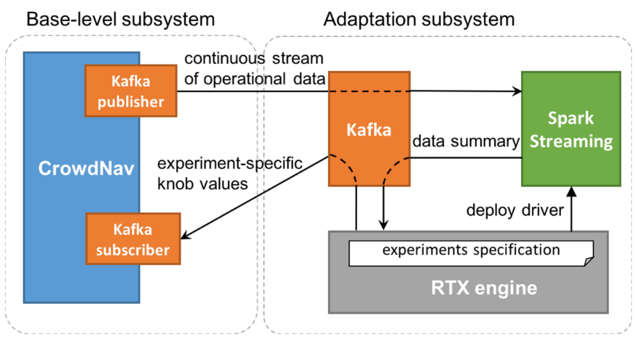

Figure 2. Overview of complete system architecture, including the RTX engine.

approx. 450 streets and 1200 intersections. We are simulating 600 cars that continuously travel from a randomly selected destination to another. Out of them, 60 cars (10% of all cars) belong to CrowdNav and receive routes from our custom router, implemented in Python, following the Dijkstra algorithm described in Section III.B. The rest of the cars represent the environment in our setting and are routed by the default SUMO router. Based on the *exploration percentage,* our custom router probabilistically treats a car as an explorer. Figure 1 shows a snapshot of the SUMO GUI that can used to observe and debug a simulation.

There is a single place in the code ("knobs.json" file) where the values of the knobs can be set. A user can simply change the value of a knob and run the simulation for an amount of time, observing the results in the visualization pane.

In addition, a user can use our tool, described in the next section, and let the system self-optimize by picking the best values for the knobs in a fully automatic way.

IV. Tool: Real-Time Experimentation

The Real-Time Experimentation (RTX) tool provided as part of this artifact allows for endorsing an arbitrary system with self-adaptation based on analysis of operational data coming from the system. The strength of this approach to self-adaptation is that it requires a very simple input-output model of the system to be provided at design time.

RTX is particularly useful in analyzing operational data that have the properties to be regarded as Big Data, i.e. high volume and velocity, although not restricted to this domain. Its strength lies in automatically and continuously adapting a system based on high level specifications. RTX also takes care of the tedious, time-consuming, and error-prone process of setting up Kafka and Spark and connecting them together. The user can then focus on the inputs and outputs of the analysis and on the adaptation logic.

Although we have been using CrowdNav as a testbed for our tool, RTX can be used with any other system that offers interfaces to collect data and enact changes to the system.

A. *RTX Concepts and Architecture*

A main concept in RTX is the *experiment*. An experiment involves (i) enacting a change to a running system, (ii) gathering a large enough amount of operational data from the system by observing the system for some time, and (iii) evaluating the result of the change based on the gathered data.

```
def state_initializer(state):
    state["count"] = 0
    state["avg_overhead"] = 0
    return state

def data_reducer(state, newData):
    cnt = state["count"]
    state["avg_overhead "] = (state["avg_overhead "] * cnt +
                                  newData["overhead "]) / (cnt + 1)
    state["count"] = cnt + 1
    return state

def change_event_creator(knobs):
    return knobs

def evaluator(state):
    return state["avg_overhead "]
```

Listing 1: Example of state initializer, data reducer, change event creator and evaluator functions for CrowdNav.

More concretely, RTX assumes the architecture depicted in Figure 2 for enacting changes and gathering data from the running system (*base-level subsystem* in the Figure). In it, a single Kafka broker acts both as *data provider* and *change provider*, mediating communication between the base-level subsystem and the adaptation subsystem and in particular the RTX engine, which includes all the RTX libraries and experiment definitions. The base-level subsystem continuously publishes a part of its operational data to a data provider topic. It also subscribes to a change provider topic to receive change directives (requests to change the values of one or more knobs). For illustration, when using RTX to adapt CrowdNav, a data provider topic can be "crowd_nav_trips", on which RTX receives trip overheads. A change provider topic can be "crowd_nav_commands", where RTX publishes changes in values of CrowdNav's knobs (Table 1).

To evaluate the result of a change in the system, four functions must be provided: a *state initializer*, a *data reducer*, a *change event creator* and an *evaluator.* State initializer initializes a local dictionary variable representing the state of the analysis. Data reducer calculates a summary of the incoming data stream and saves it to the local state. Change event creator is used to transform knob values into change directives. Evaluator receives the local state and calculates a number quantifying the satisfaction of the base-level subsystem's goals. For illustration, for CrowdNav, four possible functions, written in Python, are depicted in Listing 1. The data reducer calculates the cumulative moving average of trip overheads (second self-adaptation goal in Section III.A), and the evaluator simply returns it.

To reduce the amount of the incoming data on the data reducer, a *preprocessor* can be used. In RTX architecture, this is realized by a Spark Streaming engine. The preprocessor subscribes to the data provider Kafka topic, calculates a summary of the input data stream on a predefined sliding window, and periodically publishes it back to a summary Kafka topic. If a preprocessor is used, the incoming data on the data reducer comes from the summary topic, instead of the data provider one. When applied to CrowdNav, the Spark Streaming driver may pre-calculate the cumulative moving average of trip overheads (essentially replicating the data reducer of Listing 1).

Having the setting described above, different experiments can be run. Each experiment starts by initializing the local analysis state and requesting a change on the base-level subsystem via publishing the appropriate message to the change provider. It then applies the data reducer function to the summary topic to update the local state according to the incoming data. To avoid hysteresis effects (i.e. allow the effects of the change to take place), each experiment disregards a predefined number of Kafka messages on the summary topic. An experiment runs until a target number of Kafka messages has been successfully processed.

B. *User's perspective*

A user of RTX should provide a folder containing a Python file called "definition.py", and, in case a preprocessor is needed, a jar file with the Spark driver. The Python file has to include the following:

- The four essential Python functions: state initializer, data reducer, change event creator and evaluator (Listing 1).
- A configuration of Kafka producer, Kafka consumer, and Spark, corresponding to the change provider, data provider, and preprocessor, respectively (Listing 2).
- An execution strategy, which specifies how multiple experiments are to be performed. We are supporting three strategies: sequential, step and self-optimizer.

If the sequential strategy is used, a sequence of experiments has to be provided (Listing 3). In this case, each experiment is specified by (i) the changes it assumes to the knobs of the base-level subsystem (lines 2, 5) (ii) the number of Kafka messages that will be ignored in the beginning (lines 3, 6), and (iii) the number of Kafka messages to be processed until the end of the experiment (lines 4, 7).

The step strategy is essentially a succinct way of specifying sequential experiments (Listing 4). In this case, only the permissible bounds of the values of the knobs have to be provided, together with a step value (line 2). A sequence of experiments is generated by starting from the lower bound and increasing every time the value by the step until the value exceeds the upper bound. Each generated experiment inherits the number of Kafka messages to be ignored and the target number of messages to be processed from the step explorer specification (lines 3-4).

The self-optimizer strategy is also generating experiments (Listing 5). This time, only the permissible bounds of the values of the knobs have to be provided (line 2). The exact values of the knobs are set by the optimization process in the generation of experiments. Similar to the step strategy, each generated experiment inherits the number of Kafka messages to be ignored and the target number of messages to be processed from the self-optimizer specification (lines 4-5). For the optimization process, we are providing the option of Bayesian optimization using Gaussian processes [1] (line 3), which attempts to find the minimum value of an unknown function in as few iterations as possible [12].

```
configuration = {
   "kafka_producer": {
      "kafka_uri": "kafka:9092",
      "topic": "crowd-nav-commands",
      "serializer": "JSON"  # alternatives: raw, CSV, …
   },
   "kafka_consumer": {
      "kafka_uri": "kafka:9092",
      "topic": "crowd-nav-summary",
      "serializer": "JSON"  # alternatives: raw, CSV, …
   },
   "spark": {
      "submit_mode": "client_jar",  # alternatives: python, R, …
      "job_file": "CrowdNavSpark-assembly-1.0.jar",
      "job_class": "crowdnav.Main "
   }
}
```

Listing 2: Kafka and Spark configuration in RTX.

```
1.  experiments_seq = [
2.     { "knobs": { "exploration_percentage": 0.05 },
3.       "ignore_first_n_results": 1000,
4.       "sample_size": 5000 },
5.     { "knobs": { "exploration_percentage": 0.1 }
6.       "ignore_first_n_results ": 1000
7.       "sample_size": 5000 },
8.     …  # more experiments here
9.  ]
```

Listing 3: Example of sequential strategy.

```
1.  step_explorer = {
2.     "knobs": { "exploration_percentage": ([0.0, 0.3], 0.1) }
3.     "first_samples_to_ignore": 1000,
4.     "sample_size": 5000,
5.  }
```

Listing 4: Example of step strategy.

```
1.  self-optimizer = {
2.     "knobs": { "exploration_percentage": [0.0, 0.3] }
3.     "method": "gauss_process",
4.     "first_samples_to_ignore": 1000,
5.     "sample_size": 5000,
6.  }
```

Listing 5: Example of self-optimizer strategy.

Note that all strategies support specification of multiple knobs, allowing for multivariable optimization. Examples of such specifications are provided in the artifact.

To run RTX, a command-line utility is provided which reads the definition.py, executes a series of experiments, and creates a CSV file with knob values of the experiments that were performed and their results. It also plots the results into a scatter plot (one knob) or heat map (two knobs). The knob values that produced the best effect are not automatically applied on the base-level subsystem; this is the task of the user, who reviews the results and chooses whether to fix certain knobs and/or continue experimenting.

V. Discussion

A. *Experience and Lessons Learned*

Building the artifacts led us to interesting observations, which we believe are of general interest. We share them here.

[1] https://scikit-optimize.github.io/#skopt.gp_minimize

Controlling a SUMO simulation (C++) through TraCI Python interface is quite flexible and robust. Nevertheless, we had to make a number of workarounds to allow for custom routing and for having cars that move continuously in the map. In the current implementation, we are representing the entire SUMO map in Python and re-creating a car when it reaches its destination into a new car with the same ID, the reached destination as starting point and a route to a new randomly selected destination. To minimize the amount of interaction between SUMO and Python, we used event register functions.

Setting up Kafka via a pre-existing Docker image was a smooth experience. However, the existence of consumer-specific offsets in Kafka, enabling a consumer to "catch up" with missed messages, did not fit well to our setting. We actually needed to consume only fresh data upon starting an experiment. To do this, we chose to assign a new groupID to each experiment (effectively resetting the Kafka queue).

Finally, although we had originally planned to use Python for the Spark driver we decided to switch to Scala as it is generally better supported: using Python to process streams in parallel works only when setting up Spark on top of Hadoop YARN, whereas Scala works also with the standalone cluster deployment mode of Spark.

B. Possible Extensions

RTX is a project under active development. We plan to extend its functionalities in several directions, namely:

Support for dynamically calculated sample size. Instead of having the user specify a minimum number of messages to be processed for each experiment, the tool itself could infer this number from the minimum sample size for rejecting a null hypothesis in a statistical test. In that case, the user would need to specify their hypothesis on the result (e.g. avg_overhead < 2), the target confidence level (e.g. 95%) and the maximum number of messages. The tool would stop processing data once it has enough evidence to reject the null hypothesis or if the maximum number of messages is reached (thus not rejecting the hypothesis).

Support for specification of multiple data sources. In many cases, evaluation functions are composed of different metrics that need to be separately quantified by incoming data. For example, in CrowdNav, we would ideally like to calculate the overall utility as a function of both the average overhead in trip durations and average (or close to worst case, e.g. 90% percentile) of drivers' complains and drop-out event rates. This calls for (i) extending RTX to support multiple data sources, i.e. data provider Kafka topics, and multiple data reducer functions, (ii) extending CrowdNav with probes (in Python) that emit data other than trip overheads, to support the rest of the self-adaptation goals described in Section III.A.

Support for other domains apart from Big Data. We would like to test the generality of the RTX concepts by considering additional data and change providers and different preprocessors, e.g. Flink. Also, to tailor it to the IoT domain, we plan to support MQTT as data and change provider (similar to Kafka) and HTTP/CoAP server as change provider.

VI. Artifact Structure

The artifacts described in this paper are available at http://dx.doi.org/10.4230/DARTS.3.1.5. The artifacts are available as a single archive which contains the source code, together with installation and usage instructions. We have also included a pre-configured virtual machine image and two Docker containers.

VII. Related Work

Big Data approaches have been proven important in learning and analytics with massive data [13]. However, they have not yet been used in self-adaptation research.

In particular, it is surprising that there seems to be a vacuum in approaches that base adaptation on Big Data analytics (which is the primary aim of RTX). We scouted all papers of SEAMS 2010-2016, ICAC 2010-2016, BIGDSE 2015, 2016 (it did not exist before 2015), SASO 2010-2016, and ACM TAAS 2010-2016 for publications that would hint in their titles the use of big data technologies for driving adaptation. When a title mentioned some big data technique or technology, we reviewed the whole paper. Though there are a number of approaches that use adaptation to optimize the Big Data stack (typically allocation of map-reduce and shuffling in Hadoop), we did not find any approach that would use Big Data to run analytics for the sake of adaptation. We obtained similar results by searching for "self-adaptation", "big data", "hadoop", "spark", etc. in Google Scholar.

Focusing on works related to CrowdNav, there is research on traffic optimization (including routing vehicles [14]) and on using Big Data technologies to perform analytics over traffic data [13], [15]. From the perspective of artifacts usable for experimenting with adaptation, however, there is a lack of existing works. Research on vehicular traffic typically uses existing datasets of observed traffic, which is however unusable for experimenting with adaptation since the datasets do not reflect the adaptation of the system. This is alleviated by traffic simulation tools (mainly MATSim and SUMO) and concrete model problems and frameworks for adaptation built around them (e.g. DEECo [16]) or on top of dedicated simulators (e.g. ADASIM [17]). Though these are also suitable for experimenting with Big Data-driven adaptation, CrowdNav provides readily addressable decision problems suitable for the use of Big Data analytics. It also provides binding to Kafka to produce observations and accept adaptation commands. Finally, although CrowdNav uses SUMO, it does not depend on SUMO features as it is entirely written in Python and can easily extended with custom logic—e.g. types of drivers (patient/impatient, etc.).

VIII. Contributions

In conclusion, RTX is to our knowledge the only framework of its kind that provides configurable adaptation of an application based on Big Data analysis (exploiting Kafka and Spark). Combined with CrowdNav, which provides a model problem and a testbed of Big Data-driven adaptation, the artifacts presented in this paper make an interesting first step towards experimenting with Big Data-driven adaptation.

Acknowledgements

This work has been partly funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie as part of the TUM Living Lab Connected Mobility Project and partly sponsored by the German Ministry of Education and Research (BMBF) under grant n° 01Is16043A.

References

[1] D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue, "Learning Revised Models for Planning in Adaptive Systems," in *Proceedings of the 2013 International Conference on Software Engineering*, Piscataway, NJ, USA, 2013, pp. 63–71.

[2] C. Ghezzi, J. Greenyer, and V. P. L. Manna, "Synthesizing dynamically updating controllers from changes in scenario-based specifications," in *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2012, pp. 145–154.

[3] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic, "PLASMA: a plan-based layered architecture for software model-driven adaptation," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 467–476.

[4] A. Filieri, C. Ghezzi, A. Leva, M. Maggio, and P. Milano, "Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements," in *Proc. of ASE '11*, 2011, pp. 283–292.

[5] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2015.

[6] "Kafka website," 17-Jan-2017. [Online]. Available: https://kafka.apache.org/.

[7] J. Kreps, N. Narkhede, J. Rao, and others, "Kafka: A distributed messaging system for log processing," in *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB'11)*, 2011, pp. 1–7.

[8] "Zookeeper website," 17-Nov-2017. [Online]. Available: https://zookeeper.apache.org/.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, Berkeley, CA, USA, 2010, pp. 10–10.

[10] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," 2013, pp. 423–438.

[11] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent Development and Applications of SUMO - Simulation of Urban MObility," *Int. J. Adv. Syst. Meas.*, vol. 5, no. 3&4, pp. 128–138, Dec. 2012.

[12] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," *ArXiv12062944 Cs Stat*, Jun. 2012.

[13] J. Yu, F. Jiang, and T. Zhu, "RTIC-C: A Big Data System for Massive Traffic Information Mining," in *2013 International Conference on Cloud Computing and Big Data*, 2013, pp. 395–402.

[14] V. Pillac, M. Gendreau, C. Guéret, and A. L. Medaglia, "A review of dynamic vehicle routing problems," *Eur. J. Oper. Res.*, vol. 225, no. 1, pp. 1–11, Feb. 2013.

[15] Y. Lv, Y. Duan, W. Kang, Z. Li, and F. Y. Wang, "Traffic Flow Prediction With Big Data: A Deep Learning Approach," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 2, pp. 865–873, Apr. 2015.

[16] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, and F. Plasil, "An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems," in *SEAMS'15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015.

[17] Jochen Wuttke, Yuriy Brun, Alessandra Gorla, and Jonathan Ramaswamy, "Traffic Routing for Evaluating Self-Adaptation," in *Proc. of SEAMS '12*, 2012, pp. 27–32.