SpecK: Composition of Stream Processing Applications over Fog Environments

Davaadorj Battulga, Daniele Miorandi, Cédric Tedeschi

SpecK: Composition of Stream Processing Applications over Fog Environments

Davaadorj Battulga[1,2], Daniele Miorandi[2], and Cédric Tedeschi[1]

[1] Univ Rennes, Inria, CNRS, IRISA, Rennes, France
[2] U-Hopper, Trento, Italy
{davaadorj.battulga, daniele.miorandi}@u-hopper.com
cedric.tedeschi@inria.fr

Abstract. Stream Processing (SP), i.e., the processing of data in motion, as soon as it becomes available, is a hot topic in cloud computing. Various SP stacks exist today, with applications ranging from IoT analytics to processing of payment transactions. The backbone of said stacks are Stream Processing Engines (SPEs), software packages offering a high-level programming model and scalable execution of data stream processing pipelines. SPEs have been traditionally developed to work inside a single datacenter, and optimised for speed. With the advent of Fog computing, however, the processing of data streams needs to be carried out over multiple geographically distributed computing sites: Data gets typically pre-processed close to where they are generated, then aggregated at intermediate nodes, and finally globally and persistently stored in the Cloud. SPEs were not designed to address these new scenarios. In this paper, we argue that large scale Fog-based stream processing should rely on the coordinated composition of geographically dispersed SPE instances. We propose an architecture based on the composition of multiple SPE instances and their communication via distributed message brokers. We introduce SpecK, a tool to automate the deployment and adaptation of pipelines over a Fog computing platform. Given a description of the pipeline, SpecKcovers all the operations needed to deploy a stream processing computation over the different SPE instances targeted, using their own APIs and establishing the required communication channels to forward data among them. A prototypical implementation of SpecK is presented, and its performance is evaluated over Grid'5000, a large-scale, distributed experimental facility.

Keywords: stream processing, deployment, geographically distributed platforms

1 Introduction

Stream Processing (SP) is a major research theme within the general area of big data infrastructures and applications. Practitioners, who need to deploy SP pipelines, are presented with different options, rather mature, in terms of available (typically open source) software stacks. The cornerstones of these stacks

are stream processing engines (SPEs), such as Storm [32], Flink [7] and Spark streaming [33]. SPEs generally provide two main features: i) a *high-level programming model*, allowing developers to describe and code the intended behaviour, and ii) a *scalable execution model*, implemented by software able to deploy and monitor the application at run time.

Programmers define SP applications by combining operations to be applied on an incoming stream in a certain order. This combination can be linear (commonly referred to as a pipeline), but, more generally can be represented by a directed acyclic graph (DAG), whereby each vertex represent a single operation. Once implemented, such a program is deployed on the underlying computing infrastructure by the SP engine, thereby becoming a running *job*. The engine deploys the job over the compute nodes of the underlying infrastructure, trying to optimize application's throughput and resource usage. The main component supporting this deployment is commonly referred to as the *Job Manager*.

Datacenters, or more generally geographically-restricted infrastructures connecting compute nodes through a high-speed network are the natural target to deploy such jobs. Yet, with the advent of fog computing [19], we moved to a platform model made of a possibly large number of geographically distributed computing resources. This makes it difficult for a single Job Manager to remain efficient, due to the difficulties in maintaining an updated view over all possible available resources. The net result is that SPEs have some limitations when applied to fog computing scenarios, whereby data need to be processed in a coordinated fashion over a geographically distributed infrastructure. In a typical fog application, part of the processing is carried out directly close to the data origin (at the edge), aggregation steps are carried out at intermediate nodes (where data from different edges are joined), to be finally post-processed and stored in a more stable platform such as a Cloud [19]. This limitation calls for new execution models for SP applications over fog infrastructures. Until now, most works tackling this issue based their solution on revising scheduling policies by injecting some latency-awareness and some form of hierarchy into it. While such approaches are interesting, they present some limitation; furthermore, most of them stay at the prototype level.

This paper explores a radically different approach. Instead of revising existing SPEs, we advocate for a federated stream processing platform, able to combine multiple standard Job Managers (typically those provided by Flink, Storm or Spark Streaming), each one being responsible for the management of a geographically-restricted (local) portion of the infrastructure. Adopting this idea requires to revisit the traditional programming and execution models of SPEs. Considering geographically-distributed infrastructures will not only extend the range of computing platforms on which to deploy SP applications, but will also pave the way to the construction of SP applications through the composition of a set of ready-made SP Jobs. Similarly to the more traditional *service composition*, the notion of SP job composition carries the idea that developing complex SP applications out of the blue is not a reasonable option anymore. Such complex-

ity should be handled by a higher-level programming model: the composition of existing SP jobs into a new *composite* SP application.

Let us explore the example of a road traffic monitoring application: the road traffic is sensed locally and a first cleaning of the data and the computation of real-time statistics about the traffic can be done locally (for instance to be offered to people in this area). Yet to be exploited further (e.g., for statistical purposes and to support the design and monitoring of transportation policies), data need to be aggregated at the regional/national level, which is typically done at a centralized Cloud-based location. This pipeline is composed of jobs that will strongly benefit from running over different sites: cleaning and local statistics at edges, and global statistics in a centralized Cloud. Finally, the user is at risk that those different jobs exist but were developed using different SPEs, in which case, there is a need to be able to have heterogeneous jobs composed.

This paper proposes a novel programming model for SP, based on the composition of existing SP Jobs, and their execution over a geographically-distributed computing infrastructure. We devise the SpecKframework bringing this proposal into reality: based on the description of the composite application to be deployed, SpecKstarts each job composing the application over the resources of one computing site. Our assumption is that each computing site is equipped with an instance of a stream processing engine (a Flink Job Manager) able to deploy jobs over the local computing resources, and a message broker managing the needed message queues to transfer data and control messages between sites. SpecK provides two core APIs: the **Job Management API** focuses on the management of a single job. It allows to start, modify, and delete jobs in a unified fashion, regardless of the targeted SPE instance. The **Composition Management API**, given the description of a complex application composed of several jobs for which the code is available, deploys each job on the SPE instance specified for this job. This API supports dynamic on-the-fly reconfiguration/adaptation of the composition: The user simply needs to submit a new version of the description and SpecK will trigger all changes needed by contacting the SPE instances running jobs affected by the modification. SpecK was prototyped and deployed over the Grid'5000 large-scale testbed [2], where we evaluated its performance using real traffic traces.

Section 2 positions this work with respect to the relevant state of the art. Section 3 presents the programming and execution models of SpecK and details its usage, architecture and internals. Section 4 reviews the experimental results obtained over Grid'5000, focusing on scalability and on the ability of SpecK to bring the benefits of the Fog into reality when deploying SP applications at large scale. Section 5 concludes the paper, outlining a roadmap for the further development and integration of SpecK

2 Related Work

Stream processing is gaining momentum, as application domains such as smart cities and the Internet of things are becoming mainstream. Stream processing

engines represent the software response to the need for real-time analysis of large amount of data produced at a high rate [7, 32, 33].

Stream processing engines have adopted different programming models that can get categorized into two coarse-grain families. Low-level SP programming models, as for instance implemented by Storm [32], requires the user to code the application as an explicit directed acyclic graph of operators, the code of each of these operators being also left at the charge of the developer. While relatively cumbersome, this programming models brings about more flexibility and allows the user to define any operation. Higher-level SP programming models, as implemented by the Flink Datastream API [7], allows to express easily most pipelines by chaining predefined Flink operations in cascade.

Because we target environment where multiple running SPEs collaborate over a geographically-dispersed infrastructure (say one Flink *Job Manager* responsible for deploying and managing jobs over Cluster i, one Storm *Nimbus* responsible for deploying and managing jobs over Cluster j, *etc.*), we need to define programming abstractions able to express compositions of jobs, possibly running over different clusters and their dependencies). SpecK, presented in Section 3, given such a description, will be able to deploy and dynamically adapt such a composite application over such an environment.

A lot of work went into finding ways to optimize the deployment and placement of an application's operators over a computing platform, especially for cluster environments which are the natural playgrounds for traditional SPEs [16]. Yet, with the advent of Cloud computing, stream processing had the opportunity to deal with less constrained computing platforms and scale up and down dynamically as the velocity of the stream evolves [9, 13, 15, 23].

With the advent of Fog computing [3, 19], we are currently witnessing the emergence of works that will shape the fourth generation of stream processing platforms [4]: Stream processing is becoming highly distributed and deployed over hybrid Edge-Cloud platforms with a strong incentive to move processing at the edge where possible [24]. First, this leads to the development of light processing systems especially designed for the edge and its limited computing power [1,10,12,18,24]. These systems can be seen as lightweight SPEs. Choosing between different SPEs is not in the scope of the present work: the SpecK framework is designed so as favor SPE-agnosticism, so the execution of a composite application can be shared over SPE instances relying on different stacks.

A second series of research works was born from the need to consider Fog-like environments as the next natural playground for Stream Processing. They investigate scheduling strategies to place operators composing applications over geographically distributed compute resources, pursuing different metrics to optimize. Cardellini et al. [8] introduced a QoS-aware distributed scheduling algorithm taking into account the heterogeneous network capacity of the targeted platform. Frontier [22] explores strategies to optimize the performance and resilience of edge processing platforms for IoT, by dynamically routing streams according to network conditions. Planner [25] automates the deployment over hybrid platforms, taking decisions on what portion of an application graph should

be taken care of at the edge, and what portion should stay in the Cloud, trying to minimize the network traffic cost. The work in [29] exhibits similar objectives while focusing on specific yet very common families of graphs found in data stream analytics, namely series parallel graphs. These works focus on modelling the placement problem and propose strategies to optimize certain metrics, statically or dynamically, they do require to modify the schedulers and deployers at the core of existing stream processing engines. While these works show the great vitality of the domain, they are mostly performance oriented. In this sense, they are quite orthogonal to the present work: SpecK does not intend to provide new scheduling strategies, but advocates for an alternative way to program and run stream processing applications in the Fog, through composition.

Automating software deployment over large scale platform is not new in itself. Tools have been proposed for platforms such as grids [11] or clouds [21]. Yet the problem of deploying stream processing platforms over Fog architectures is still widely open. To our knowledge, only few works have been presented on the topic. R-pulsar provides a user-level API for operator placement [14]. R-pulsar offers a programming model similar to Storm, but where the user can choose what operator has to be placed at the edge, and what operator has to be placed in the Cloud. Then, the framework decides on what precise node to place the operator. Also, standardizing the way to benchmark Fog-deployed data stream processing applications is explored in [27, 28].

To our knowledge, our closest related work is E2CLab [26]. E2CLab is a framework easing the deployment of SP applications over platforms interconnecting the whole range of possible computing resources, from IoT devices to HPC clusters. E2CLab relies on a high-level description of the whole deployment process, from the installation of the stacks to the execution of the jobs, thus facilitating large scale experiments over such platforms. SpecK differs from E2CLab in the sense that while E2CLab is a tool for experimenters and covers the whole deployment process, SpecK is end-user oriented and provides simplified interfaces for the description of the application. Also, while E2CLab targets complex initial deployments only, SpecK also supports on-the-fly adaptation.

3 SpecK: an SPE Coordinator

SpecK deploys and dynamically adapts stream processing applications built as a composition of independent stream processing jobs running over different stream processing stacks. The architectural framework it relies on provides two user interfaces, one to abstract out the details of the deployment of a single job, whatever its target SPE stack, one to coordinate the initial deployment and subsequent user-driven adaptations of compositions over multiple independently running stacks. Section 3.1 describes SpecK targeted underlying platform. Then, Section 3.2 adopts the user's viewpoint and describes SpecK usage. Finally, Section 3.3 exhibits a SpecK software prototype and its internals.

3.1 Targeted platform

The present work targets infrastructures gathering geographically-dispersed computing resources, resources being grouped into what we refer to as a *computing site*. A computing site is typically a set of tightly coupled compute nodes, such as a cluster of small single-board computers located at the edge of the network. A larger datacenter, at the other end of the spectrum, can be seen as one site. All those sites are more and more aggregated into what the emerging *Edge-to-Cloud continuum* [5,6,20], the final objective being to be able to operate such a continuum in a unified manner. SpecK participates to this objective, focusing on stream processing applications.

Each site includes a running instance of a stream processing stack such as Storm, Flink or Spark streaming. This means that an orchestrator, commonly referred to as the *Job Manager*, is responsible to distribute the workloads submitted over the compute nodes of the site it is responsible of. The SPE instance constitutes the first software element we assume to be present on the sites. The second elements answer the need to link sites together so as to build the continuum: as SPE engines will be responsible only for portions of the applications deployed, different portions will run on resources of different sites. These jobs having dependencies, nodes from one site will have to communicate with nodes on another site. Inter-site communication is typically handled by Message Brokers (MB) such as Mosquitto [17], ActiveMQ [30] and Kafka [31].

Such an infrastructure is depicted in Figure 1: each site, of different size reflecting its computing power, is equipped with its own instance of stream processing engine and message broker. As mentioned, some of these sites can be referred to as *Edges* (small sites in the picture) and will typically use edge-optimized stream processing engine such as Edgent [1]. One of these sites can be a *Cloud* (the bigger site in picture 1) and will be typically equipped with Cluster-ready stream processing engines such as Storm, Flink or Spark Streaming. We assume that direct communication between the sites is always possible. Security constraints that may appear fall outside the scope of the paper. Also, the choice of the specific message broker used locally at each site depends on local requirements and user preferences and is not discussed here.

As it is detailed in the following, SpecK acts as a coordinator between those sites to deploy composite SP applications over the whole infrastructure. The following sections review its usage, architecture and internal mechanisms.

3.2 SpecK usage

SpecK acts as a coordinator between SPE instances, based on the user's input, as depicted in Figure 2. It can be seen as a wrapper on top of a pool of available running SPE instances, to be exploited as specified by the user in its application's description. The interaction between SpecK and the users are one-way: the user pushes the description to SpecK. The interactions between SpecK and the SPE instances go in both directions, typically through HTTP.

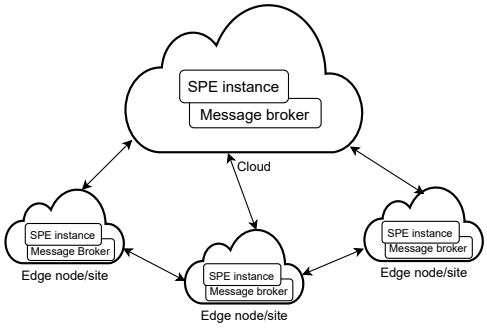

Fig. 1. SpecK targeted platform.

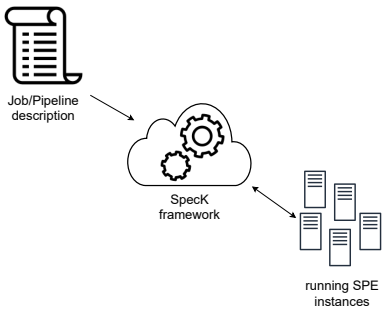

Fig. 2. SpecK overview

SpecK offers two interfaces to users. Firstly, it provides a **Job Management API**, a restful interface abstracting out the details of the flavour of the underlying specific Job Manager API: based on a simple description of a stream processing job to be deployed, this API deploys the job on whatever SPE is targeted for this job. It also supports adaptation: when the description of a job changes, this interface can stop, start and move jobs individually, again hiding the specifics of the underlying SPE. Secondly, on top of the Job Management API, SpecK provides the **Composition Management API**, able to manage a composition of jobs, typically expressed as a DAG of individual jobs. While the traditional granularity of DAGs in stream processing is an *operator*, SpecK handles DAGs of jobs, that themselves are internally possibly composed of several operators. The operators composing each job are not considered individually: each job is a black box with an input source and an output destination.

3.2.1 Job Management API

Within SpecK a job is described by three elements: i) the code it runs, ii) the SPE instance it runs on, and iii) its data source and sink. More precisely, a job description will contain the following elements. Firstly, the elements related to the job itself, namely: i) a (unique) **job name**, ii) an **entry class** which indicates the main class of a job's code, and iii) the **job path** which specifies the local path where to find the code of the job (typically a bytecode archive). Secondly, the SPE instance where to deploy the job needs to be specified, through the **SPE address**, represented by the IP address and the port of its Job Manager. Finally, information regarding the input and output data of the Job needs to be given: the two **message brokers** represented by their respective IP addresses and ports, managing its data source and sink respectively, and the two **data topics** identified by their respective names, where to read the incoming stream and where to write the outgoing stream. Given this description of the job, typically encapsulated into a JSON description, the user can issue the following commands:

– `POST /jobs` - deploys and registers the job described
– `GET /jobs` - lists all running jobs
– `GET /jobs/<job_name>` - gets the details of job `job_name`
– `DELETE /jobs/<job_name>` - deletes the job `job_name`

The Job Management API is an SPE-agnostic block to start, get the status of and stop jobs, on top of which, job migration and monitoring can be implemented them. In particular, it paves the way for higher-level management programs such as the pipeline coordinator described in the next section.

3.2.2 Composition Management API

Let us now focus on the second interface provided to the users and which allows them to manage job compositions over geographically-distributed platforms. Let us consider the deployment of a simple composition, to be modified in a second step. Figure 3 illustrates the initial graph: the composition is a pipeline composed of four jobs, each to be deployed over a different site.

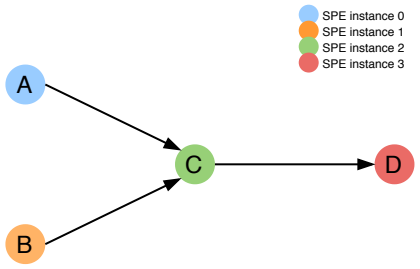

Fig. 3. Initial pipeline

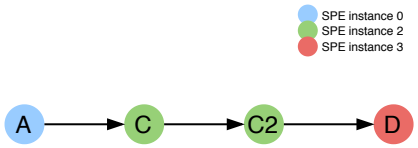

Fig. 4. Adapted pipeline

More precisely, jobs `A` and `B` read their respective input streams from two different data *topics* managed by two distinct message brokers. They both have their output sent to a third topic managed by the broker of the third site. The third topic is read by Job `C`, which, on its turn, processes the data and sends the results via a message queue managed by one last message broker on Site 3 where the fourth Flink instance hosts Job `D`. Listing 1.1 gives the description of this pipeline as to be sent to the Composition API. We observe, for each job composing the pipeline, the elements mentioned in Section 3.2.1. As further described in Section 3.3, SpecK coordination module parses this file and, relying on the Job Management API, deploys the jobs as specified by the user. Note that, even if a job can read from a single input topic and write to a single output topic, by having multiple jobs writing their output into a common topic, any DAG can be specified.

Let us assume that the user, at some point, wishes to modify the pipeline for that of Figure 4. It means that i) Job `B` gets removed, ii) Job `C2` appears, as an extra processing step between `C` and `D`. Note that, consequently, SPE instance 1,

which was hosting Job `B` is no longer part of the instances supporting the pipeline. `A`, `C` and `D` do not move. Let us assume that the user wants `C2` to be grouped with `C` on SPE instance 2. Job `C` needs to be modified so as to redirect its output stream to Job `C2`. These changes are highlighted in Listing 1.2. Job `B` disappears, while job `C2` (in green) is introduced. The information for the outgoing stream of `C` is modified (in red in Listing 1.1 and in cyan in Listing 1.2).

```
---
jobs:
  - job_name: A
    spe_address: http://172.16.39.7:8081/
    source_broker: tcp://172.16.39.7:1883
    sink_broker: tcp://172.16.192.18:1883
    source_topic: T-1
    sink_topic: T-C-filter
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-1.jar

  - job_name: B
    spe_address: http://172.16.48.8:8081/
    source_broker: tcp://172.16.48.8:1883
    sink_broker: tcp://172.16.192.18:1883
    source_topic: T-2
    sink_topic: T-C-filter
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-2.jar

  - job_name: C
    spe_address: http://172.16.192.18:8081/
    source_broker: tcp://172.16.192.18:1883
    sink_broker: tcp://172.16.177.7:1883
    source_topic: T-C-filter
    sink_topic: T-D-merger
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-3.jar

  - job_name: D
    spe_address: http://172.16.177.7:8081/
    source_broker: tcp://172.16.177.7:1883
    sink_broker: tcp://172.16.177.7:1883
    source_topic: T-D-merger
    sink_topic: T-D-total
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-4.jar
```

Listing 1.1. Initial pipeline.

```
---
jobs:
  - job_name: A
    spe_address: http://172.16.39.7:8081/
    source_broker: tcp://172.16.39.7:1883
    sink_broker: tcp://172.16.192.18:1883
    source_topic: T-1
    sink_topic: T-C-filter
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-1.jar

  - job_name: C
    spe_address: http://172.16.192.18:8081/
    source_broker: tcp://172.16.192.18:1883
    sink_broker: tcp://172.16.192.18:1883
    source_topic: T-C-filter
    sink_topic: T-C2-filter
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-3.jar

  - job_name: C2
    spe_address: http://172.16.192.18:8081/
    source_broker: tcp://172.16.192.18:1883
    sink_broker: tcp://172.16.193.22:1883
    source_topic: T-C2-filter
    sink_topic: T-D-merger
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-4.jar

  - job_name: D
    spe_address: http://172.16.177.20:8081/
    source_broker: tcp://172.16.193.22:1883
    sink_broker: tcp://172.16.193.22:1883
    source_topic: T-D-merger
    sink_topic: T-D-total
    entry_class: package.ExampleJob
    job_path: /usr/src/app/jars/test-4.jar
```

Listing 1.2. Adapted pipeline.

3.3 SpecK architecture and internals

Figure 5 depicts the components of the SpecK architecture. It is composed of four interrelated components.

The client typically submits a complete pipeline of jobs by invoking the *composition management API* and passes it the pipeline description file. The composition API relies on the *pipeline coordinator*, a Python-based component which generates, for each job of the pipeline, the HTTP queries to be transmitted to the Job Management API described, as if they were coming directly from the

user. The Job Management API is a REST API and was implemented using the Flask Python web framework.[3]

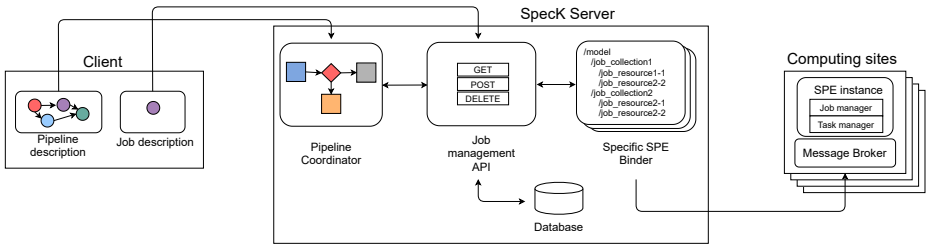

Fig. 5. SpecK software architecture.

The Job Management module is connected to the last two components: a database storing the state of the pipeline currently deployed, and a set of binders to different SPEs and MBs to make SpecK generic. When the Job Management API deploys a job, it records its description into the database, so when the user submits a modified version of the description of a pipeline, SpecK compares the running jobs described in the database with the newly submitted one and triggers the needed removals and introductions of jobs.

The actual deployment of jobs rely on SPE *binders*, each of them being able to communicate with a particular SPE flavour (Flink, Storm, *etc.*). As each SPE has its own API, basic commands such as starting, getting the status or stopping a job can differ depending on the specific SPE in use. SPE binders abstracts out this variability by taking care of formatting queries correctly for each SPE technology. In other words, SpecK also acts as a client sending queries to the Job Managers of available SPE instances on the sites of the infrastructure. Upon receiving the modified version of an existing pipeline, the same interactions take place. The coordinator compares the incoming jobs' arguments with the existing jobs description stored in the internal database and requests the status of jobs running on SPE instances via the Job Management API. It then decides whether jobs should be migrated to other SPE instance or not. Note that, doing so, only the necessary actions are performed. For instance, jobs that do not need to be migrated to another instance allow SpecK to save the cost of re-uploading the job. Again, all these movements are enforced by the Job Management API by communicating with the SPE instances using their specific APIs.

Remind that the sites composing the Fog are typically equipped with their own SPE and message broker instances. While this is not mandatory, using message brokers along with SPE in each instance not only allows to distribute the load of transmitting data between the computing sites, but also avoids unnecessary traffic between instances: the traffic between two jobs placed at the same site is kept local.

[3] https://flask.palletsprojects.com

4 Experiments

The experimental campaign conducted has several objectives. Firstly, the scalability of the solution itself, the time it takes to deploy and modify large pipelines is evaluated in Section 4.1. Secondly, in Section 4.2, we place ourselves on top of a hybrid edge-cloud platform to show that SpecK can bring the benefits of such platforms into reality. The prototype of SpecK includes specific binders to Apache Flink SPE and to Mosquitto MB. Thus, we assume in the following that the computing sites are equipped with Flink and Mosquitto. This uniformity is desirable as it allows to focus on the validation of the functionalities and evaluating the performance of SpecK without having to estimate the influence of the specificity of SPE/MB technologies. The experiments were conducted over Grid'5000, a large-scale geographically-distributed computing platform bringing together thousands of computing cores grouped in clusters located in 8 different computing sites in France and Luxembourg [2].

4.1 Scalability and overhead

We first evaluated the ability of SpecKto quickly deploy and modify large pipelines. These experiments were conducted on 6 instances distributed over 3 geographically-distant computing clusters (respectively located in Nantes, Lyon and Luxembourg). Each cluster includes two instances. The measured average network latency between clusters was of 18.6ms. Each instance run its own Mosquitto MQTT broker and an Apache Flink Job Manager. Each Flink instance managed 24 to 32 Task slots to place the jobs on the workers of the cluster, depending on the number of cores available on the compute nodes, which made, depending on the experiment, a total of between 166 and 192 available task slots on the whole platform including the three distant clusters. We generated simple chains of between one and one hundred jobs. Each job consisted of a trivial SP program receiving an input from the previous job in the chain and passing it to the next one. The jobs composing these pipelines were deployed uniformly at random across the instances.

We measured the time elapsed for the initial deployment, modification, and removal of jobs, given in Figure 6. For each configuration considered, 10 runs were executed, and the performance displayed was averaged over said 10 runs. We also report min and max over the set of runs whenever relevant. The green curve shows the time spent to initially deploy pipelines of sizes ranging from 1 to 100. The blue curve shows, once 100 jobs have been deployed, the time it takes to modify a number of jobs ranging from 1 to 100. Finally, the red curve shows the time taken to delete a variable number of jobs.

The main takeaways from this first experiment are the following: i) The modification of a large number of jobs is faster than its deployment: most of the time, it is better to modify the pipeline than to stop and restart it. ii) The time it takes to deploy, modify, delete jobs is linear in the number of jobs, showing that, at least up to a significant number of jobs, the system does not exhibit any scalability issue.

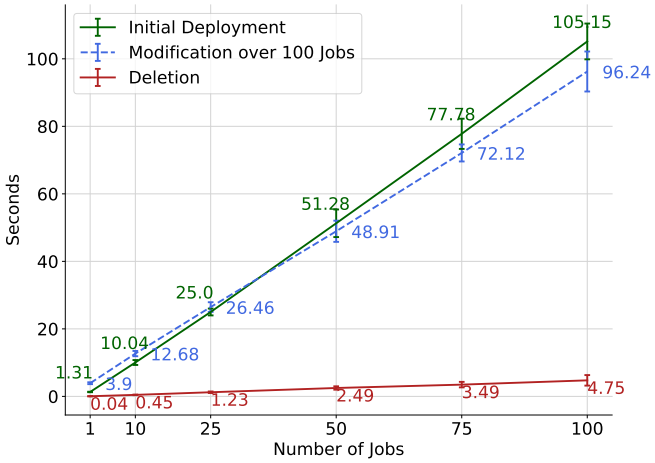

Fig. 6. Multiple deployment measurements

SpecK depends on a restful API. One common issue with restful services is that a large number of client requests may slow down (or even lead to the failure of) the underpinning server; yet SpecK does not create requests unless the user submits a new pipeline. In other words, SpecK does not generate any traffic unless the user emits requests, and the traffic generated is no more no less the traffic needed to deploy jobs. In other words, SpecK does not bring any measurable overhead in terms of traffic. The only measurable overhead is the disk space needed by SpecK to store the state of the deployed pipeline: As mentioned in Section 3.3, a persistent key-value store stores the state of the currently deployed jobs. The store is written on disk. Throughout all experiments the space used on disk remains very low. As an example, we mesured that a 1000-job pipelines only requires 754KB.

4.2 Edge-to-Cloud deployment

In order to validate the usage of the spec API in more realistic settings, we developed a Fog-targeted road-traffic monitoring application, consisting of 4 jobs, some of the jobs being duplicated over geographically distributed sites, making a total of 17 single-operator Flink Jobs. The dataset used included real-world data from 245,369 connected vehicles moving across Italy, and artificially expanded so as to increase the scale of the deployment and the velocity of the streams. This expansion does not means that artificial data were added, but that data were played at a higher rate than in the original dataset. The data are basically a list of cars passing by at specific points at specific times. Each time a car is detected at a sensor point, it is added in the stream.

The deployed pipeline is illustrated in Figure 7. It is composed of four jobs, which aims at producing both timely statistics about the local road traffic and long term statistics at the global scale. The first type of statistics is supposed to

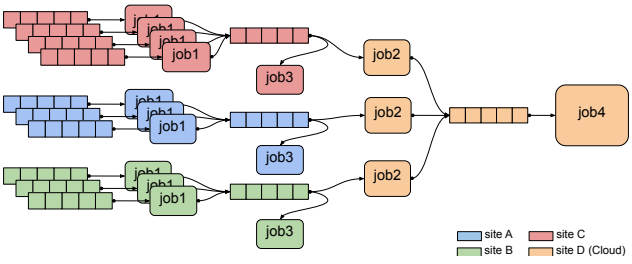

Fig. 7. Deployment of the application over multiple sites.

be generated locally, on each edge where data is ingested. The second type is for storage and later reuse and is thus typically computed in the Cloud. Each site includes its own Flink SPE and Mosquitto MQTT broker. Figure 7 illustrates its deployment over 3 edge sites and 1 cloud site. Let us review the four jobs: `Job 1` preprocesses the data received locally by filtering and cleaning them before they are injected into the rest of the pipeline. It removes erroneous data items or badly formatted ones. This cleaning is a stateless operation, not very time-consuming or compute-intensive and can typically be performed locally, close to where the data are *sensed*. As filtering can be done in parallel on each data source, we assume that one instance of Job 1 is deployed for each stream of data. `Job 2` is a forwarder: it collects data produced by Job 1 instances and sends them to the entry topic of Job 4 which will merge all data coming from the different sites. `Job 3` performs windowed statistics of data received locally on one site. It produces timely statistics about the recent - near real-time - local traffic. These statistics generation cannot be parallelized due to their stateful nature. In other words, there is a single instance of this job per site. `Job 4` is a merging operator which establishes global statistics over the data sent by the different sites, so later global post-processing can be conducted.

The benefits of such a deployment allowed by SpecK are twofold. By placing Job 1 at the edge, fewer data are sent across the network to Job 4, and faster data processing and monitoring rate is obtained when Job 3 runs closer to the source. To prove this, we focused on the performance of Job 3. We deployed two scenarios, both deployed using SpecK. In the *remote scenario*, Job 3 does not run on the same site where the data are generated. In the *local scenario*, Job 3 is placed on the same site where the data are generated. Then we gradually increased the data generation rate of the data we gathered from the real-world and compared the outcomes. Figure 8 shows the results. Having an ingestion rate of one message produced every 2ms already allows this benefit: as shown by Figure 8 (b), the local count of the car increases slower when done in the Cloud. The difference increases when the ingestion rate is 1 msg/ms (see Figure 8 (c)).

We then experimented with two global deployments, referred to as the *Fog* and the *Cloud* scenarios, respectively. In the **Fog scenario**, all 4 jobs were placed across A, B, C, D sites as shown in Figure 7. Colors on the figure denotes

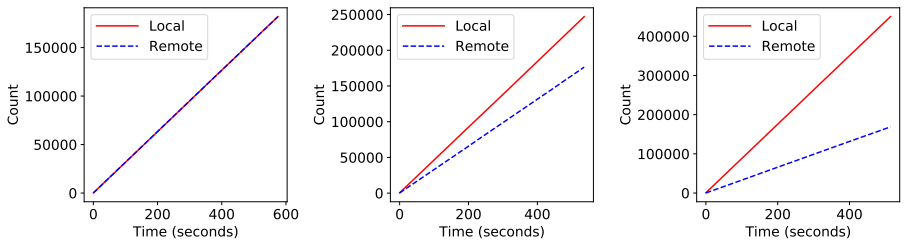

Fig. 8. Data processing rate sample.

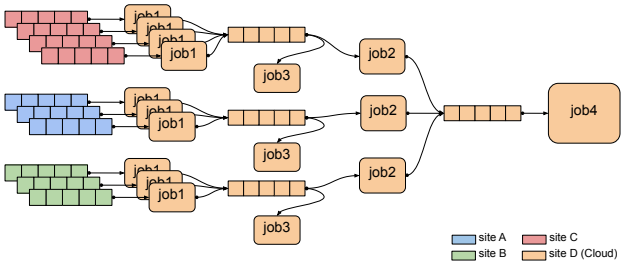

Fig. 9. Deployment of the application on a single site.

the site where each job and message queues are located. In the **Cloud scenario** (Figure 9), all 4 jobs are placed on site D, meaning the regionally generated data transit through the network to be directly processed on site D.

Figure 10 illustrates the output rate of each Job of the pipeline obtained in both deployment scenarios shown in Figure 7 and Figure 9. The X-axis gives the data input rate at each site. The Y-axis gives the benefit ratio of using the Fog scenario compared to the Cloud in terms of processing speed for Job 3 and 4, which provides local and global statistics, respectively. Here, processing speed is to be understood as the time taken to process a fixed amount of messages. Each deployment scenario was conducted for 15 minutes to measure the difference between the data processing rate of Fog and Cloud. The performance was averaged over 5 runs. The differential output rate can be expressed as $D = (100/F) * (F - C)$ where F is the amount of data processed in Fog scenario and C is the amount of data processed in Cloud scenario over these 15 minutes.

When a curve is above 0, data processing in Fog is quicker for this job. When a curve is below 0, it means the Cloud is quicker. Initially, when the data ingestion rate is low, the Fog deployment does not provide faster data processing rate compared to the Cloud for the final job (Job 4). This can be explained as in the Fog scenario, data need to traverse multiple MQTT brokers,

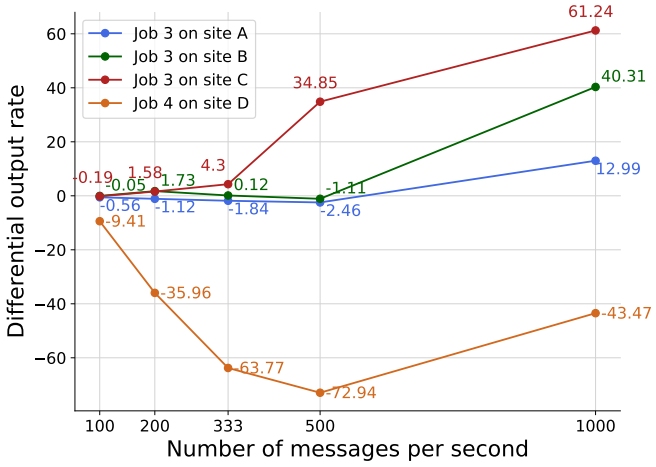

Fig. 10. Differential output rate.

each adding latency to the global data transmission from Edge to Cloud. Yet on the other hand, when we increased the ingestion rate up to 1 message sent per millisecond, data processing becomes closer on both scenarios due to the MQTT broker message buffering. Also, we observed that the fog scenario compared to cloud gave better results for Job 3 which provides the local statistics, showing that multiple-site Job deployments benefit from SpecK.

5 Conclusion

While the Fog is increasingly cited as a key enabler for a new generation of latency-sensitive applications, reference architectures and concrete tools to fully benefit from it are still missing. In this work, we tackle the problem of how to effectively manage stream processing over a geographically distributed compute infrastructure. Our approach, SpecK is based on the idea of *composing* SPE instances, each one running their SPE of choice and managing local computing resources. By adding this further layer of coordination, we are able to effectively leverage locally-available resources, while providing application developers with an easy set of primitives to deal with, so as to facilitate the cumbersome process of deploying an SP pipeline over a large scale platform. Experiments with the SpecK prototype showed that it is able to effectively handle complex stream computations, coordinating the usage of locally-available resources over a geographically-distributed infrastructure with a limited overhead.

Future work will include enhancements in the programming language used to define computations and adding support for stateful operators, requiring a mechanism for managing state migrations. Also, we are moving to the performance and reliability aspects of SpecK. In particular, we are developing monitoring tools to automate migrations of jobs in function of the platform's conditions and users' performance requirements. Finally, we will extend the current SpecK implementation so it can support a larger set of SPE and message brokers.

Acknowledgements

This work is part of the FogGuru project which has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

References

1. Apache edgent. https://edgent.incubator.apache.org/
2. Grid'5000. https://www.grid5000.fr
3. Antonini, M., Vecchio, M., Antonelli, F.: Fog computing architectures: a reference for practitioners. CoRR abs/1909.01020 (2019), http://arxiv.org/abs/1909.01020
4. de Assunção, M.D., Veith, A.D.S., Buyya, R.: Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. J. Network and Computer Applications 103 (2018)
5. Balouek-Thomert, D., Renart, E.G., Zamani, A.R., Simonet, A., Parashar, M.: Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. Int. J. High Perform. Comput. Appl. 33(6) (2019)
6. Beckman, P., Dongarra, J., Ferrier, N., Fox, G., Moore, T., Reed, D., Beck, M.: Harnessing the Computing Continuum for Programming Our World, chap. 7, pp. 215–230. John Wiley Sons, Ltd (2020)
7. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36(4) (2015)
8. Cardellini, V., Grassi, V., Lo Presti, F., Nardelli, M.: Distributed qos-aware scheduling in storm. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. pp. 344–347 (2015)
9. Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. In: ACM SIGMOD'13. pp. 725–736 (2013)
10. Cheng, B., Papageorgiou, A., Bauer, M.: Geelytics: Enabling on-demand edge analytics over scoped data sources. In: 2016 IEEE International Congress on Big Data (BigData Congress). pp. 101–108 (2016)
11. Claudel, B., Huard, G., Richard, O.: Taktuk, adaptive deployment of remote executions. In: Kranzlmüller, D., Bode, A., Hegering, H., Casanova, H., Gerndt, M. (eds.) Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC 2009, Garching, Germany, June 11-13, 2009. pp. 91–100. ACM (2009), https://doi.org/10.1145/1551609.1551629
12. Fu, X., Ghaffar, T., Davis, J.C., Lee, D.: Edgewise: A better stream processing engine for the edge. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 929–946. USENIX Association, Renton, WA (Jul 2019)
13. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IEEE Transactions on Parallel and Distributed Systems 25(6), 1447–1463 (2013)

14. Gibert Renart, E., Da Silva Veith, A., Balouek-Thomert, D., De Assunção, M.D., Lefèvre, L., Parashar, M.: Distributed operator placement for iot data analytics across edge and cloud resources. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 459–468 (2019)
15. Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., Soriente, C., Valduriez, P.: Streamcloud: An elastic and scalable data streaming system. IEEE Transactions on Parallel and Distributed Systems 23(12), 2351–2365 (Dec 2012)
16. Lakshmanan, G.T., Li, Y., Strom, R.: Placement strategies for internet-scale data stream systems. IEEE Internet Computing 12(6), 50–60 (2008)
17. Light, R.A.: Mosquitto: server and client implementation of the mqtt protocol. Journal of Open Source Software 2(13), 265 (2017)
18. Liu, F., Tang, G., Li, Y., Cai, Z., Zhang, X., Zhou, T.: A survey on edge computing systems and tools. Proceedings of the IEEE 107(8), 1537–1562 (2019)
19. Mahmud, R., Kotagiri, R., Buyya, R.: Fog Computing: A Taxonomy, Survey and Future Directions, pp. 103–130. Springer Singapore (2018)
20. Milojicic, D.: The edge-to-cloud continuum. Computer 53(11), 16–25 (2020)
21. Nicolae, B., Bresnahan, J., Keahey, K., Antoniu, G.: Going back and forth: Efficient multideployment and multisnapshotting on clouds. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC). pp. 147–158 (2011)
22. O'Keeffe, D., Salonidis, T., Pietzuch, P.: Frontier: Resilient edge processing for the internet of things. Proceedings of the VLDB Endowment 11(10), 1178–1191 (2018)
23. Peng, B., Hosseini, M., Hong, Z., Farivar, R., Campbell, R.: R-storm: Resource-aware scheduling in storm. In: Proceedings of the 16th Annual Middleware Conference. pp. 149–161. Middleware '15 (2015)
24. Pisani, F., Brunetta, J.R., Martins Do Rosario, V., Borin, E.: Beyond the fog: Bringing cross-platform code execution to constrained iot devices. In: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 17–24 (2017)
25. Prosperi, L., Costan, A., Silva, P., Antoniu, G.: Planner: Cost-efficient execution plans placement for uniform stream analytics on edge and cloud. In: 2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS). pp. 42–51 (2018)
26. Rosendo, D., Silva, P., Simonin, M., Costan, A., Antoniu, G.: E2Clab: Exploring the Computing Continuum through Repeatable, Replicable and Reproducible Edge-to-Cloud Experiments. In: Cluster 2020 - IEEE International Conference on Cluster Computing. pp. 1–11. Kobe, Japan (Sep 2020)
27. Silva, P., Costan, A., Antoniu, G.: Investigating edge vs. cloud computing trade-offs for stream processing. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 469–474 (2019)
28. Silva, P., Costan, A., Antoniu, G.: Towards a methodology for benchmarking edge processing frameworks. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 904–907 (2019)
29. da Silva Veith, A., de Assunção, M.D., Lefèvre, L.: Latency-aware placement of data stream analytics on edge computing. In: Pahl, C., Vukovic, M., Yin, J., Yu, Q. (eds.) Service-Oriented Computing. pp. 215–229. Springer International Publishing, Cham (2018)
30. Snyder, B., Bosanac, D., Davies, R.: Introduction to apache activemq. Active MQ in action pp. 6–16 (2017)

31. Thein, K.M.M.: Apache kafka: Next generation distributed messaging system. International Journal of Scientific Engineering and Technology Research 3(47), 9478–9483 (2014)
32. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.: Storm@twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. p. 147–156. SIGMOD '14, Association for Computing Machinery, New York, NY, USA (2014), https://doi.org/10.1145/2588555.2595641
33. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: fault-tolerant streaming computation at scale. In: 24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'13). pp. 423–438. Farmington, USA (Nov 2013)