Elastic Pulsar Functions for Distributed Stream Processing

Gabriele Russo Russo
russo.russo@ing.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

Antonio Schiazza
antonio.schiazza@gmail.com
University of Rome Tor Vergata
Rome, Italy

Valeria Cardellini
cardellini@ing.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

ABSTRACT

An increasing number of data-driven applications rely on the ability of processing data flows in a timely manner, exploiting for this purpose Data Stream Processing (DSP) systems. Elasticity is an essential feature for DSP systems, as workload variability calls for automatic scaling of the application processing capacity, to avoid both overload and resource wastage. In this work, we implement auto-scaling in Pulsar Functions, a function-based streaming framework built on top of Apache Pulsar. The latter is is a distributed publish-subscribe messaging platform that natively supports serverless functions. Considering various state-of-the-art policies, we show that the proposed solution is able to scale application parallelism with minimal overhead.

CCS CONCEPTS

• **Information systems** → **Stream management**; • **Computer systems organization** → Distributed architectures.

KEYWORDS

Data Stream Processing, Auto-Scaling, Self-Adaptation

ACM Reference Format:
Gabriele Russo Russo, Antonio Schiazza, and Valeria Cardellini. 2021. Elastic Pulsar Functions for Distributed Stream Processing. In *Companion of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21 Companion), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3447545.3451901

1 INTRODUCTION

The increasing, ubiquitous presence of sensors in our houses, offices and cities has contributed for years to a unending growth in the volume of daily produced data. In turn, such data availability has fostered the development of new data-driven applications and services, which pervade and assist our everyday life, by extracting valuable insight from raw data sets. In particular, many of these applications deal with continuous data flows, also called *streams*, which must be analyzed in a timely manner.

In this context, Data Stream Processing (DSP) systems have emerged as a *de facto* standard for near real-time processing of high-volume data streams [5]. DSP applications are usually defined as directed acyclic graphs (DAG) whose vertices are named *operators* and edges are streams, that is unbounded sequences of data units, which connect operators. Operators are processing elements that take one or more streams as input, apply transformations or functions to the data, and possibly output a new data stream. Letting data flow through multiple operators, complex processing functions can be realized. Among the set of vertices in the application DAG, we distinguish special vertices: sources and consumers. Vertices with no input edges represent data *sources*, which generate the input streams. Vertices with no outgoing edges represent *consumers*, such as dashboards or files, which receive the application results.

To keep up with fast and high-volume input streams, DSP applications often exploit distributed computing infrastructures. By doing so, operators can be executed concurrently as independent threads or processes on multiple computing nodes. Furthermore, multiple parallel replicas of each operator can be launched to increase the overall processing capacity. On the one hand, such parallelization allows the input stream to be distributed among the parallel instances, reducing the load imposed to each one and, thus, improving the resulting performance. On the other hand, since DSP workloads are highly variable over time, dynamically adjusting operator parallelism at run-time is necessary to avoid both overload and resource wastage. As such, *elasticity* (or, operator *auto-scaling*) has been identified as an essential feature for modern DSP systems and a lot of effort has been spent by researchers on this topic [6, 25].

Besides several open-source frameworks for the development and execution of distributed DSP applications (e.g., Flink, Heron, Storm), recently, systems originally designed for data ingestion and distributed messaging, such as Kafka and Pulsar, have been extended to enable computation on the incoming data streams. By doing so, it becomes easier to deploy DSP applications, removing the need of configuring and managing different systems for data ingestion and processing. In particular, in this work we focus our attention on Pulsar, which is a framework for distributed and scalable messaging. Pulsar enables the development of DSP applications through a function-based computing framework, named *Pulsar Functions*. This project allows developers to implement complex processing pipelines by combining simple functions, possibly written using different languages.

Pulsar Functions can be executed over distributed nodes and support parallelization for increased throughput. Unfortunately, Pulsar lacks built-in support for seamless function auto-scaling. In this paper, we integrate elasticity mechanisms and policies in Pulsar Functions, so that the parallelism level of each function is be effectively and efficiently self-adjusted at run-time in response to workload variations. The key contributions of our work are as follows:

ICPE '21 Companion, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8331-8/21/04…$15.00
https://doi.org/10.1145/3447545.3451901

- We design and implement an auto-scaling framework for Pulsar Functions, based on the *Monitor, Analyze, Plan and Execute* (MAPE) pattern for self-adaptive systems;
- We extend Pulsar Functions to allow function parallelism to be modified at run-time without interrupting the operation of the existing instances, hence enabling seamless adaptation;
- We integrate and evaluate different auto-scaling policies in our solution, including a widely used threshold-based approach, a policy based on queueing theory, and a policy derived from the literature [13].

The remainder of this paper is organized as follows. We provide background information about Pulsar in Sec. 2. We describe our auto-scaling solution and its integration in Pulsar in Sec. 3. We present the policies integrated in our solution in Sec. 4. An experimental evaluation of our work is described in Sec. 5. We review related work in Sec. 6 and conclude in Sec. 7.

2 OVERVIEW OF APACHE PULSAR

Apache Pulsar [1] is a cloud-native, distributed messaging and streaming platform, originally developed at Yahoo and now within the Apache Software Foundation. Pulsar aims at enabling scalable, low-latency and durable messaging based on the publish-subscribe paradigm, with support for multi-tenancy and geographical replication. Being able to ingest millions of messages with very low latency, Pulsar well suits use cases involving streaming data. Furthermore, *Pulsar Functions*, a lightweight computing framework, adds support for stream-native data processing on top of Pulsar. As such, Pulsar provides a comprehensive platform to build streaming applications, able to both ingest and process high-volume and fast data flows in a distributed environment.

Messaging in Pulsar is built on top of the publish-subscribe pattern, where *producers* publish messages to *topics*, and *consumers* subscribe to these topics to read, process and acknowledge the messages. Topics are named channels that allow exchange of messages between producers and consumers. Pulsar supports both persistent and non-persistent topics: messages sent to persistent topics are durably persisted on disks, whereas data sent to non-persistent topics are not durably stored on disk.

Consumers can choose from various subscription modes when subscribing to topics, each supporting a class of communication use cases. For instance, in the *failover* subscription mode all the messages are delivered to a single consumer, denoted as *master* consumer. If the master consumer disconnects, it is replaced by another consumer. Conversely, the behavior of a message queue is obtained with the *shared* subscription mode, where multiple consumers can attach to the same subscription, and messages are delivered in a round robin distribution across consumers. The shared subscription provides at-least-once semantics: any given message is initially delivered to a single consumer and, if it is not acknowledged, it will be re-transmitted to a different consumer. Furthermore, the *key-shared* mode extends the shared subscription mode causing messages with the same *key* to be sent to the same consumer.

[1] https://pulsar.apache.org/

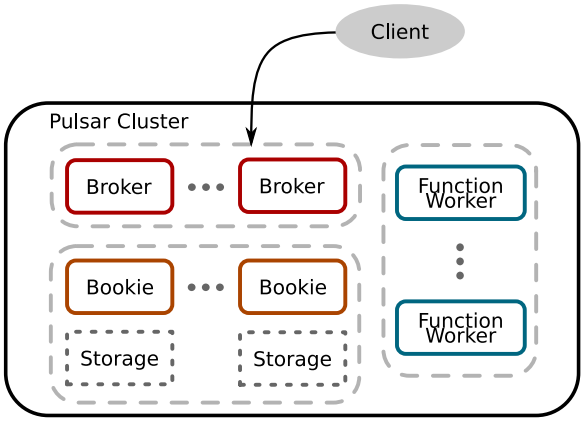

Figure 1: High-level Pulsar architecture.

2.1 Architecture

Pulsar has an elegant architecture, which was designed with the goals to provide scalability and flexibility so to support applications needing any combination of queuing, messaging, streaming and lightweight compute for events [23]. The high-level architecture of Pulsar is shown in Fig. 1. An instance of Pulsar comprises one or more *clusters*. Clusters may be geographically distributed and data can be replicated among different clusters. Each cluster is populated by three groups of components: brokers, bookies, and ZooKeeper instances. *Brokers* are the fundamental components for messaging and build the serving layer of Pulsar. Brokers handle the incoming messages, dispatch them to consumers and store them in the so-called bookies. Pulsar message brokers are stateless components that primarily run two other components: an HTTP server and a dispatcher. The HTTP server exposes a REST API for both administrative tasks and topic lookup for producers and consumers; the dispatcher is a TCP server that handles all the data transfers over a custom binary protocol. For the sake of performance, messages are dispatched out of a managed ledger cache, which avoids unnecessary reads from the storage layer. When the backlog exceeds the cache size, the broker starts reading entries from the bookies.

Bookies represent the storage layer of Pulsar, which must guarantee message durability. Bookies are instances of Apache BookKeeper[2], a distributed write-ahead log that well suits the needs of Pulsar. Indeed, BookKeeper provides efficient storage for sequential data over distributed nodes. BookKeeper automatically handles data replication among its instances and is horizontally scalable, as new bookies can be seamlessly added to a cluster. It is worth noting that, by decoupling the serving and storage layers, Pulsar allows them to be scaled independently as needed. Besides message data, bookies also persist *cursors*, i.e., the subscription positions for consumers.

To coordinate the various components and store cluster-level configuration, each Pulsar cluster is also equipped with an Apache ZooKeeper[3] ensemble, an open-source service for distributed coordination.

[2] https://bookkeeper.apache.org/
[3] https://zookeeper.apach.org/

2.2 Pulsar Functions

On top of the core messaging architecture of Pulsar, a data processing layer has been introduced by means of the Pulsar Functions[4]. Pulsar Functions are processing entities that (i) consume messages from one or more topics, (ii) apply a user-defined processing logic to each message, and (iii) publish the results of the computation to another topic. As such, a Pulsar Function can be regarded as an *operator*, in the terminology of DSP systems. Moreover, by defining multiple functions, each consuming messages produced by another function, it is possible to develop complex stream processing applications, without the need for a separate data processing system (e.g., Apache Flink). This can significantly simplify the operations related to deploying and managing DSP applications, while enjoying the features of a highly scalable ingestion platform.

Inspired by the increasingly popular Function-as-a-Service (FaaS) paradigm, Pulsar Functions are Lambda-style functions specifically designed to use Pulsar as a message bus and serves as a means to support analytics on real-time data streams in a serverless fashion. The programmer needs only implement the processing logic to be applied to each incoming message, possibly producing new messages to be published to other topics (including a special "log" topic for logging purposes). Pulsar currently allows developers to write the function logic in Java, Python or Go. For all these languages, Pulsar provides specific SDK libraries for function development; for Java and Python, developers are also allowed to use language-native interfaces (e.g., the `java.util.function.Function` interface in Java), with no specific dependency on Pulsar. Pulsar Functions enable the definition of both stateless and stateful operators. For state management, Pulsar Functions rely on the *table service* provided by BookKeeper, which is a key-value data store. In order to use the table service, functions must be defined using the Pulsar SDK.

Internally, a Pulsar Function comprises three core components: consumer, executor, and producer. The *consumer* is responsible for the subscriptions to the input topics and, thus, for message retrieval. The *executor* is in charge of applying the user-defined logic to each message, passing its output to the producer. The *producer* publishes the processing results to the output topics.

Each Pulsar Function can be associated with one or more parallel instances at run-time. These function instances are executed within *Function Workers*, additional components that enrich the core Pulsar architecture described above. Function Workers can be either activated alongside brokers or as standalone nodes in the cluster. Within Workers, each function instance can be executed either as a thread or process, depending on the selected configuration. Alternatively, if a Kubernetes cluster is available, functions can be spawned as *StatefulSets* within Kubernetes. In this work, we focus on the case of functions running as threads, and Function Workers deployed in standalone nodes.

3 AUTO-SCALING OF PULSAR FUNCTIONS

Pulsar Function introduce a stream processing layer on top of Pulsar messaging system. A fundamental requirement of modern stream processing systems is elasticity, that is the ability of adjusting the allocated processing capacity to respond to workload variations [6, 25]. While changing the parallelism level of Pulsar Functions at run-time is possible, reconfiguration decisions are under the responsibility of the system administrator and must be manually triggered. Therefore, in this work we aim at integrating mechanisms and policies for *automatic* scaling of the Pulsar Functions.

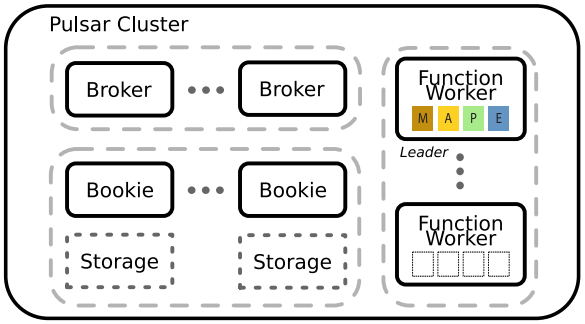

Figure 2: Extended Pulsar architecture.

The solution we present is organized according to the well-known *Monitor, Analyze, Plan and Execute* (MAPE) pattern for self-adaptive systems [14]. Application adaptation (i.e., function parallelism adaptation) consequently relies on four key adaptation components: Monitor, Analyzer, Planner and Executor. The *Monitor* component is in charge of monitoring data arrivals and function executions. Based on this information, the *Analyzer* component is able to characterize the current system state at any time. The output of the system analysis phase is used by the *Planner* component to make adaptation decisions and, thus, plan possible parallelism reconfigurations. Finally, planned reconfigurations are enacted by the *Executor* component.

Our auto-scaling components are integrated in the existing Pulsar architecture and, specifically, they extend the control functionality of the Function Workers, as depicted in Fig. 2. In particular, auto-scaling control is delegated to the *leader* Function Worker, which is identified by means of a leader election among all the Workers in the cluster.

The Monitor component is built on top of the metrics sub-system of Pulsar, which provides information about existing topics and functions. To support auto-scaling, we introduce further metrics and collect them for analysis. Moreover, the Monitor also constructs a graph representation of the DSP application based on the gathered information, as Pulsar does not provide any higher-level abstraction on top of topics and functions. Specifically, we model the application as a DAG $G = (V, E)$, where V is the set of vertices (i.e., functions), and E the set of edges (i.e., streams flowing between functions through a topic). The Analyzer and Planner components use this information to make scaling decisions, which depend on the specific auto-scaling policy in use (details about the policies will be given in the next section).

The Executor component is responsible for changing function parallelism based on the planned scaling operations. For this purpose, we rely on the *update* mechanism provided by Pulsar Functions, which allows users to modify various parameters related to function execution. Unfortunately, we observed that every parallelism update (and, in general, every configuration update) forces

[4] https://pulsar.apache.org/docs/en/functions-overview/

the termination of all the current function instances and the creation of new instances. This represents a major limitation, as every parallelism reconfiguration interrupts application processing. Although such reconfiguration overhead exists in most the DSP frameworks (see, e.g., [3, 4]), given the lightweight nature of Pulsar Functions, we aim to obtain seamless elasticity. Therefore, we extend the update functionality and avoid the replacement of all the function instances when only the parallelism is affected. By doing so, scaling actions do not cause noticeable interruptions in the normal data processing. Our extended version of Pulsar is publicly available.[5]

4 AUTO-SCALING POLICIES

To control function auto-scaling, we consider different policies. First, we consider a simple-yet-popular threshold-based scaling policy, which triggers reconfigurations based on the monitored queue length. As this policy is unaware of the perceived application performance (e.g., processing latency), we also present a scaling policy based on queueing theory, which estimates function response time to make decisions. For comparison, we also consider the auto-scaling policy for DSP operators presented by Kalavri et al. [13]. In the next section, we introduce each policy.

4.1 Threshold-based policy

A widely adopted approach for the definition of auto-scaling policies consists in triggering reconfigurations whenever the observed value of a key metric (e.g., resource utilization, queue length) is beyond (or, below) a pre-defined threshold. The main advantage of such threshold-based heuristic policies is their implementation simplicity, as they do not require any system modeling effort or significant computation.

In this work we consider the topic backlog size Q_f, that is the amount of data ready to be processed by each function f, as the reference metric for scaling. Values for Q_f range in the interval $[0, \infty)$, under the modeling assumption of an infinite amount of memory available for the topic. We define two threshold values T_{in} and T_{out}, which partition the value space of the metric into three intervals, as follows:

- $[0, T_{in}]$: low-load interval
- $(T_{in}, T_{out}]$: in-range interval
- (T_{out}, ∞): overload interval

At run-time, based on the measured value for Q_f, we identify the system state among the possible load conditions defined by the intervals above. The function parallelism is updated according to the system state, as follows:

- low-load $\longrightarrow$ scale-in (terminate 1 instance)
- in-range $\longrightarrow$ do nothing
- overload $\longrightarrow$ scale-out (add 1 instance)

As exceptions to these rules, no scale-in is clearly allowed when a single instance is active, in order to keep at least one running replica; no scale-out is allowed when the maximum number of instances has been reached.

[5] https://bitbucket.org/aschiazza/pulsar-fork-master/src/aschi-fork-2.4.1/

On the one hand, this approach is very easy to implement, as mentioned above. On the other hand, its adaptation behavior directly depends on the choice of the threshold values to use. Identifying the best thresholds to be used to satisfy users' performance requirements is not always obvious, and the thresholds must often be tuned by means of a trial-and-error approach.

4.2 Queueing theory-based auto-scaling

We also consider model-based auto-scaling policies, based on queueing theory. The core idea is exploiting queueing networks to model DSP applications and estimate performance at run-time (e.g., processing latency or throughput). The availability of these performance estimates allows us to identify the minimum amount of allocated resources (i.e., function parallelism) able to satisfy users' requirements. In particular, in this work we consider performance requirements expressed in terms of maximum processing latency along each path in the application DAG, which usually corresponds to a single query computed by the application.

To apply queueing theory to the considered problem, we first need to model DSP applications running on top of Pulsar as queueing networks. In particular, we model applications as *open* queueing networks, where external arrivals model data arrivals to the application input topic, and each station represents a function (i.e., an operator). As multiple instances can be associated with each function, all fetching data from the same topic, we model each function as a GI/G/k queueing system. We are clearly interested in estimating the response time of the different stations and, thus, the total response time along end-to-end paths.

Unfortunately, the GI/G/k model is a general model for which no exact analytical results are available. Instead, we need to resort to approximations and, specifically, to the Allen-Cunneen approximate formula [2]. The Allen-Cunnen formula provides an approximation of the waiting time T_q of GI/G/k queues:

$$T_q \backsimeq \frac{c_a^2 + c_s^2}{2k} \frac{P_k}{\mu\,(1-\rho)} \tag{1}$$

where c_a and c_s are coefficients of variation of, respectively, arrival times and service times; k is the parallelism level of the function and, hence, the number of servers; μ is the average service rate of the function; ρ is the average utilization of the function; and, P_k is the probability for a data unit to wait in the queue because all the function instances are busy. The latter term can be in turn approximated as follows:

$$P_k \backsimeq \begin{cases} \dfrac{\rho^k + \rho}{2} & \rho \geq 0.7 \\ \rho^{\sqrt{k+1}} & \rho < 0.7 \end{cases} \tag{2}$$

By means of the equations above, we directly obtain an estimate for the response time T of each function:

$$T = \frac{1}{\mu} + T_q \tag{3}$$

Moreover, we can evaluate the total latency along every path π in the application DAG as the sum of the response times T^i of each function i appearing on the path:

$$T^\pi = \sum_{i \in \pi} T^i \tag{4}$$

In our solution, all the parameters involved in the equations above are estimated online exploiting information collected in the monitoring phase. To this end, we introduced new metrics in Pulsar (e.g., the function processing time, the inter-completion time). The idea behind our auto-scaling policy is searching the minimum number of servers k that is necessary to keep the response time below a maximum value R_{max} specified by users. In the following, we will consider cases where the maximum response time is specified either at level of single functions or whole application paths.

4.2.1 Function response time requirements. If a maximum response time requirement is expressed for each function, scaling decisions can be made independently for each of them by the auto-scaling controller. In particular, applying (3) the policy identifies the minimum parallelism level which prevents performance violations. It proceeds in a greedy fashion: the policy first evaluates the response time for the minimum parallelism configuration and, then, increases the parallelism level until the estimated performance is within R_{max} (or the maximum allowed parallelism is reached).

This approach is quite simple to analyze and implement. However, setting a proper performance constraint on each function may not be straightforward, as users' requirements are usually specified at level of whole processing pipelines.

4.2.2 Application response time requirements. If the maximum response time requirements are specified at level of whole paths in the application DAG (hence, queries), we need to use (4) to evaluate the total average response time, given a parallelism configuration. The planning algorithm we use again relies on a heuristic approach. The algorithm, when executed, classifies paths in the application graph into different groups, based on the estimated response time along them. Paths along which the response time will exceed R_{max} are considered for scale-out. In this case, the function with largest estimated response time in the path is selected for the scale-out. Paths along which the response time is between R_{max} and a fraction $\alpha \in (0, 1)$ of it are not considered for scaling. Paths with response time smaller than αR_{max} are considered for scaling in. In particular, the function with lowest estimated response time among those running more than a single instance (if any) is selected for the scale-in.

4.3 DS2 Approach

We also consider and implement the auto-scaling policy named DS2 that is presented in [13] and has been successfully integrated in different DSP frameworks, including Heron and Flink. DS2 aims at overcoming the limitations of simple threshold-based scaling policies, as well as other methodologies based on coarse-grained monitoring information. Conversely, DS2 relies on accurate characterizations of the operator processing rates to ensure adequate resource provisioning to sustain the incoming data flows.

In particular, DS2 tries to measure the *true* processing rate of each operator, observing that actual operator throughput is influenced by external factors, such as the maximum output rate of upstream operators. To do so, DS2 relies on fixed-length observation windows W. Within each window, DS2 measures the *useful time* of each operator W_u, that is the amount of time the operator spends deserializing input data, processing them, and serializing output data.

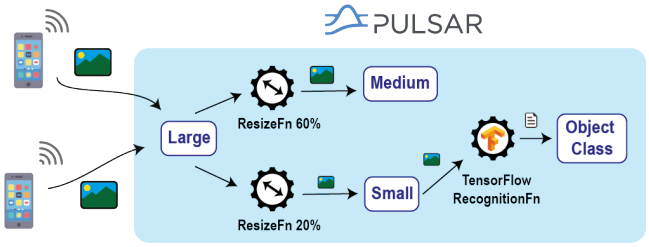

Figure 3: Reference application, which processes a stream of images.

Based on this and other measurements, DS2 computes the optimal parallelism level for the i-th operator π_i as the ratio of the aggregated true output rate of its upstream operators (assuming that they keep up with their inputs) to the average true processing rate per instance of operator i. Overall, DS2 requires a single visit of the application DAG to compute the parallelism of every operator.

5 EVALUATION

In this section, we present the experiments we performed to evaluate the auto-scaling mechanisms we implemented and the considered policies.

5.1 Experimental Setup

We deploy Pulsar on top of a Kubernetes cluster running in the Cloud, relying on the *Elastic Kubernetes Service* provided by AWS. The cluster we use is made of 4 computing nodes, corresponding to `t3.xlarge` EC2 instances. One node hosts the services associated to ZooKeeper, BookKeeper and the brokers, along with Grafana and Prometheus, which are used for monitoring. The remaining three nodes host the Function Workers.

For the experiments we consider a reference application that processes streams of images. Input images are published to a Pulsar topic. The application, whose topology is depicted in Fig. 3, creates two resized copies of each image, by means of two functions. While one copy is stored to disk, the other one is the input for an object recognition function, which relies on a pre-trained TensorFlow model.

The inter-arrival times for the input data units are generated randomly according to Exponential distributions. In each experiment, we consider a sequence of 10-minute intervals in which the average arrival rate is kept constant. The arrival rate is changed across different intervals, to evaluate how the system self-adapts to the workload variations – as illustrated in Fig. 4.

We use the following parameters for the auto-scaling policies. The invocation period for the analysis and planning phases of the MAPE loop, as well as the observation window W of the DS2 approach, is set to 30 seconds. For the threshold-based policy, we set the queue length threshold values as T_{in}=50 tuples and T_{out} = 150 tuples. For the queueing-based policy, we require the response time along both the paths in the application DAG to be kept within 1s. The maximum parallelism level for each function is set to 15.

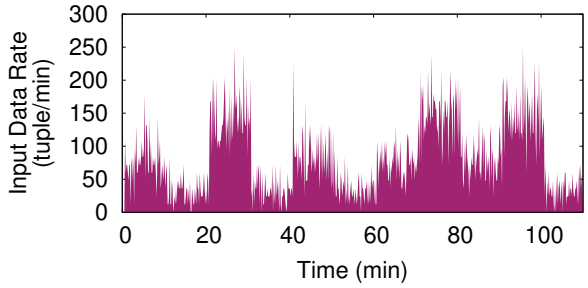

Figure 4: Workload used in the experiments.

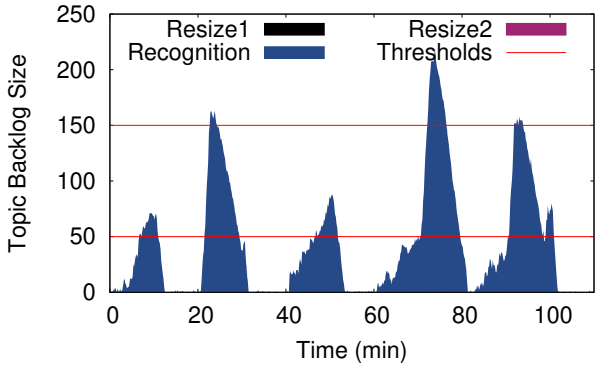

Figure 5: Input queue length for the different functions when using the threshold-based policy.

5.2 Results

We first consider the threshold-based auto-scaling policy. This policy keeps the amount of queued data for each function within the given thresholds values. Figure 5 shows the backlog size of the input topic of each function in the application throughout the experiment (the only function showing a significant amount of queued data is the object recognition one though). We can see that every time the queue length grows beyond the upper limit value, the system reacts and brings it back under control. In Fig. 6, we report the measured application response time and the total number of active function instances. As regards the response time, we only show the results related to the application path involving the object recognition function, as the computational load along the other path was much lower.

The total number of active function instances varies between 3 (one per operator) and 7 in this experiment, with an average of 3.5 running instances. The threshold-based policy manages to keep the backlog size within the specified thresholds. However, as most rule-based scaling policies, it does not provide guarantees on the perceived response time, which is 23s on average, with a peak value of 74s. Clearly, the used thresholds can be tuned to obtain different scaling behaviors, if the observed response time is not acceptable for the users. Nonetheless, such manual tuning process is far from ideal for systems hosting multiple applications, each likely subject to different performance requirements.

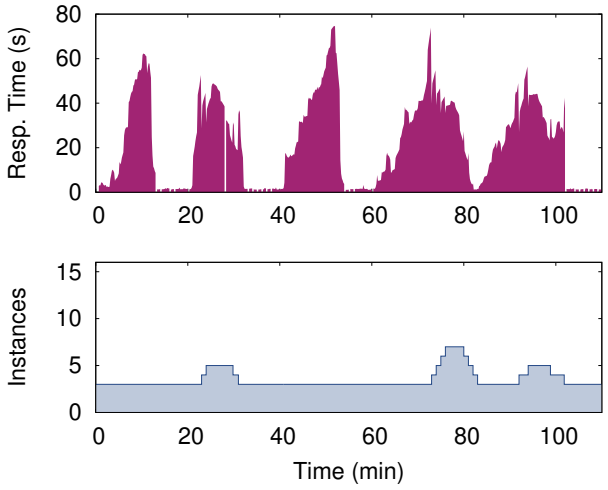

Figure 6: Results with the threshold-based policy.

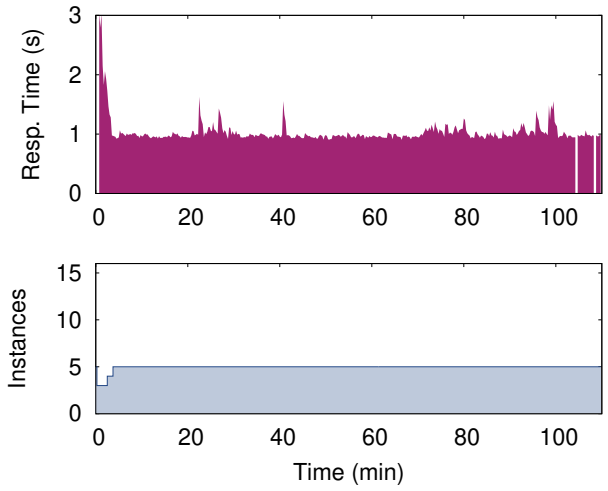

Figure 7: Results with the queueing-based policy.

Differently from the threshold-based approach, the queueing-based policy takes users' requirements as input and, specifically, the maximum desired response time. We verify what happens using a strict average response time requirement and, specifically, 1 second. The results of this experiment are reported in Fig. 7. We can note that the policy manages to satisfy the performance requirement, with the average measured response time being 1.03s. In general, the response time never exceeded 3s. Increased performance comes at the price of higher resource usage, as the policy keeps 5 active instances for most the experiment duration.

We also evaluate the presented auto-scaling solution using the DS2 policy from [13]. Similarly to the threshold-based policy, this approach is unaware of users' response time requirements. However, differently from the rule-based policy, DS2 does not require the specification of threshold values, which can be difficult to tune. Figure 8 shows the results of this experiment.

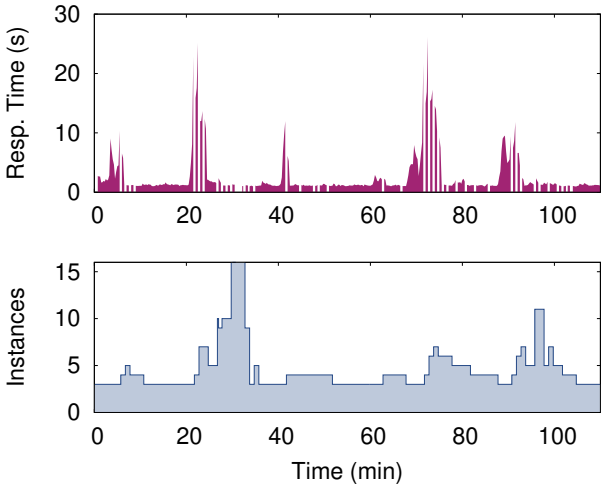

Figure 8: Results with the DS2 policy.

DS2 runs 4.5 function instances on average, slightly less than the queueing-based policy. The measured average response time in this case is 2.7s, much better than the simple threshold-based approach. However, we also observe response time peaks up to 26s. Therefore, this policy provides very good performance on average. Unfortunately, it does not allow response time requirements to be specified and taken into account when planning scaling actions.

Independently from the policies, it is worth noting that the implemented scaling solution allows parallelism adjustments to be enacted with negligible overhead and, in particular, with not significant interruption of the ongoing processing. This is a major advantage compared to the often large overhead required by operator scaling in distributed DSP frameworks (see, e.g., [3, 4]).

6 RELATED WORK

Processing unbounded data sets, once deployed DSP applications usually execute for an indefinite amount of time. As such, applications likely face different working conditions and workloads over time, which require self-adaptation capabilities in order to keep a consistent service level over time. For this reason, it is not surprising that researchers have spent a lot of effort studying the integration of adaptation mechanisms in DSP systems [22], especially as regards run-time resource and deployment management [16].

Because of the workload variability that often characterizes DSP applications, among the available adaptation mechanisms, special attention has been reserved so far to operator auto-scaling [6, 25], which allows DSP systems to elastically acquire and release computing resources as needed. In particular, operators can be scaled either *horizontally* (i.e., changing their parallelism level) or *vertically* (i.e., adjusting resource allocation to operator instances, without altering their number).

The majority of the existing approaches focus on *horizontal* operator scaling, as we also do in this work. Horizontal operator scaling allows application to exploit the parallelism offered by modern distributed infrastructures. However, operator scaling implementation also poses a few challenges, especially in presence of stateful operators. Indeed, in order to preserve stream and state integrity, parallelism adjustments require specific reconfiguration protocols, which often cause significant overhead (see, e.g., [3]). For this reason, researchers have studied enhanced mechanisms to reduce scaling overhead and enable seamless elasticity (e.g., [27, 28]), as we also do in Pulsar.

As surveyed in [25], a variety of techniques have been used to devise operator scaling policies, including heuristics [10, 13, 24], control theory [3, 7, 12], queueing theory [17], reinforcement learning [4, 11]. As regards the architecture adopted for auto-scaling control, most of the existing approaches rely on centralized controllers (e.g., [10, 12, 17]). A few works consider hierarchical control schemes, trying to enjoy both the scalability benefits provided by multiple controllers and the coordination ability of centralized controllers (e.g., [4, 24]). Fully decentralized schemes are the most difficult to handle because of their potential lack of stability. Nonetheless, they have been applied in a few works for auto-scaling (e.g., [1, 20]).

A limited number of works have adopted *vertical* operator scaling so far (e.g., [8, 12, 21]), where the amount of resources allocated to operator instances is adjusted at run-time, rather than the parallelism level. For this purpose, existing works leverage various mechanisms, such as CPU frequency scaling [8], OS-level resource limitation [12] and application-level adaptive thread scheduling [21].

A few works (e.g., [18, 19, 26]) have explored *multi-level* auto-scaling solutions, aiming to adapt operator parallelism along with the number of computing nodes in the infrastructure, so as to reduce operational costs. In Pulsar, it is possible to independently scale each component (i.e., brokers, bookies and function workers). Therefore, we will consider multi-level elasticity for future work.

Pulsar as a message queuing system has been analyzed and compared to other competitors in [9], where the authors have put in evidence Pulsar's flexibility and the availability of a larger number of functionalities with respect to the other message queuing systems. In [15] Khandelwal et al. describe how Apache Pulsar can be used in tandem with Jiffy, a virtual memory layer for serverless applications, to enable stateful computation in the serverless context.

7 CONCLUSION

In this paper we have presented a solution for elastic DSP using Pulsar Functions, which provide a stream processing framework on top of Apache Pulsar, a scalable and distributed messaging system. We organized our auto-scaling solution according to the MAPE pattern for self-adaptive systems and extended Pulsar to enable low-overhead function parallelism adaptation, targeting both stateless and stateful functions. We integrated different auto-scaling policies in our solution, exploiting a threshold-based heuristic, queueing theory and an approach from the literature, which had been previously implemented on top of other DSP frameworks. Our experiments demonstrate the effectiveness of our solution, which enables elastic execution of Pulsar Functions.

As all the considered approaches rely on reactive policies, we plan to extend this work devising proactive auto-scaling policies by means of workload forecasting. Furthermore, we plan to extend the

auto-scaling functionalities implemented in Pulsar to realize a multi-level elasticity solution. By integrating mechanisms and policies for infrastructure-level scaling, we would be able to dynamically provision function workers as needed.

REFERENCES

[1] M. M. Belkhiria and C. Tedeschi. 2019. A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications. In *Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops (LNCS, Vol. 11997)*. Springer, 42–53. https://doi.org/10.1007/978-3-030-48340-1_4

[2] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. 2006. *Queueing Networks and Markov Chains - Modeling and Performance Evaluation with Computer Science Applications, Second Edition.* Wiley.

[3] M. Borkowski, C. Hochreiner, and S. Schulte. 2019. Minimizing Cost by Reducing Scaling Operations in Distributed Stream Processing. *Proc. VLDB Endowment* 12, 7 (March 2019), 724–737. https://doi.org/10.14778/3317315.3317316

[4] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo. 2018. Decentralized Self-adaptation for Elastic Data Stream Processing. *Future Gener. Comput. Syst.* 87 (2018), 171–185. https://doi.org/10.1016/j.future.2018.05.025

[5] G. Cugola and A. Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surveys* 44, 3 (2012), 15:1–15:62. https://doi.org/10.1145/2187671.2187677

[6] M. D. de Assunção, A. Da Silva Veith, and R. Buyya. 2018. Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions. *J Netw. Comput. Appl.* 103 (2018), 1–17. https://doi.org/10.1016/j.jnca.2017.12.001

[7] T. De Matteis and G. Mencagli. 2017. Elastic Scaling for Distributed Latency-Sensitive Data Stream Operators. In *Proc. PDP '17*. IEEE Computer Society, 61–68. https://doi.org/10.1109/PDP.2017.31

[8] T. De Matteis and G. Mencagli. 2017. Proactive Elasticity and Energy Awareness in Data Stream Processing. *J. Syst. Software* 127 (2017), 302–319. https://doi.org/10.1016/j.jss.2016.08.037

[9] G. Fu, Y. Zhang, and G. Yu. 2021. A Fair Comparison of Message Queuing Systems. *IEEE Access* 9 (2021), 421–432. https://doi.org/10.1109/ACCESS.2020.3046503

[10] B. Gedik, S. Schneider, M. Hirzel, and K. Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1447–1463. https://doi.org/10.1109/TPDS.2013.295

[11] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. 2014. Auto-Scaling Techniques for Elastic Data Stream Processing. In *Proc. ICDE '14*. IEEE Computer Society, 296–302. https://doi.org/10.1109/ICDEW.2014.6818344

[12] M. R. Hoseiny Farahabady, A. Jannesari, J. Taheri, W. Bao, A. Y. Zomaya, and Z. Tari. 2020. Q-Flink: A QoS-Aware Controller for Apache Flink. In *Proc. CCGRID '20*. IEEE, 629–638. https://doi.org/10.1109/CCGrid49817.2020.00-30

[13] V. Kalavri, J. Liagouris, M. Hoffmann, D. C. Dimitrova, M. Forshaw, and T. Roscoe. 2018. Three Steps is All you Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *Proc. OSDI '18*. USENIX Association, 783–798. https://www.usenix.org/conference/osdi18/presentation/kalavri

[14] J. O. Kephart and D. M. Chess. 2003. The Vision of Autonomic Computing. *IEEE Computer* 36, 1 (2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[15] Anurag Khandelwal, Arun Kejariwal, and Karthikeyan Ramasamy. 2020. Le Taureau: Deconstructing the Serverless Landscape & A Look Forward. In *Proc. ACM SIGMOD '20*. 2641–2650. https://doi.org/10.1145/3318464.3383130

[16] X. Liu and R. Buyya. 2020. Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions. *ACM Comput. Surveys* 53, 3 (2020), 50:1–50:41. https://doi.org/10.1145/3355399

[17] B. Lohrmann, P. Janacik, and O. Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *Proc. ICDCS '15*. IEEE Computer Society, 399–410. https://doi.org/10.1109/ICDCS.2015.48

[18] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni. 2018. Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems. *IEEE Trans. Parallel Distrib. Syst.* 29, 3 (2018), 572–585. https://doi.org/10.1109/TPDS.2017.2762683

[19] V. Marangozova-Martin, N. De Palma, and A. El-Rheddane. 2019. Multi-Level Elasticity for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.* 30, 10 (2019), 2326–2337. https://doi.org/10.1109/TPDS.2019.2907950

[20] G. Mencagli. 2016. A Game-Theoretic Approach for Elastic Distributed Data Stream Processing. *ACM Trans. Auton. Adapt. Syst.* 11, 2, Article 13 (2016), 34 pages. https://doi.org/10.1145/2903146

[21] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou. 2019. Haren: A Framework for Ad-Hoc Thread Scheduling Policies for Data Streaming Applications. In *Proc. DEBS '19*. ACM, 19–30. https://doi.org/10.1145/3328905.3329505

[22] C. Qin, H. Eichelberger, and K. Schmid. 2019. Enactment of Adaptation in Data Stream Processing with Latency Implications—A Systematic Literature Review. *Inf. Softw. Technol.* 111 (2019), 1–21. https://doi.org/10.1016/j.infsof.2019.03.006

[23] Karthik Ramasamy. 2019. Unifying Messaging, Queuing, Streaming and Light Weight Compute for Online Event Processing. In *Proc. ACM DEBS '19*. https://doi.org/10.1145/3328905.3338224

[24] H. Röger, S. Bhowmik, and K. Rothermel. 2019. Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures. In *Proc. Middleware '19*. ACM, 255–267.

[25] H. Röger and R. Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *Comput. Surveys* 52, 2, Article 36 (2019), 37 pages. https://doi.org/10.1145/3303849

[26] G. Russo Russo, M. Nardelli, V. Cardellini, and F. Lo Presti. 2018. Multi-Level Elasticity for Wide-Area Data Streaming Systems: A Reinforcement Learning Approach. *Algorithms* 11, 9 (2018), 134. https://doi.org/10.3390/a11090134

[27] A. Shukla and Y. Simmhan. 2018. Toward Reliable and Rapid Elasticity for Streaming Dataflows on Clouds. In *Proc. ICDCS '18*. 1096–1106. https://doi.org/10.1109/ICDCS.2018.00109

[28] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang. 2019. Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing. In *Proc. SIGMOD '19*. ACM, 573–588. https://doi.org/10.1145/3299869.3319868