PASCAL: an Architecture for Proactive Auto-Scaling of Distributed Services

Federico Lombardi[a,b,*], Andrea Muti[a], Leonardo Aniello[b], Roberto Baldoni[a], Silvia Bonomi[a], Leonardo Querzoni[a]

[a] *Research Center of Cyber Intelligence and Information Security - Sapienza University of Rome*
[b] *University of Southampton - Cyber Security Research*

Abstract

One of the main characteristics that today makes cloud services so popular is their ability to be *elastic*, i.e., they can adapt their provisioning to variable workloads, thus increasing resource utilization and reducing operating costs. At the core of any elastic service lies an automatic scaling mechanism that drives provisioning on the basis of a given strategy. In this paper we propose *PASCAL*, an architecture for **P**roactive **A**uto-**SCAL**ing of generic distributed services. PASCAL combines a proactive approach, to forecast incoming workloads, with a profiling system, to estimate required provision. Scale-in/out operations are decided according to an application-specific strategy, which aims at provisioning the minimum number of resources needed to sustain the foreseen workload. The main novelties introduced with PASCAL architecture are: (i) a strategy to proactively auto-scale a distributed stream processing system (namely, Apache Storm) with the aim of load balancing operators through an accurate system performance estimation model, and (ii) a strategy to proactively auto-scale a distributed datastore (namely, Apache Cassandra), focussed on how to choose when executing scaling actions on the basis of the time needed for the activation/deactivation of storage nodes so as to have the configuration ready when needed. We provide a prototype implementation of PASCAL for both use cases and, through an experimental evaluation conducted on a private cloud, we validate our approach and demonstrate the effectiveness of the proposed strategies in terms of saved resources and response time.

Keywords:
Cloud, Elasticity, Automatic Scaling, Stream Processing, Distributed Storage, Storm, Cassandra.

1. Introduction

The increasing need of high performance computing in data centers, as well as the recent spread of cloud computing with *pay-as-you-go* pricing models, are fostering the widespread adoption of distributed services. These allow to achieve high levels of availability with low response time despite the heaviness of input loads. Saving costs while ensuring high performance against variable workload is a very challenging aspect that requires the ability to adequately choose at runtime the right amount of computational resources.

Automatic scaling (or *auto-scaling*) emerged as an approach to dynamically provision distributed services in responses to variations in the computing environment, and, in particular, in the incoming load. The goal of any auto-scaling solution is to avoid as much as possible over- and/or under-provisioning states, which would bring the running service to either waste resources or fail to deliver expected performances. Most of the modern solutions in this field work by tuning the level of horizontal scaling of the target service, that is the number of computing resources (either physical hosts, virtual machines or containers) that the service can rely upon for its operations.

A number of auto-scaling solutions have been developed both in commercial and academic areas [1]. Even though such solutions allow to efficiently scale a distributed system, there is still an open problem concerning the timeliness of the reaction to load variations. As an example, if sudden load increments are not spotted "early enough", then temporary performance degradations are likely to occur or, if the reaction happens too late once the load falls down, then it is possible to have short over-provisioning periods. The reaction delay depends on how long does it take to detect the load variation and on the time required for the new system configuration to be ready. In particular, applying new system configurations may require resource and/or service activation and state transfer, which may take very different times depending on the resource/service needed (e.g., the new configuration activation time can change according to the amount of data to transfer). In order to minimise the impact of a configuration change, proactive approaches based on workload forecasting have been proposed, and they show effective results by anticipating scaling decisions [1]. The majority

*Please address correspondence to Federico Lombardi
Email addresses: `lombardi@dis.uniroma1.it;` `f.lombardi@soton.ac.uk` (Federico Lombardi), `muti.1192113@studenti.uniroma1.it` (Andrea Muti), `l.aniello@soton.ac.uk` (Leonardo Aniello), `baldoni@dis.uniroma1.it` (Roberto Baldoni), `bonomi@dis.uniroma1.it` (Silvia Bonomi), `querzoni@dis.uniroma1.it` (Leonardo Querzoni)

of proactive solutions work under the assumption of load balancing among the machines, providing a coarse grained workload prediction. However, this assumption does not hold in several scenarios like, e.g., in distributed stream processing systems, and prevents them to be used in an effective way. In such kind of systems a computation is modelled as a directed acyclic graph of operators, which are allocated to available machines on the basis of some scheduling policy. In general, the input of an operator depends on several parameters like the stream processing system workload, the topology of operators graph, and the scheduling of operators on machines. Thus, when the stream processing system workload is the only parameter that changes, the load balancing assumption does not hold anymore and current proactive schemes cannot be directly applied. Some works do not address the problem at all, and assume in a simplistic way that for a given input load there exists a correct minimum amount of machines able to handle it, regardless of how operators are allocated to machines and of the heterogeneity of load among operators [2]. A similar problem arises when considering distributed storages where the load of each machines depends on the subset of data allocated on it and that can be queried differently.

In this paper we introduce PASCAL, an architecture for proactive auto-scaling of distributed services. PASCAL works by learning and predicting workload patterns at fine grained level (e.g., at the level of the operator in stream processing systems), and then using this knowledge to dynamically provision the required amount of resources to the target system. The main novelties introduced by PASCAL are (i) the possibility to accurately tune the timing for (de-)provisioning resources by taking into account the time needed to complete such operations, and (ii) a model for estimating system performance even when the load is not balanced among resources.

Being PASCAL a generic solution, we propose an instantiations on two very relevant scenarios presenting complementary issues, i.e. a (i) a distributed stream processing system (dSPS) where load between resources may be temporary unbalanced due to workload oscillation and operator allocations, making challenging to estimate their performance, and (ii) a distributed datastore where the time to complete scaling actions is strictly dependent by the amount of data to transfer to keep the storage updated and may be potentially very high.

Although PASCAL architecture is general and application-independent, the implementation of its modules, including the one executing the scaling strategy (see Section 4) depends on the specific application scenario. We designed two distinct auto-scaling strategies for the considered two scenarios, implemented them in PASCAL, and integrated it within two prototype deployments, one working with *Apache Storm* (a framework for distributed stream processing applications) and the other with *Apache Cassandra* (a NoSQL distributed datastore). The extensive experimental evaluation performed on these two prototypes demonstrates PASCAL's ability to timely and accurately provision resources to the target system, thus reducing operating costs without negatively impacting on performance.

The rest of the paper is organized as follows: related works are presented in Section 2; Section 3 introduces the system model and defines the problem to solve; Section 4 describes the PASCAL architecture; Section 5, 6 present the solutions and prototype implementations proposed for distributed stream processing and distributed datastore scenarios, respectively; Section 7 shows the experimental evaluation of the two prototypes; finally, Sections 8 and 9 outline conclusions and future directions.

2. Related Work

The design of PASCAL is based on the lessons learned by the authors while developing and deploying in real settings the MYSE architecture [3], a former prototype of a proactive auto-scaling solution. MYSE takes an analytical-based approach by relying on queuing theory to estimate service performance (i.e., the expected service time), with the assumption of balance among nodes' load, and uses machine learning for workload prediction. With PASCAL we aim to address the main limitations of MYSE, i.e. (i) the analytical model based on queuing theory, which is not accurate enough in realistic use cases, and (ii) the assumption of load balancing, which made MYSE not usable in several scenarios of interest, such as dSPS.

Regarding the latter point, it is to note that most auto-scaling solutions are based on the assumption of load balancing among nodes to simplify the problem of estimating performances. This kind of approach is specifically designed for cloud web applications based on multi-tier architectures, such as [4], making it unfeasible to use in environments where load cannot be expected to be balanced. Many works indeed assume load balancing, thus suffering from the same limitation of MYSE. Ghanbari et al. [5] present an auto-scaling approach based on *model predictive control*, which aims to meet service level agreements (SLAs) and save resources. They target clouds with heterogeneous resources, and employ Amazon EC2 instances that balance the load through a round robin policy. Moore et al. [6] describe an elasticity framework composed by two controllers operating in a coordinated manner: one works reactively on the basis of static rules, and the other uses a time-series forecaster (based on support vector machines) and two Naive Bayes models to predict both the workload and the target system performance. Also this work assumes load balancing between service instances, as well as a fixed time interval between consecutive scaling actions (30 minutes in their experimental evaluation). In [7], the authors propose DC2, a proactive solution that does not need any application profiling. They infer system parameters through Kalman Filters. Also in this work perfect load balancing is assumed.

The solution we propose in this paper does not assume load balancing among nodes and uses a proactive approach. Although a reactive version of PASCAL is mentioned in Section 7, its sole purpose is to quantitatively show the advantages of proactivity over reactivity. While Subsection 2.1 describes related work on reactive techniques for automatic scaling, Subsection 2.2 delves with existing works following a proactive strategy and their comparison with PASCAL.

A relevant feature of PASCAL, inherited from MYSE, is its general purpose design to make it usable for distinct types of systems. A number of works addressed the auto-scaling problem by designing solutions for specific scenarios such as high performance computing [8], publish-subscribe systems [9], computation platforms [10], cloud datacenter [11, 12] and smart grids [13]. Most of these solutions are strictly application-dependent and cannot be directly applied in a different context. Contrarily, PASCAL has been designed to be application-independent and employable in different scenarios. Indeed we show the design, prototype implementation and evaluation of PASCAL for dSPSs and distributed datastores. We report for completeness a discussion on related work on automatic scaling for dSPSs (see Subsection 2.3) and distributed datastores (see Subsection 2.4).

2.1. Reactive Auto-scaling

In their survey on automatic scaling, Lorido-Botrán et al. [1] state that the most widely used approach by Cloud providers relies on *static, threshold-based policies*, where the system configuration is changed according to a set of *rules* [14, 15, 16]. All these approaches are *reactive*, i.e. they monitor the workload and react to changes by re-provisioning the system. Due to long delays needed to activate new VM instances [1], reactive approaches can sometimes lag-behind frequent workload fluctuations, leaving the system in a state where it is underperforming or it is over-provisioned.

2.2. Proactive Auto-scaling

Proactive approaches are starting to be used as a way to anticipate scale-in/-out operations, hence mitigating some limitations of reactive approaches. Nevertheless, it is worth noticing that proactive approaches can provide clear advantages over reactive ones only in those scenarios where historical data is available to correctly train prediction models and where such models do not get outdated too quickly due to the dynamic nature of the considered setting.

Proactive approaches are based on *time-series analysis* to learn recurring workload patterns over time through techniques like averaging methods, regression and artificial neural networks (ANNs) [18, 19, 20, 21]. While reactive solutions allow to observe metrics of interest to decide how to take scaling decision, proactive approaches need to accurately estimate the performance that the distributed service would provide when fed with the foreseen workload. More in details, [18, 19, 21] use time-series analysis to dynamically adjust resources in a machine (i.e., *vertical scaling*) while in PASCAL we take an orthogonal perspective using the estimation to scale the number of machines used to provide the service (i.e., *horizontal scaling*). In [20], the authors focus on horizontal scaling proposing an architecture similar to PASCAL. However, the scaling decision is driven by a different objective function with respect to the one considered in this paper and the evaluation in performed just through simulations and not in real environments. Additionally, in [18, 19, 20, 21] the time window used to perform scaling decision is fixed while we designed a solution able to estimate when to trigger scaling actions to have a configuration ready when necessary, thus limiting over/under-provisioning periods. Some works employ *Reinforcement learning* to automatically learn online the performance model of the target system without any a priori knowledge [22, 23, 24]. These works mainly focus on improving the learning phase used to perform scaling actions and do not really provide a real system. Some works use *Queuing Theory* to analytically estimate the performance of the target system given a small set of parameters, like input rates and service times [25, 26]. The main limitation of these works is in the usage of an analytical model to estimate scaling actions that does not always abstract real systems executions. Other works employ *Control Theory* to automate the management of scaling decisions through the employment of a feedback/feedforward controller [27, 28, 29].

2.3. Autoscaling Distributed Stream Processing Systems

Heinze et al. present a set of works that are closely related to PASCAL. In [30] they adapt *threshold-based* and *reinforcement learning* solutions presented in [1] to auto-scale a stream processing system in a reactive manner. They also designed a latency-aware approach [31] and a solution to properly choose the scaling strategy through online parameter optimization [32]. PASCAL differs from such solution since it is able to estimate the performance of a dSPS instead of reacting to observed metrics. This feature allows it to auto-scale the dSPS proactively.

Other works combine elasticity and fault tolerance. In [33] the authors consider the problem of scaling stateful operators deployed over a large cloud infrastructure. Also in this case, the approach adopted to scale the system is purely reactive. Ishii et al. [2] propose a proactive solution to move part of the computation to the cloud when a local cluster becomes unable to handle the predicted workload. They use a simple benchmark to map a workload to the number of nodes required to sustain it. The proactive model we propose is more fine-grained due to a resource estimator that allows to accurately compute the expected resource consumption given an input load and a configuration (see Section 5). Similarly, ELYSIUM [34] uses an

[1] instance startup times in EC2 may range from 10 to 45 minutes, as reported in [17].

approach for symbiotic scaling of operators and resources. Both PASCAL and ELYSIUM rely on a performance profiler but they have different goals: ELYSIUM combines operator scaling and resource scaling, PASCAL defines a solution to enable a proactive scaling of resources in a dSPS. Recently, some efforts have been done to extend DSP architecture with elasticity capabilities at different level.

In [35], Cardellini et al. propose a hierarchical architecture for DSP systems for geo-distributed environment. In such architecture, adaptation is done by employing Reinforced Learning techniques. In [36], Mencagli et al. presented a two-level adaptation solution that handles workload variations at different time-scales: (i) fast time-scales (using a control theory based approach to deal with load imbalance) and (ii) slower time-scale (using fuzzy logic to for scaling decisions). However, both approaches are reactive and not proactive as PASCAL, not requiring any mechanism to estimate the system performance.

A similar approach has been followed by [37] where the proposed methodology is able to control the number of replicas in streaming operators. The authors proposed two algorithms to be applied at different time-scales. but with a different role with respect to [36]. Also in this case the approach is reactive.

Concerning operators scaling, Liu et al. [38] proposed a stepwise profiling framework that considers both application features and processing power of the computing resources to evaluate different configurations of parallelism.

Stela [39] relies on a throughput-based metric to estimate the impact of each operator towards the application throughput to identify operators that need to be scaled. An interesting evolution is the one presented in [40] where the authors present the development of an elastic stream processing framework for IoT environments. Also this approach is reactive and it is mainly studied for long-term steady load variations that do not causes the triggering of continuous reconfiguration actions.

2.4. Autoscaling Distributed Datastores

Several solutions have been published for automatic scaling of SQL and NoSQL storage systems. In [41] the authors propose ElastMan, an elasticity manager for key-value stores in the Cloud. Their work relies on both feed-forward and feedback controllers for effective scaling actions. The main differences with our approach are that (i) they use a reactive approach and (ii) they do not consider activation/deactivation time to match the demand point with the time when the configuration is ready. Different solutions have been proposed at distinct *XaaS* levels. Many works (e.g. AGILE [42]) address the scaling problem from a IaaS perspective. Likewise our solution, they profile the target system and apply AGILE to scale a Cassandra cluster. Their work is orthogonal with respect to PASCAL, indeed their goal is to minimize the start-up time of a new instance through a pre-copy live cloning approach at VM level (i.e. they act at IaaS level), while we act at the datastore level (PaaS level). Barker et al. propose ShuttleDB [43], a solution for database live migration and replication which combines VM level and database level scaling. They monitor the query latency and/or predict it to trigger scaling actions. Our approach, differently from theirs, also considers the activation time of newly provisioned nodes. Huang et al. [44] deal with the auto-scaling and data distribution of a MongoDB datastore. Their solution scales a sharded MongoDB cluster reactively, while we operate proactively. Finally, Casalicchio et al. proposed an energy-aware autoscaling solution for Cassandra [45] where they evaluate and compare three heuristics under a model similar to ours, with homogeneous machines and assuming a CPU-bound workload. Their results show that horizontal scaling of Cassandra is slow due to the amount of data to move. Differently from our work, they do not provide a solution to dynamically trigger scaling actions based on time.

3. System Model and Problem Statement

3.1. Distributed Service Model

We consider a cluster as a fixed set with cardinality N of available servers (or *nodes*), i.e., virtual or physical machines. Servers are homogeneous, they have the same finite computational power and can be used interchangeably. We consider a *distributed service*, consisting of a set of *service instances* deployed over the available nodes. At most one service instance is deployed on each node. Any service instance can be in one of these four states:

- *inactive*: the service instance is not running, thus it is not part of the distributed service;
- *active*: the service instance is running, thus it is part of the distributed service;
- *joining*: the service instance is becoming part of the distributed service but is not active yet;
- *decommissioning*: the service instance is leaving the distributed service but is not inactive yet.

3.2. System Configuration Model

We refer to *configuration* as the number of active service instances (i.e., service instances in the active state), which also corresponds to the number of used nodes. The configuration can change over time in response to *scaling actions*: *scale-in* actions reduce the configuration by taking active service instances to inactive state, while *scale-out* actions augment the configuration by making inactive service instances become active (see Figure 1). Any state management for the distributed service, required to keep the service state across reconfigurations, is assumed to be handled by the service itself. A scaling action take a time T_{sa} which depends on the number of service instances to provide/remove: we refer to *joining_time* as the time needed for a service instance to switch from the inactive state to the active one during a scale-out action.

Conversely, we refer to *decommissioning_time* as the time needed for a service instance to switch from the active state to the inactive one during a scale-in action.

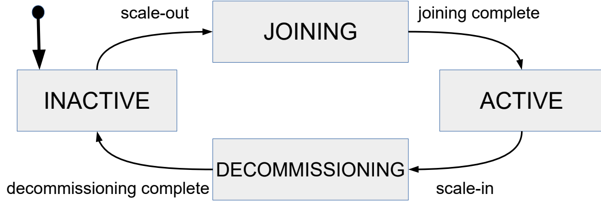


Figure 1: State diagram of a service instance: in response to a *scale-out action*, it transitions to the *joining state*, which takes *joining_time* to complete and move to the *active state*, where the service instance can serve requests. A *scale-in action* brings an active service instance to *decommissioning state*, which takes *decommissioning_time* to complete and transition to *inactive state.*

A scaling action sa is characterised by three points in time [46]: (i) *sa triggering point* (TP_{sa}), when the scaling action is triggered and the service instance transitions from inactive (active) to joining (decommissioning) state, and (ii) *sa reconfiguration point* (RP_{sa}), when the scaling action completes and the service instance passes from joining (decommissioning) to active (inactive) state. Finally, the *sa demand point* (DP_{sa}), the point in time when the new configuration is needed. Note that for a scaling action sa, joining/decommissioning times can be computed as the distance between TP_{sa} and RP_{sa}.

3.3. Workload and Performance Model

A number of clients interact with the distributed service by sending request messages. We refer to *input rate* or *workload* $\lambda(t)$ as the number of requests per time unit issued by clients towards the distributed service at a given time t. We consider also a temporal *horizon* h in which we can aggregate the workload, thus we define $\lambda_{max}(h)$ as the maximum workload for $t = t_0, \cdots, t_h$.

Once a request is received by the distributed service, a certain amount of time is required to serve it. We refer to that time as the *response time*, and to the number of requests served per time unit by the distributed service as the *throughput*. Depending on the specific application scenario where the distributed service is employed, provided performances (i.e., response time and throughput) may need to ensure certain properties in spite of possible changes of the workload during time, e.g., response time should not exceed a given upper bound, or should not diverge over time. For example, if the workload were too high for the current configuration and response times started increasing, then a scale-out action would be required to keep response times below a given upper bound. A distributed service, setup with a given configuration, is said to *sustain* a certain workload if provided performances satisfy a given set of application-specific requirements. On the base of whether the current configuration is enough to sustain incoming workload, a distributed service can be in one of these two states:

- *normal*: the distributed service *sustains* current workload with the present configuration;
- *overloaded*: the distributed service *does not sustain* current workload with the present configuration.

If the distributed service is overloaded, there likely is some bottleneck to be solved through a scale-out action. Otherwise, maybe a scale-in action is required to decrease the number of used nodes and consequently save resources (e.g., save money by using less nodes).

Under the assumption that sa would set a configuration such to bring the distributed service in normal state, the ideal situation would be having $RP_{sa} = DP_{sa}$, i.e., the new configuration is ready exactly when it is required. In real settings, for a scaling action sa we have that one of these two situations occurs:

- $RP_{sa} > DP_{sa}$: the scaling action completes late of a time interval of length $RP_{sa} - DP_{sa}$. In case of a scale-in action, this is a period of *over-provisioning* (since a smaller configuration could sustain the workload), while in case of a scale-out action this is an *under-provisioning* period (since there should be more active service instances).
- $RP_{sa} < DP_{sa}$: the scaling action finishes in advance of a time interval of length $RP_{sa} - DP_{sa}$. In case of a scale-in action, this is a period of *under-provisioning* (since a larger configuration should be needed to sustain the workload), while in case of a scale-out action this is an *over-provisioning* period (since there should be less active service instances).

3.4. Problem Statement

The goal is to find the *minimum configuration*, i.e. the smallest configuration such that the distributed service is in not overloaded (i.e., in normal state), and thus able to sustain the input workload.

Furthermore, as scaling actions need some time to complete, the goal is identifying the minimum configuration early enough to minimize over-provisioning and under-provisioning periods. Thus, for the required scaling action sa we want to minimize $|DP_{sa} - RP_{sa}|$.

Since we assume the workload to be dynamic, the minimum configuration has to be computed periodically over time. We refer to this computation period as *configuration assessment period*. We assume that computing the minimum configuration takes much lower than the configuration assessment period, thus consecutive minimum configuration computations do not overlap.

4. PASCAL Architecture

PASCAL works in two consecutive phases: a *profiling phase* and an *auto-scaling phase*. The basic idea that underlines PASCAL is to use machine learning techniques to

learn, during the profiling phase, the workload patterns (i.e., the *workload model*) and performance behaviours (i.e., the *performance model*) typical for the target distributed service. These models are then used at runtime, during the auto-scaling phase, to proactively scale the distributed service: future input rate is predicted on the base of the workload model, the corresponding minimum configuration is estimated using the performance model, and the consequent scaling action is triggered to minimise over/under-provisioning periods.

The architecture of PASCAL includes three main functional modules: (i) a Service Monitor collecting during both phases the metrics related to the target distributed service, (ii) a Service Profiler implementing the first phase and (iii) an AutoScaler for the second phase. Figure 2 shows PASCAL's functional architecture and how it integrates with the target distributed service.

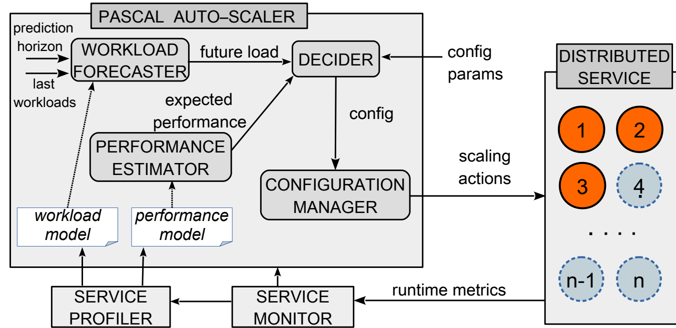


Figure 2: PASCAL's functional architecture integrated with the target distributed service.

Next sub-sections detail these three main functional modules. The instantiation of this high-level architecture heavily depends on the peculiar characteristics of the distributed service to auto-scale. Anyway, its constituting modules capture the key aspects of realising a proactive auto-scaling for a broad range of distributed services, thus PASCAL's architecture constitutes an effective guideline to instantiate specific solutions, as will be shown for the two application scenarios detailed in Sections 5 and 6.

4.1. Service Monitor

The *Service Monitor* monitors at runtime a set of metrics (such as CPU and memory) related to service instances. Metrics are collected periodically and used in both profiling and auto-scaling phases. They include the input rate, to be used (i) to learn the workload model during the profiling phase and (ii) to predict future workload during the auto-scaling phase. The other included metrics depend on the specific application scenario, e.g., the type of distributed service, the algorithm used to compute the minimum configuration.

4.2. Service Profiler and Performance/Workload Models

The *Service Profiler* is used during the profiling phase to learn the performance model and the workload model on the base of the metrics collected by the Service Monitor.

Different workloads are provided to the target distributed service for sufficiently long time, and related performance metrics are collected. The duration of the profiling phase depends both on the distributed service and on the algorithm used for minimum configuration computation[2]. Collected metrics are processed to learn the *performance model*, which is used by the AutoScaler in auto-scaling phase to estimate the performance of the distributed service under a certain workload.

Real workloads in input to the target distributed service are observed over time, and related metrics are gathered. These metrics are analysed to learn the *workload model*, aimed at providing predictions as accurate as possible about what the workload will be like during the considered prediction horizon.

4.3. AutoScaler

The AutoScaler comes into play in the auto-scaling phase. Every configuration assessment period, it uses its internal modules to (i) predict forthcoming input rate, (ii) estimate the minimum configuration and (iii) possibly trigger the required scaling action, if current configuration differs from the minimum one.

The *Workload Forecaster* is in charge of forecasting the input rate over some prediction horizon h. It exposes a primitive *predict()* which requires as input the list w of the workloads monitored during the last *observation period*, and outputs the expected workloads $\lambda_f(h)$ during the forthcoming prediction horizon h. The lengths of observation and prediction periods, as well as the sampling rate of observed workloads, and how many distinct workloads to forecast during the prediction horizon, have to be chosen at a preliminary stage (i.e., before starting the profiling phase), as they relevantly affect how workload metrics are gathered and what machine learning technique to use to train the workload model. Note that the observation period w and the forecast horizon h can be quite different. Specifically, the observation period impact the accuracy of the timeseries prediction [47], while the prediction horizon is used to avoid oscillations by providing an estimation of the maximum future workload in the future h time unit. Thus, while the observation period can be very long producing a good accuracy, a too long prediction horizon can lead to high over-provisioning and so it has to be tuned properly (we evaluate how to set this period in Section 7).

The *Performance Estimator* is in charge of estimating the performances of the target distributed service according to the *performance model* learned during the profiling phase. It exposes the primitive *estimate()* which, taking as input the expected input rate $\lambda_f(h)$ and a configuration,

[2]In Section 7, we evaluated this time empirically for the two considered application scenarios.

outputs a set of metrics describing the estimated performances of the distributed service. The set of output metrics depends on the specific distributed service and what are the performance indicators of interest in the application scenario.

The *Decider* module, through the primitive *getMinConfig()*, computes the minimum configuration to sustain a forecasted workload and sends the consequent scaling actions to the Configuration Manager. The Decider leverages the Workload Forecaster to predict workloads, and the Performance Estimator to estimate the expected performances of the distributed services given a certain configuration and an expected input rate.

The *Configuration Manager* is in charge of applying a configuration. It accepts scaling actions from the Decider and applies the requested configuration to the target distributed service by activating (or deactivating) service instances.

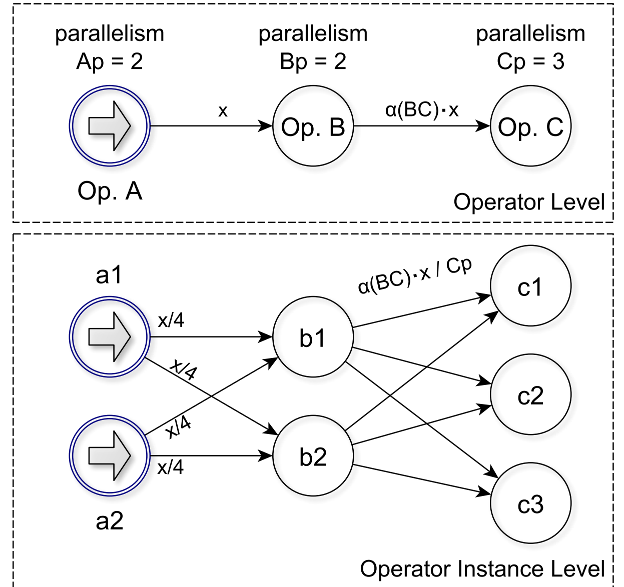


Figure 3: Example of a simple DAG representing a dSPS application. The above square shows the logic of the application (operator level). The square below represents as operators can be parallelized at operator instance level. Those are the actual thread that will be deployed over the dSPS worker nodes.

5. PASCAL for Distributed Stream Processing System

Distributed stream processing systems represent the most widely adopted solution for on-line processing of large data streams [48]. A computation in a dSPS is modelled as a Directed Acyclic Graph (DAG) where vertices represent *operators* and edges represent *streams* of tuples connecting operators. Such a graph is referred to as *application*. There can be more applications running in a dSPS. Each operator carries out a piece of the overall computation on tuples received from input streams, and emits downstream the results of its partial elaboration through output streams. In general, an operator has a certain number of input streams (none for source operators) and output streams (none for sink operators). Each application is also characterized by a workload that varies over time and represents the rate of tuples fed to the dSPS for such application through its source operators. Each tuple fed in input to the dSPS potentially generates multiple tuples that traverse several streams in the application. The processing of some of these tuples may fail, and in this case we say that the input tuple is *failed*. Conversely, if all the tuples generated from a given input tuple are correctly processed, then we say that the corresponding input tuple is *acked*.

For the sake of simplicity, and without loss of generality, we assume that a stream connecting operators A (upstream operator) and B (downstream operator) can be uniquely identified by the pair (A, B), which means that no two distinct streams can connect the same pair of operators. The square above of Figure 3 shows an example of a DAG representing an application with three operators A,B and C. The *selectivity* for a stream (A, B) is defined as the ratio α between the tuple rate of (A, B) and the sum of the tuple rates of all the input streams of A, i.e. the selectivity of (A, B) measures its tuple rate as a function of the total input rate of A [49]. We assume to work with applications having operators with constant selectivity, as in the case of *synchronous data flow* languages and signal processing applications [50]. In practice, as will be shown in Section 7.1, the solution we propose for dSPS works well also when selectivities have some reasonably small deviations.

Each operator can be instantiated multiple times, so that each operator instance (not to be confused with a service instance in our system model) handles a fraction of the operator input rate. Here we assume that each operator instance processes input tuples sequentially by using a single core at a time. The number of instances for a given operator is defined as its *level of parallelism*, and we assume that it is fixed at application deployment time and does not change over time. As a consequence, a stream (A, B) can be constituted by several sub-streams, each connecting one of the instances of operator A to one of the instances of operator B, as it possible to see from the bottom square of Figure 3. To simplify the discussion, here we assume that the dSPS is able to fairly distribute the load among the available instances of each operator. This is usually achieved through round-robin policies, hashing or by means of grouping functions that manage how tuples in a stream are mapped to its sub-streams. When an application is run, the dSPS uses a scheduler to assign the execution of each operator instance to a node among those available in the dSPS cluster.

5.1. Auto-scaling Solution Outline

A dSPS able to sustain a workload maintains a throughput almost equal to the input rate, with not di-

verging response times. Once a dSPS is no more able to sustain a workload, response times start diverging and the throughput begins to fall behind the input rate. The bottleneck is usually due to some saturated node, i.e., a node not able to keep up with the rate of tuples that are being received by the operator instances it is executing. Note that even a single saturated node can lead to a bottleneck, so it is important to keep all nodes under a certain usage. Thus, with reference to our system model, we consider a dSPS cluster to be in *overloaded* state if at least one node has its $CPU_usage \geq max_threshold$, in *normal* state otherwise. We consider the CPU as indicator as it is the most prominent bottleneck for dSPS as many works stated and used in the state of the art [51, 52]. We aim to include also memory and bandwidth in a more complete model as future work.

Hence, one of the main challenges in designing a proactive auto-scaling solution for a dSPS lies in the ability to accurately estimate the CPU usage of nodes. In many scenarios, the load can be considered as balanced among cluster nodes (see Section 2), which relevantly ease the estimation of CPU usage. Instead, in a dSPS, the load generally differs among nodes as it depends on what operator instances are running on each node, and on the rate of tuples that each operator instance is receiving. Indeed, many allocation optimizations for dSPSs have been proposed in literature [50, 49].

With reference to the PASCAL architecture described in Section 4 and shown in Figure 2, the Service Profiler for a dSPS is composed by four sub-modules [3]: (i) the *Selectivity Profiler* (SP), which learns the selectivity of each stream, (ii) the *Operator CPU Usage Profiler* (OCUP), which learns how the CPU usage of each operator instance varies as a function of its input rate, (iii) the *Overhead Profiler* (OP), which learns how to estimate the CPU usage of a node when having at disposal the CPU usages of the operator instances running on that node, and (iv) the *Input Load Profiler* (ILP), which learns the *workload model*, used to predict the workload of each application. The output of the first three sub-modules is the *performance model*.

On the base of performance model, the Performance Estimator for a dSPS has to estimate the CPU usage of each node, given the expected input rate of each application and the allocation of operator instances to nodes. When its *estimate()* method is invoked, the Performance Estimator works as follows:

1. using the selectivities provided by the SP and the input rates of each application, it estimates the tuple rates of all the streams of these applications (i.e., streams among operators, not sub-streams among operator instances);
2. having an estimation of the input rate of each operator, it assesses the input rate for each of each operator instance by simply dividing by the level of parallelism of the operator itself;
3. using the expected input rate of each operator instance, it estimates its CPU usage leveraging the profiles generated by the OCUP;
4. with the given configuration and the allocation of operator instances to nodes, it uses such CPU usage estimations to compute an evaluation of the overall CPU usage of each node, which can be then refined by exploiting the profiles learned by the OP.

[3]The profiler proposed here has been also used to enable a comparison with the reactive approach of the symbiotic solution proposed in the ELYSIUM autoscaler [34].

Every *configuration assessment period*, the Decider executes the `getMinConfig()` function (see Algorithm 1). This function takes as input the prediction horizon h, the cluster size N, the *max_threshold* parameter, the reference to the dSPS scheduler and the list of running applications. Firstly, it gets the last workloads monitored during the last observation period and, through the `predict()` primitive of the Workload Forecaster module, computes the forecasted workload $\lambda_f(h)$ for the next prediction horizon h (lines 2-3). Then, it starts looking for the minimum configuration by iterating in ascending order over the possible configurations (lines 4-8). For each iteration, it uses the dSPS scheduler to compute the allocation of operator instances to service instances. Then, through the Performance Estimator, it computes the expected CPU usages of used nodes. If every node is expected to have a CPU usage under the *max_threshold* parameter, then the configuration is the minimum one.

Algorithm 1 AutoScaling Algorithm for SPS

1: **function** GETMINCONFIG(int h, int N, double *max_threshold*, Scheduler s, List⟨App⟩ *apps*)
2: $w \leftarrow getLastWorkloads()$
3: $\lambda_f(h) \leftarrow predict(t_h, w)$
4: **for** $conf \leftarrow 1$ to N **do**
5: $allocation \leftarrow s.allocate(apps, conf)$
6: $cpuUsages \leftarrow estimate(allocation, \lambda_f(h))$
7: **if** $\forall x \in cpuUsages : x < max_threshold$ **then**
8: **return** $conf$
9: **return** N

5.2. Implementing PASCAL in Storm

In this paper we consider Apache Storm [53] as reference dSPS. In Storm, operators are referred to as *components* and the application is called *topology*. Source components are called *spouts*, while operators that perform computations on streams are named *bolts*. Spouts are usually simple wrappers for external sources of tuples, that generate the input load for an application. At runtime, each component is executed by a configurable number of threads, called *executors* (i.e., the operator instances).

A Storm cluster comprises a single master machine (*Nimbus*), which coordinates a number of slave machines (*Supervisors*). Each Supervisor provides a fixed number

of Java processes where to run executors, namely *workers*. Each worker can only run the executors of a single topology, and represents a *service instance* in our system model. As a rule of thumb, each topology should use a single worker per Supervisor in order to avoid the overhead of inter-process communication. With reference to our system model, the *nodes* are the Supervisors. The Nimbus can dynamically vary the number of active workers for a topology through a *rebalance* operation, which corresponds to trigger *scaling actions* in our system model. We consider the joining/decommissioning_time as the period required for the Nimbus to change the state of the workers during a rebalance operation. In Storm this operation takes a few seconds, hence we can consider $TP_{sa} \approx RP_{sa}$, thus we consider those periods as negligible[4].

With respect to our system model, tuple rate acked over time corresponds to the *throughput*, and the time it takes for a tuple from its emission by a spout to its acking by some bolt represents the *response time*.

Figure 4 shows the integration of PASCAL within a Storm cluster. In the following paragraphs we will describe the implementations of each subsystem.

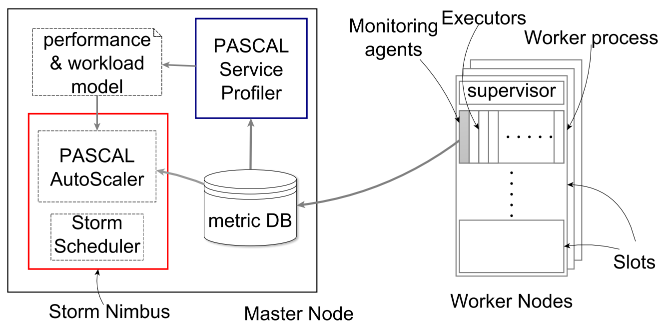


Figure 4: PASCAL integrated within Storm. The node on the left represent the cluster coordinator which executes the Storm Nimbus node, the metric DB and the PASCAL autoscaling system. The blue square represents the PASCAL Service Profiler which acts in the profiling phase and produces the performance and workload models. The red square represents instead the modules involved in the autoscaling phase, thus the PASCAL AutoScaler and the Nimbus which triggers the scaling actions. The nodes on the right highlights the internal module of a Storm Worker.

Service Monitor Implementation

The Service Monitor comprises a set of *monitoring agents*, which are threads running inside workers (see Figure 4) and monitoring several executor metrics, by leveraging a metrics framework provided by Storm itself [54]. Monitored metrics are (i) the rate of tuples emitted by spouts (to monitor the workload), (ii) the rate of tuples received by bolts (to monitor inter-operator traffic), (iii) the CPU usage of the executors, and (iv) the CPU usage of the workers. All these metrics are periodically stored into an Apache Derby DB [55] used as a *metric DB* hosted on the Nimbus.

Service Profiler Implementation

The Service Profiler is implemented through a series of Java modules which realise its sub-modules. They access the metric DB, extract the required dataset and produce their output as Java objects serialized to files, which actually are the *performance model* and the *workload model*.

The SP computes the selectivities for each stream by averaging collected selectivities over time. This is motivated by the initial assumption that selectivities remain stable during the execution, as also confirmed by the empirical results shown in Section 7.1.

The OCUP and OP use Artificial Neural Networks [5] (ANNs) as output functions. Each ANN is trained with the dataset resulting from the profiling phase. The OCUP employs an ANN for each operator; each one has one input node for the input rate, and an output node for the estimated corresponding CPU usage. In a similar way, the OP uses a single input node with the aggregated sum of CPU usages of operator instances and a single output node for the estimation of the worker node CPU usage.

The ILP instead employs an ANN which combines a timeseries approach with a date approach, so it takes as input (i) features extracted from the current timestamp and (ii) the input loads seen in the last w minutes. The number of hidden layers and neurons in each layer is tuned empirically (see Section 7). The outputs of the ANN are the input loads predicted for each next minute up to the prediction horizon. With the aim of over-provisioning over the next prediction horizon, the function learned by the ILP simply returns the maximum of these predicted input loads. We use Encog [56] to configure all profilers' ANNs.

Auto-Scaler Implementation

The AutoScaler is implemented as a Java library to be imported by the Nimbus and it is invoked periodically with period equals to the chosen assessment period. In this way, assessments are executed at the right frequency and have access to all the required information about allocations. The Workload Forecaster module loads the ANN previously serialised by the ILP and the workload model to compute the future workload.

The Performance Estimator loads the performance model i.e. the profiles produced by the Service Profiler (selectivities, OCUP ANN and OP ANN) to estimate the performance.

The Decider implements the scheduler interface and executes Algorithm 1. It wraps the default scheduler of Storm and uses it to simulate allocations when checking the effectiveness of configurations. In case the chosen configuration is different from the current one, it issues a re-

[4] We experimentally evaluated those periods in Section 7.1.

[5] We chose Feed-Forward ANNs after an intensive empirical evaluation of different estimators, such as Recurrent Neural Networks, Elman and Jordan.

balance operation through the Nimbus API to apply the new configuration.

6. PASCAL for Distributed Datastores

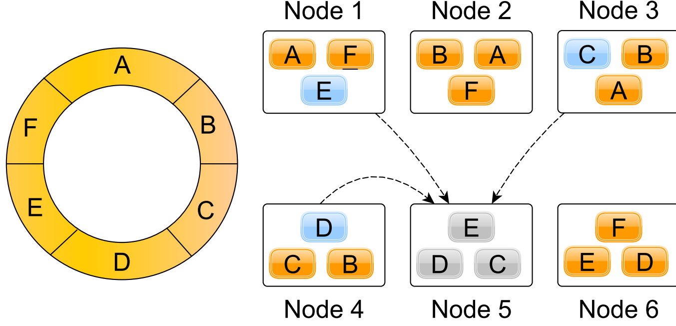


Figure 5: Representation of the datastore model. The ring on the left represents different chunk of data, and on the right is depicted how nodes handle those data. The node #5 represents a new node joining the cluster which receive the blue parts of data from nodes #1, #3 and #4.

We consider a *key-value* datastore, distributed over a cluster of nodes Each node stores a fraction of the whole data, managed by the datastore instance (i.e., service instance) running on that node. The amount of stored data is balanced among nodes. Each *value* associated to a *key* is replicated among nodes with a given replication factor α. Scaling operations lead to data migration. Following a scale-out operation, joining datastore instances receive from already active instances the portions of data they will have to managed once the scale-out will complete. Contrarily, as a result of a scale-in operation, decommissioning datastore instances transfer their data to remaining instances. Figure 5 presents an example of such kind of datastore with the new Node #5 which obtains chunk of data from other active nodes.

We refer respectively to $T_{add}(data)$ and $T_{rem}(data)$ as the time to add and to remove a node according to the amount of *data* stored in the database.

In many real distributed datastores, adding/removing more nodes per time is not recommended because it may lead to consistency issue, especially for system based on eventual consistency, thus the best practice is to add/remove one node per time [57], hence we can consider $T_{sa} \approx T_{add|rem}(data) \cdot x$ to be the time of a scaling action that involves x service instances. Those reconfiguration periods, contrarily to stream processing scenario, are not negligible, hence we cannot consider $TP_{sa} \approx RP_{sa}$.

We consider the distributed datastore able to sustain a workload if it maintains a throughput almost equal to the input rate, with not diverging response times. Once a cluster is no more able to sustain a workload, response times start diverging and the throughput begins to fall behind the input rate. Thus, with reference to our system model, we consider a distributed datastore to be in *overloaded* state if it is unable to sustain an input rate with response times under a certain *max_threshold*, in *normal* state otherwise.

6.1. Auto-scaling Solution Outline

Let us recall that the main challenge in auto-scaling lies in correctly anticipating the TP in order to make the RP matches the DP. This issue is even more important when dealing with datastore where the time required to activate/deactivate a service instance is non-negligible due to the potential huge amount of data that must be transferred to preserve consistency.

Algorithm 2 AutoScaling Algorithm for Datastore

1: **function** GETMINCONFIG(int M, int $curr$, int data)
2: $conf \leftarrow curr$
3: $w \leftarrow getLastWorkloads()$
▷ ***start scale-out evaluation***
4: **for** $i \leftarrow 0$ to $N - curr - 1$ **do**
5: $h \leftarrow T_{add}(data) \cdot N - curr - i$
6: $t_h \leftarrow t_{now} + h$
7: $\lambda_f(h) \leftarrow predict(t_h, w)$
8: $X_{max}(conf) \leftarrow estimate(conf)$
9: **while** $\lambda_f(h) > X_{max}(conf) \cdot \mathsf{p}$ **do**
10: $conf \leftarrow conf + 1$
11: **if** $conf == curr + M$ **then**
12: **return** $conf \leftarrow curr + M$
13: **else if** $conf == N - curr - i$ **then**
14: **return** $conf$
15: **else**
16: $X_{max}(conf) \leftarrow estimate(conf)$
17: $conf \leftarrow curr$
▷ ***start scale-in evaluation***
18: **for** $i \leftarrow 0$ to $curr - \alpha - 1$ **do**
19: $h \leftarrow T_{rem}(data) \cdot curr - \alpha - i$
20: $t_h \leftarrow t_{now} + h$
21: $\lambda_f(h) \leftarrow predict(t_h, w)$
22: $X_{max}(conf) \leftarrow estimate(conf)$
23: **while** $\lambda_f(h) < X_{max}(conf) \cdot \mathsf{p}$ **do**
24: $conf \leftarrow conf - 1$
25: **if** $conf == curr - M$ **then**
26: **return** $conf \leftarrow curr - M$
27: **else if** $conf == curr - \alpha - i$ **then**
28: **return** $conf$
29: **else**
30: $X_{max}(conf) \leftarrow estimate(conf)$
31: $conf \leftarrow curr$
32: **return** $curr$ ▷ ***no scaling action necessary***

Suppose that $curr + x$ (or $curr - x$) service instances are required to handle a predicted input rate, where $curr$ is the current number of provisioned service instances. It is therefore necessary to provision (or release) x service instances by means of a proper scaling action which takes a

time $T_{sa} \approx T_{add}(data) \cdot x$. In order to make $RP_{sa} \approx DP_{sa}$, our idea is to set the prediction horizon $h = T_{sa}$.

During the first phase (profiling), PASCAL profiles the durations of the scaling actions in order to pull out a *time table* which stores the amount of time necessary to add or remove a service instance (respectively $T_{add}(data)$ and $T_{rem}(data)$) according to the amount of *data* stored. PASCAL profiles also the maximum sustainable throughput $X_{max}(conf)$ for each configuration with the related response times to pull out a *performance table* containing for each configuration (i) the input rate, (ii) the throughput and (iii) the response time.

In the second phase (auto-scaling) the AutoScaler invokes continuously over time the function `getMinConfig()` reported in Algorithm 2. Each configuration assessment consists of two main parts: a scale-out evaluation followed by a scale-in evaluation.

In each assessment, knowing the period $T_{add}(data)$ needed to add a service instance, we are able to compute the duration of the entire scaling-action T_{sa} to add a certain number of service instances.

As mentioned before we want impose $h = T_{sa}$, thus we consider t_h as the future time instant up to which forecasting the workload $\lambda_f(h)$, i.e. the maximum forecasted workload during the next horizon h.

Then the Performance Estimator by relying on the performance table, is able to estimate the maximum throughput sustainable with the current configuration, so we decide whether a scale-out action is needed, otherwise we evaluate in the same way a scale-in action.

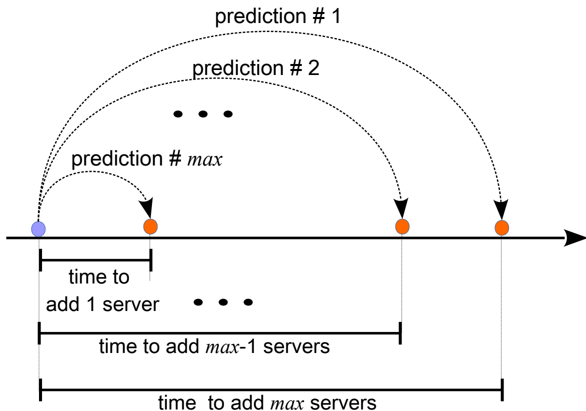


Figure 6: Representation of a scaling assessment step of for both scale-out and scale-in evaluation of the datastore solution Auto-Scaling algorithm.

Note that we introduced a parameter $\mathtt{p} \in (0, 1]$ referring to a max percentage of achievable maximum throughput. We call $X_B(conf) = X_{max}(conf) \cdot \mathtt{p}$ the *bounded throughput* of a configuration with *conf* service instances. The *p* parameter can be set by the user according to a desired level of performance; in our solution we set p according to a maximum desiderata response time (see Section 7.2 for more details.

Furthermore, introducing bounds on the maximum achievable throughput of each configuration allows the cluster nodes work under their maximum processing capability, so as to handle (within a certain extent) unexpected workload spikes whose intensity is beyond what can be predicted by the Workload Forecaster.

The configuration assessment period very is low so as to execute the algorithm very frequently and maximize its efficiency in founding the right point in time to trigger scaling actions. However, to avoid oscillations, after a scaling action the Decider execution is suspended until the ongoing reconfiguration process terminates.

We denote the minimum and maximum number of service instances that can be provisioned respectively as α and N [6], but note that a single scaling action is limited to $M \leq N$ for scalability reason, indeed without a limit, a cluster with hundreds of nodes should provide too far predictions. M can be set according to how far the user wants to over-provide: big values of M allow to handle workload with high variations at the cost of higher over-provisioning; on the contrary, low values of M allow more accuracy in predictions, .i.e., lower over-provisioning values at the cost of some possible under-provisioning if are not set enough new resources.

Figure 6 provides a graphical representation of a configuration assessment step of proposed auto-scaling algorithm; it is possible to see that the order of predictions is from the the farthest configuration (i.e. the one in which we need to add/remove M nodes) to the closest (i.e. the one in which we need to add/remove only 1 node) because the more servers are required to be added or removed in a single scaling action, the more time is required for the execution of such scaling action to complete. Therefore, it is required that adding or removing x servers is evaluated before the necessity of adding or removing $(x-1)$ servers, and so on. Specifically in each configuration assessment we first set the current configuration (*curr*) in a variable (*conf*), then we get the last observed input workload, stored in the metric DB, through the function `getLastWorkload()` (lines 2 and 3 of Algorithm 2).

Then we start evaluating whether a scale-out action is needed (lines 4-16), therefore iteratively from the farthest (i.e. adding maximum M nodes) to the closest scaling action (i.e. 1 node), we compute the horizon h by imposing the time needed to add i service instances, thus we find out the related time instant t_h as the current time t_{now} summed to the horizon h.

So, we forecast the maximum input load $\lambda_f(h)$ during the next horizon h through the `predict()` primitive exposed by the Workload Forecaster and we estimate the maximum throughput sustainable with such a configuration through the primitive `estimate()` provided by the Performance Estimator (lines 7-8).

[6]We cannot provide less service instance than the replication factor as well we cannot provide more instance than the cluster size having one service instance per node.

Then, we iteratively increase the number of service instances as long as we find out a configuration such that the estimation $X_{max}(conf) \cdot \mathtt{p}$ is lower than the forecasted input load $\lambda_f(h)$. If the new configuration needs M or more nodes the algorithm return $curr + M$ nodes, being M the maximum number of nodes foe a scaling action (line 11-12); if instead the new configuration requires exactly $N - curr - i$ nodes, it is returned as it is the minimum configuration to handle the expected workload in the future horizon h which require the horizon h to be effectively set (line 13-14). Otherwise no scale-out action is needed with the current horizon h.

If no scale-out action is necessary for any h is evaluated in a similar way if a scale-in action is necessary (lines 18-31). The main difference here is that we cannot neither scale-in more than M nodes nor scale under α, i.e. the replication factor, as it is the minimum number of nodes we can have in a configuration. Indeed, having less than α nodes implies that we cannot have each shard of data replicated among a sufficient number of nodes.

Finally, if neither a scale-out nor a scale-in action is necessary is returned the current configuration (line 32).

6.2. Implementing PASCAL in Cassandra

In this paper we considered Cassandra as an example of a typical distributed datastore [58]. Cassandra is a NoSQL column-oriented datastore. In Cassandra a column represents the smallest unit of storage composed by a unique name (key), a value and a time-stamp. Data stored in the keyspace is sharded among the nodes which make up the Cassandra cluster. Each node runs a Cassandra instance that, with reference to our system model, is the service instance. In order to provide fault-tolerance and high availability, Cassandra replicates records throughout the cluster by a user-set replication factor. Cassandra offers configurable consistency levels for each single operation, providing the flexibility of trading-off latency and consistency (*ZERO*, *ONE*, *QUORUM*, *ALL* or *ANY*). Cassandra clusters can be easily scaled horizontally achieving a linear increase of performance through the *nodetool* utility [58]. During a scaling action new nodes become active after transitioning through the *joining* (scale-out) and *decommissioning* (scale-in) states. At the end of this phase a *cleanup* operation must be invoked to clean up keyspaces and partition keys no longer belonging the nodes.

Service Monitor Implementation. The Service Monitor for Cassandra is made up of three sub-modules that read throughput and response times. To read CPU values, we implemented a JMX (Java Management Extension) client that gathers samples from Cassandra nodes. To compute the throughput we implemented a module that leverages the `cassandra.metrics.ClientRequest` MBean. Finally, to collect the response times we implemented a module that uses the DataStax Java Driver, to interact with Cassandra keeping track of latencies.

Service Profiler Implementation. The Service Profiler is implemented with java modules that collect through the Service Monitor the metrics of interest mentioned in the previous paragraph. The output performance model is composed by two tables mentioned in the previous subsection: the *performance table* and the *time table*. Those tables are serialized on a file that the Performance Estimator will use. From the time table the maximum values of adding and removing nodes are set as reference T_{add} and T_{rem} for the reference dataset of 1 GB used. Other $T_{add|rem}(data)$ for different *data* are estimated through regression on the time table.

AutoScaler Implementation. The Workload Forecaster is implemented as explained in Section 5.2. Before training the ANN, the trace output from the ILP needs to be filtered to smooth any rapid fluctuations; this is performed by extracting the 95th percentile values related to trace subsets spanning a time interval of one hour each. In such a way we obtained an *approximated workload trace* (AWT) that better represents the service usage pattern and such an AWT was then used as input training set for the ANN (more details discussed in Section 7.2.1).

The Performance Estimator and the Decider are Java modules that implement respectively the *estimate()* and *getMinConfig()* primitives. The Performance Estimator loads the serialized files containing the two tables output from the Service Profiler to make the estimation that the Decider, by implementing the algorithm 2, will use to compute the configuration.

The Configuration Manager is a java module that takes a scaling action $(scale_out|scale_in, x)$ and issues the `nodetool` command to start/stop x nodes. The nodes are activated/deactivated sequentially once the redistribution during the joining/decommision phase ends.

7. Experimental Evaluation

Testbed. We evaluated the effectiveness of both proposed PASCAL prototypes in a computing system composed by four IBM HS22 blade servers, each equipped with two quad-core Intel Xeon X5560 2.28 GHz CPUs, 24 GB of RAM running VMware ESXi v5.1.0 type-1 hypervisor. We deployed Storm and Cassandra nodes on dedicated VMs deployed on these blades. Load was generated through dedicated VMs and an external Dell PowerEdge T620 server equipped with a 8-core Intel Xeon CPU E5-2640 v2 2.00GHz, 32 GB of RAM running Linux Ubuntu 12.04 Server x86_64. Please note that the very same tests executed in public-cloud environments without resource reservation may possibly produce different results, as public-cloud compute instances may suffer from multiple users competing for limited shared resources, thus impairing the ability of PASCAL to correctly provision running applications.

Workload Trace. We evaluated PASCAL for the two case studies by using both synthetic and real traces to

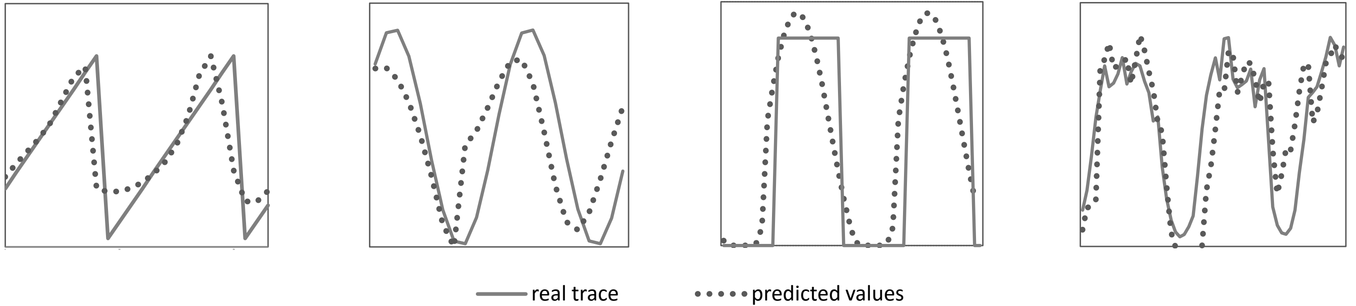


Figure 7: Accuracy of ANN prediction for the tested datasets.

generate the input load. As synthetic traces we employed (i) a square wave, (ii) a sine function and (iii) a sawtooth wave to mimic abrupt but predictable load changes. As a real trace we used a subset of a 10 GB Twitter trace with 3 months of tweets captured during the European Parliament election round of 2014 from March to May in Italy. To make tests with the real trace practical, we selected a subset ranging the 41 most intensive consecutive days having high load variations, and then applied a 60:1 time-compression factor to allow the replay of the real trace with reasonable timing. For the profiling phase we instead injected for 30 minutes a stair-shaped curve.

Artificial Neural Network Setup. To setup the Workload Forecaster ANN we followed common empirical rules presented in [47]. The final network is composed by 4 input nodes linked to time features (day, day of the week, month, hour) and 10 input nodes for the input rates measured during the last 10 scaling assessments. We empirically set a single hidden layer with 24 neurons and 5 output nodes representing the predicted workload intensity at the future time during the next forecast horizon h[7]. We trained the ANN with the Resilient Backpropagation [59] algorithm[8] and a k-cross validation with $k = 10$ to avoid overfitting. We normalized all data with min-max normalization $[0; 1]$. Figure 7 qualitatively shows the ability of the Workload Forecaster to predict both synthetic (sawtooth, sine and square synthetic loads in the three first plots) and real trace curves (twitter load in the last plot) providing an average Mean Square Error equal to 3%. Having a fine accuracy in load prediction is fundamental to accurately estimate system performance.

Performance Degradation Evaluation. To effectively assess PASCAL, we evaluate how performance metrics, i.e. throughput and response time, degrade over time, in order to keep as low as possible such performance degradations due to under-provisioning.

[7]The horizon h is discussed in Setion 7.

[8]We choice RPROP as it can set the weights of the neurons quickly and speed up the learning phase; besides, it is more robust against the choice of initial parameters, so it can relieve the user to the effort of accurately tuning the learning rate as he would need to do with the normal Backpropagation [59, 60].

CPU Max Threshold	Reward
0.60	0.53
0.65	0.57
0.70	0.62
0.75	0.66
0.80	**0.71**
0.85	0.65
0.90	0.43

Table 1: Reward of Q-Learning for CPU max_threshold

7.1. Evaluation of PASCAL on Storm

7.1.1. Environement

Storm Cluster. We distributed the Storm framework on a cluster of 5 VMs, each configured with 4 CPU cores, 4 GBs of RAM, and runnig Linux Ubuntu 14.04 Server x86_64. One was dedicated to hosting the Nimbus process and the Apache Derby DB, while the remaining 4 VMs hosted Storm's nodes.

Load Generation. One further VM was used for the Data Driver process, in charge of generating the input load. This VM was equipped with 2 CPU cores and 4 GBs of RAM. The Data Driver is Java process which generates tuples according to a given dataset, and sends them to a Java Messaging Service (JMS) queue. We setup such a queue with a HornetQ server [61]. The spouts are connected to such JMS queue to get tuples to be injected into the Storm's topology.

Reference Application. To evaluate PASCAL on Storm we implemented as a reference topology the Twitter *Rolling-Top-K-Words*. The topology is a chain of operators composed by a *TweetReader* spout that read tweets from the JMS queue and forwards them towards the subsequent operators i.e. a *WordGenerator* bolt, a *Filter* bolt, a *Counter* bolt and, finally, two *Ranker* bolts [62]. A further bolt, namely the *__acker*, allows to compute response time and throughput of every component. The reference topology is depicted in Figure 8, where is also possible to see the integration with the Data Driver process and the JMS queue.

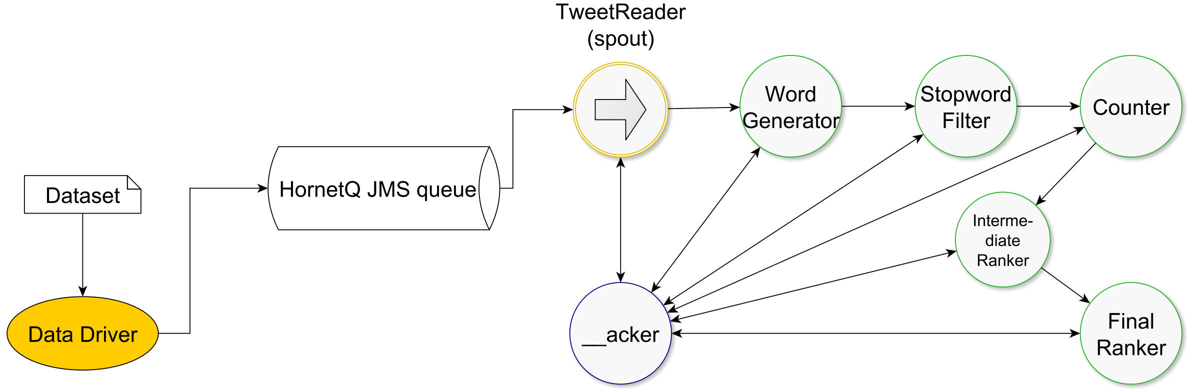


Figure 8: Twitter Rolling-Top-K-Words composed by 1 spout (TweetReader) and 6 bolts (WordGenerator, StopwordFilter, Counter, IntermediateRanker, FinalRanker, __acker). The topology is fed with tweets generated by the Data Driver process according to an input dataset. Tuples output from the Data Driver are generated so according a real workload curve and are enqueued in the JMS queue. Spout instances read those tuples and generate the stream towards the other bolts of the topology. Note that all Storm components, i.e. both spout and bolts, are parallelised and distributed among worker nodes. The number of such worker nodes change over time according to the PASCAL autoscaling algorithm.

Parameter Setup. The Forecaster maintains input rate samples in a sliding window spanning the last 10 minutes, a sample for each minute; it uses these values as ANN time-series input to compute the prediction of the next temporal horizon. We empirically set 5 output nodes representing the predicted workload intensity at the future time during the next 5 minutes; thus, the forecast horizon $h = 5$ as described in Section 5.1. To automatically set the *max_threshold*, we used a solution based on Reinforcement Learning. Specifically, we employed Q-Learning [63] during the profiling phase which starts without knowledge of the application behaviour. We propose a Reward Function $R(threshold)$ which aims to maximize the node usage, namely it looks for the maximum CPU threshold corresponding to the lowest throughput degradation, i.e., $R(cpu_max_thr) = cpu_max_thr - throughput_degradation$. The *throughput_degradation* is defined as $\frac{|input_load - throughput|}{input_load}$. Table 1 presents the rewards and is possible to see as the max reward corresponds to a threshold equals to 0.8, i.e. 80% CPU usage.

7.1.2. Load Imbalance Evaluation

This Section gives evidence of the possibility of load imbalance among worker nodes in a dSPS. Such a situation can derive from the way the scheduler allocates operators to workers nodes [9]. We carried out an evaluation in Storm by injecting a sinusoidal trace against 3 worker nodes where the reference application was deployed. As previously described, this topology has 4 operator instances for each operator, thus, for example, 2 worker nodes will be allocated one spout instance each, while the other worker node will host 2 spout instances, which is likely to lead to load imbalances.

[9]It is indeed possible to achieve load balancing by employing appropriate scheduling policies, but this is a different research problem not in the scope of this paper.

Figure 9: Comparison between the CPU load of two worker nodes hosting a different number of operator instances. Worker 1 has 1 spout instance, while Worker 2 has 2 spout instances. This leads to a load imbalance between the two worker nodes.

Figure 9 shows such a situation by comparing the load of the worker node #1 (with 1 spout instance) and the worker node #2 (which has instead 2 spout instances). It is possible to observe that in some periods the load of worker #2 is almost twice the load of the worker #1.

The next subsection shows how by aggregating the load per worker node with the OCUP and OP modules we are able to cope with such a load imbalance.

7.1.3. Performance Estimator Accuracy

The accuracy of the estimations provided by the Performance Estimator depends in turn on the accuracy of the profiles learned by the ILP, the SP, the OCUP, and the OP. Table 2 shows average and standard deviation of the selectivities observed for the streams of the reference topology, during the profiling phase. Reported standard deviations

Table 2: Average selectivity of reference topology edges with related standard deviation

Edge	Avg.	σ^2
WordGenerator - StopWordFilter	17.86	0.54
StopWordFilter - Counter	0.68	0.02
Counter - IntermediateRanker	0.41	0.34
IntermediateRanker - FinalRanker	0.01	0.00

are very small, which backups the implementation choice for the SP, described in Section 5, of modeling selectivities with constant values. The stream *Counter - IntermediateRanker* is the only one having a large standard deviation. This is due to the semantics of the *Counter* bolt; indeed, it periodically sends tuples downstream to the *IntermediateRanker* bolt, independently of its input rate. The impact on the estimation is negligible, because at runtime the input rate of the bolts downstream the Counter bolt is very low and produces limited CPU usage.

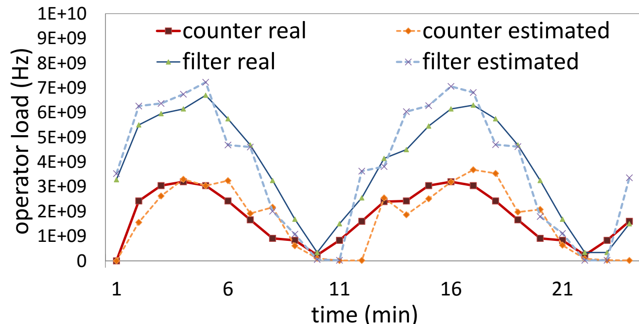


Figure 10: Comparison between real and estimated total CPU usage (in Hz) for all the instances of Counter and StopWordFilter operators

The accuracy of the OCUP is related to the estimations of CPU usage for an operator instance given its input rate. Such accuracy can be assessed by comparing these estimations against the real CPU usage. Figure 10 reports the real CPU usage over time of the instances of two operators, aggregated by operator, and the correspondent estimations provided by the OCUP. In this test, we injected a sinusoidal workload for 25 minutes. As the figure shows, the estimations faithfully reflect the actual load.

The OP allows to estimate the CPU usage of a node as a function of the sum of the CPU usage of all the operator instances sharing the same CPU. In this way it is possible to take into account the overhead due to the SPS, such as tuple dispatching and thread management. Figure 11 depicts the profiling of such overhead in a node of our cluster. Such profiles provide the remaining information needed to accurately predict the total CPU usage of a node.

7.1.4. Reconfiguration Overhead

The overhead introduced by PASCAL in Storm at runtime is negligible. Metrics about CPU usage and traffic are collected every 10 seconds, and their monitoring is extremely light. The bandwidth consumption for data collection is a few KBs, and depends on the number of operator instances, while it is independent from the input load. The real-time computation of the AutoScaler is extremely lightweight and consumes an insignificant amount of CPU periodically. When the configuration has to be changed, the throughput of an application degrades due to a rebalance operation which stops the topology, allocates the components on a different set of available nodes and re-starts the topology. Meanwhile, the input queue accumulates tuples and, as soon as the topology start again, the spouts have to drain such a queue. Reconfigurations have a non negligible cost, so we can set the maximum frequency they occur by tuning the assessment period. To objectively measure the reconfiguration overhead, we used the reference topology with an over-provisioned configuration injected by 9 minutes of sinusoidal input load. We computed the throughput degradation for different assessment periods. As expected, Figure 12 clearly shows the throughput degradation gets larger as the assessment period is decreased.

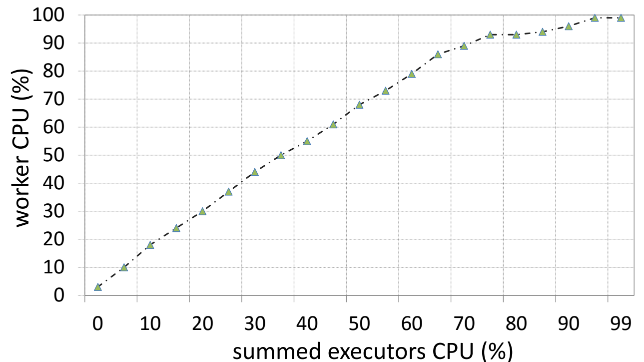


Figure 11: Node CPU usage as a function of the sum of the CPU usage of all the executors running in such node

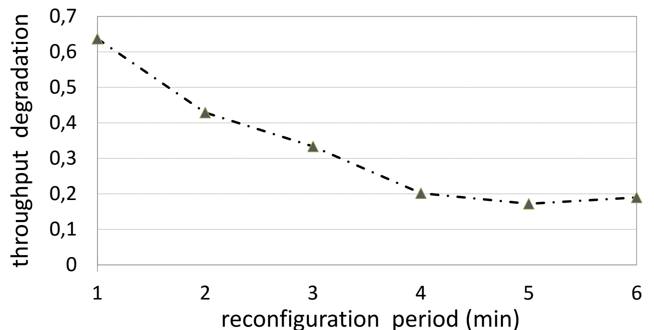
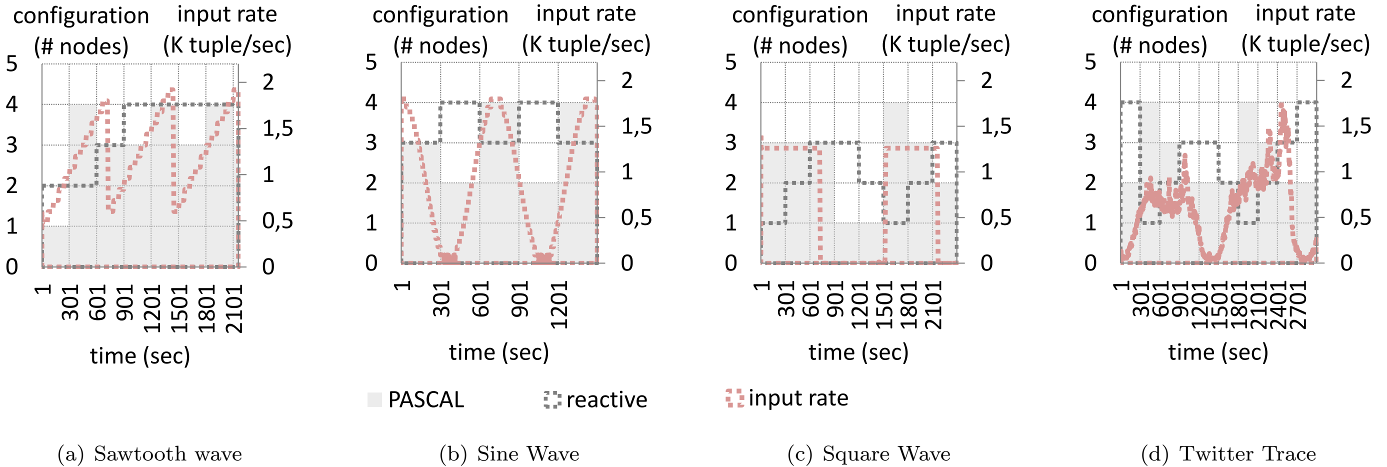


Figure 12: Throughput degradation as a function of the assessment period

By comparing these results with the quasi-zero throughput degradation obtained without reconfigurations and in an over-provisioned setting, it can be noted that re-



(a) Sawtooth wave (b) Sine Wave (c) Square Wave (d) Twitter Trace

Figure 13: Comparison between PASCAL and a reactive solution in term of used nodes while injected with different workload traces.

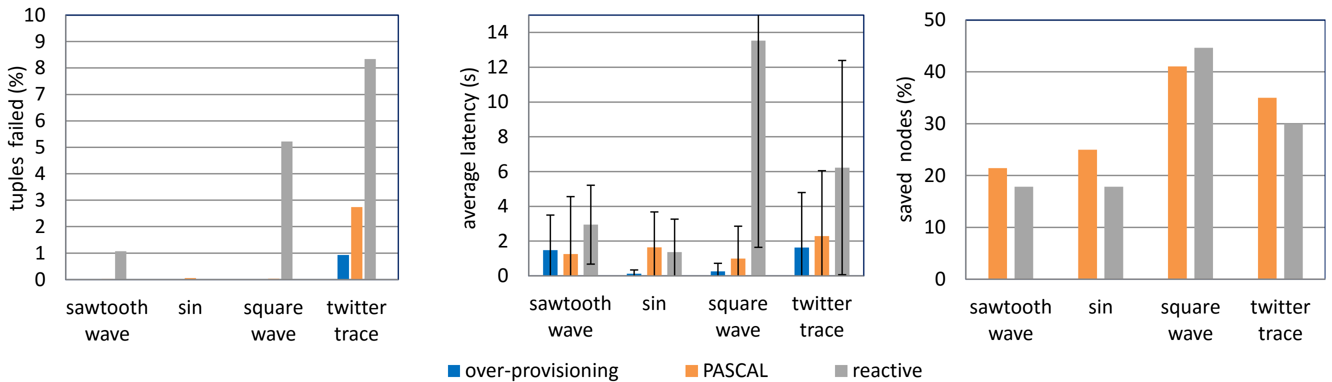


Figure 14: Auto-scaling test overall results for the different traces in term of tuples failed, average latency and saved nodes.

configuration overhead is significant. Nevertheless, reconfigurations are required when input load changes to some extent, otherwise application performance would become even worse. The assessment period has to be tuned according to input load variability and throughput degradation tolerance. In our tests, we set the assessment period to 5 minute i.e. where the throughput degradation settles.

7.1.5. Auto-scaling Test Results

We executed several tests on the four traces described in Section 7. We compared the performances of PASCAL with an over-provisioned configuration which is the minimum one able to sustain all injected workloads. In our experimentation such a configuration was composed by 4 nodes.

Figure 13(a) shows a workload generated with a sawtooth wave with a start rate of 500 event/sec that increases linearly each minute up to 1800 event/sec. On the x-axis we show the time in seconds, split in intervals of 300 seconds each, that correspond to our forecast horizon and assessment period; therefore, each tick marks a reconfiguration assessment. It is possible to note, between seconds 1-600, that PASCAL starts with a configuration of 1 node and quickly switches to a configuration with 4 nodes at the first assessment (second 300): this is caused by an input load increase forecasted for the next 5 minutes (up to second 600). PASCAL thus proactively scales-out the configuration before the system overloads.

The reactive auto-scaler, instead, keeps a configuration with 2 nodes because it is not noticing neither overloading nor underloading. At the second assessment (second 600) PASCAL forecasts the traffic in the next 5 minutes and notices that the input rate will decrease, so switches to a configuration with 3 nodes. Actually, the input rate remains high for other 2 minutes, and then decreases suddenly afterward. The reactive auto-scaler instead notices another overloading situation, so switches to a configuration with 3 nodes. In the next two assessments (seconds 900 and 1200), PASCAL keeps the configuration to 3 nodes and then switches to 4 nodes, while the reactive auto-scaler moves to a configuration with 4 nodes at once because it still notices overloading; this is due to the fact that, being the cluster overloaded, it accumulated events to handle from previous periods. So even if in the period 600-900 PASCAL and the reactive auto-scaler have the same configuration (3 nodes), PASCAL is able to handle all the workload, while the reactive auto-scaler does not, thus incurring some data losses. Overall results show how

PASCAL uses less resources than the reactive auto-scaler with 21.42% of saved nodes against 17.85% , providing a smaller and more stable average latency with 1259ms against 2944ms of the reactive approach and without tuples failure against a 1% of the reactive one. The over-provisioning approach has instead an average latency lower than the reactive auto-scaler and very similar to PASCAL, equals to 1478ms with a failure rate equals to zero.

Figure 13(b) shows the test on a sinusoidal wave. PASCAL and the reactive auto-scaler provide similar results in term of average latency with 1638ms for PASCAL against the 1370 of the reactive and in term of tuples failed as well both with quasi-zero percentage. The over-provisioning approach has a lower average latency with respect to both approaches equals to 121ms and no tuples failed as well. In term of saved nodes PASCAL saved more resources, with a 25% of nodes against the 17.85% of the reactive auto-scaler. This result can be easily explained as such a variable workload is well predicted by PASCAL that is able to save more nodes.

Figure 13(c) show the results on injected square wave. PASCAL, being able to predict the square wave spikes, prepares an adequate configuration in the previous assessment. The average latency of PASCAL is 955 ms against the 258 ms of the over-provisioning approach. Anyway PASCAL saved 41.07% resources in this test. The square wave instead resulted very difficult to handle for the reactive auto-scaler that is slow to adapt to sudden spikes. The results are: no tuples failed with PASCAL against 5.22% of the reactive auto-scaler; lower average latency for PASCAL with 995ms against 13535ms of the reactive. The reactive solution saved more nodes with 44.07% against the 41.07% of PASCAL, but this is because the reactive auto-scaler is definitely not able to follow such pattern and scaling decision is not taken properly.

Figure 13(d) shows a test on a real scenario based on the Twitter trace mentioned before. It is possible to see how PASCAL dynamically adapts the configuration to the predicted input workload. Only unpredictable spikes are a problem for PASCAL as is possible to see at seconds 2400. For instance, in the assessment at seconds 2400, PASCAL switches the configuration from 3 nodes to 2 nodes while a spike is occurring. PASCAL ignores that fail and keeps the configuration constant to 2 nodes. The results in the Twitter trace show how PASCAL has an average latency of 2284ms against the 1633ms of the over-provisioning. In term of saved nodes, PASCAL allowed to save 35% of nodes against 30% of the reactive.

The overall results presented in Figure 14 show how PASCAL has comparable and acceptable performance with respect to the over-provisioning approach in terms of latency allowing to save up to 41% of the available resources (nodes). Furthermore, all the tests showed a clear performance gain of PASCAL with respect to a reactive approach.

7.2. Evaluation of PASCAL on Cassandra

7.2.1. Environment

Cassandra Cluster. We evaluated the effectiveness of the proposed prototype using a cluster composed by 6 Cassandra 3.0 nodes, each one installed on a VM configured with 4 CPU cores, 4 GBs of RAM, running Linux Ubuntu 14.04 Server x86_64.

Load Generation. In order to properly assess the effectiveness of the PASCAL prototype, we developed a modified version of Apache JMeter [64], a workload generator able to generate massive requests against Cassandra. To make it generates a workload according to a real trace, we used the Throughput Shaping Timer (TST) plugin [65], which allows to control the rate at which client requests are generated by defining the desired shape of the workload to submit towards the target system.

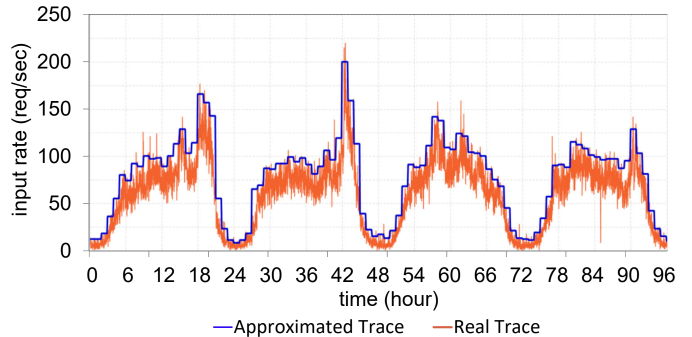


Figure 15: Real Trace and Approximated Workload Trace (AWT) of 4 consecutive days. This trace is fed into the TST plugin to make JMeter generate the workload according to the trace

We also developed a Data Driver process as a Java program which reads the input dataset and converts it to the AWT (described in Section 6.2) as an XML file containing tuples in a format compliant with the TST plugin. A *smooth* parameter allow to generate a more realistic smoother curve. Figure 15 shows the AWT generated from the real trace, while in Figure 16 is possible to see the workload plan fed towards JMeter; Specifically it shows how a part of AWT workload (figure on the left) is converted in a TST compliant workload plan that JMeter will generate. The JMeter generator is distributed over a cluster; one node is the coordinator (master) which leads the workload that each slave node have to generate against Cassandra according to the XML file within the TST plugin.

The workload consists of CQL queries requesting random keys. All the reads are executed with Consistency Level `ONE`. In such a way, we are evaluating cluster performances as if it is intended to be used by a Read-Intensive application (see Figure 17). To generate the CQL queries we developed a custom version of the CqlJmeter plugin [66], by editing the code to make it generate the workload to multiple Cassandra nodes in an asynchronous fashion to simulate a realistic multi-user workload [10].

[10]The original version of CqlJmeter plugin is synchronous, i.e., the

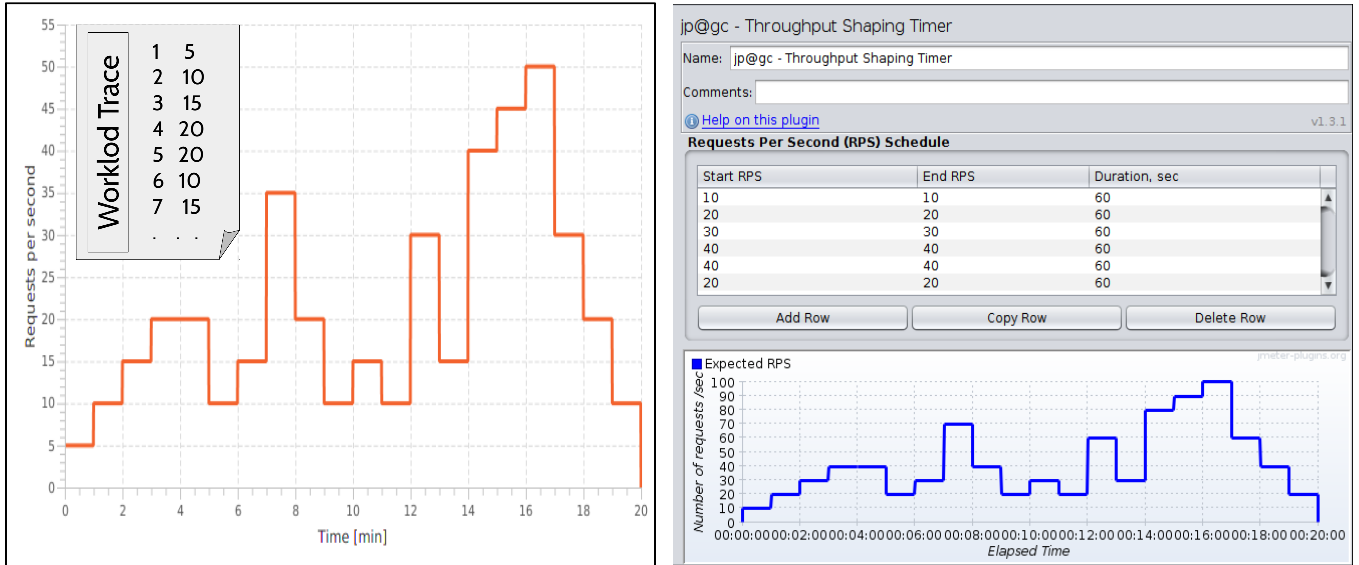


Figure 16: The left picture shows the Approximated Workload Trace (AWT) generated by the Data Driver. The picture on the right represents the related workload plan in Jmeter.

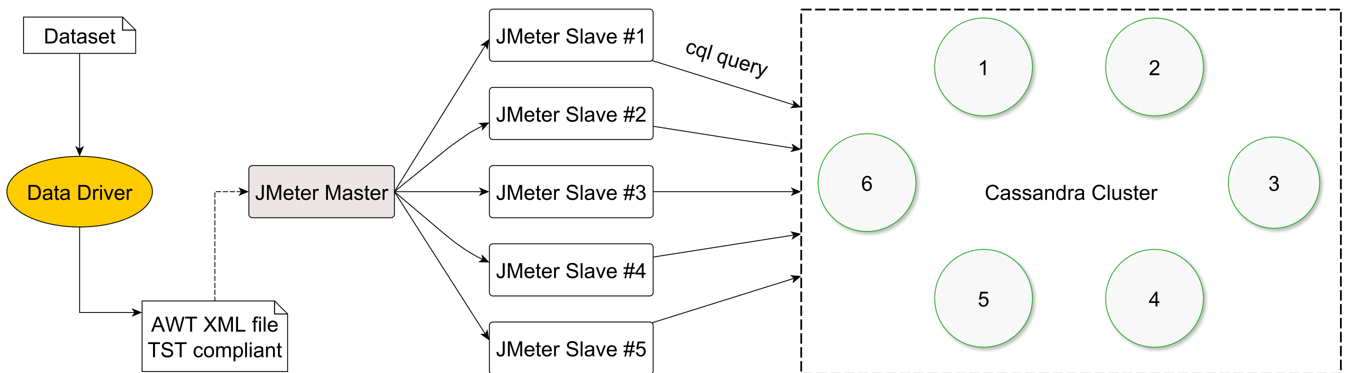


Figure 17: Integration of JMeter with the Cassandra cluster

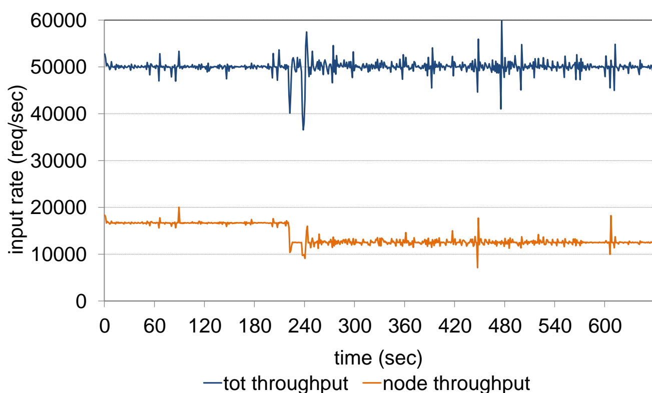


Figure 18: Joining impact on throughout when input rate is 50K rps

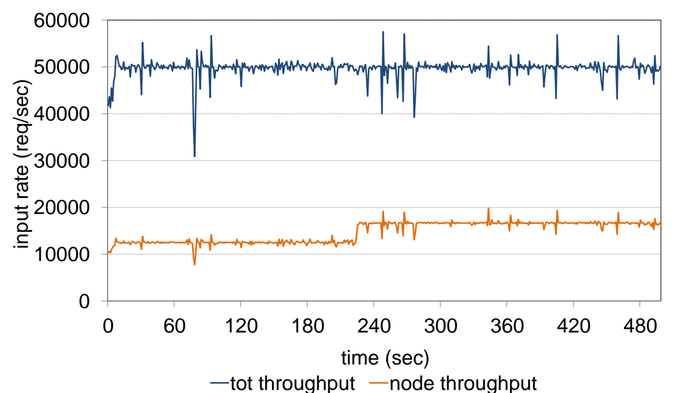


Figure 19: Decommission impact on throughout when input rate is 50K rps

workload generator client waits to receive a response before injecting the following request. This is because the plugin has been designed for benchmarking scopes and not for simulating realistic multi-user behaviour.

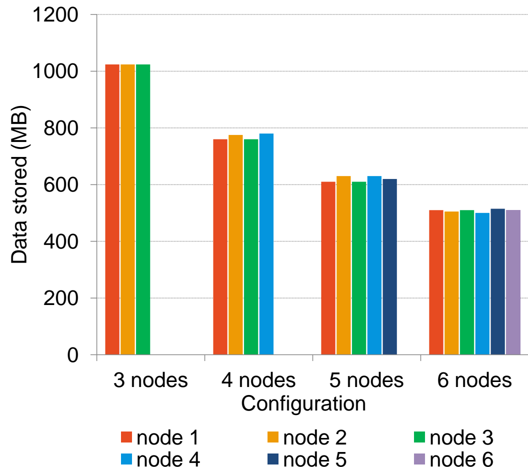


Figure 20: Distribution of 1.0 GB dataset stored with replication factor 3. By increasing nodes in the configuration, data are almost equally sharded among available nodes.

Reference Dataset. We created a Cassandra keyspace with replication factor $\alpha = 3$ [11] containing a column family with 11 columns; the first column is the primary key, while the other 10 columns contain random text strings of 20 characters for each field. The full dataset contained 1GB of data. Since we configured the keyspace with $\alpha = 3$, all experiments had a minimum number of nodes equals to 3 which have exactly the same data. By adding nodes, the 1GB of data will be sharded among the available nodes, and from Figure 20 is possible to see how the amount of data tends to fairly distribute over the nodes.

Parameters Setup. The Auto-Scaler prototype was configured with the following parameters indicating the minimum and maximum system configurations: $\alpha = 3$, $M = 6$, thus four possible system configurations were tested i.e. 3, 4, 5 and 6 nodes. In Figure 21 is represented the maximum sustainable throughput for each configurations with the related response time. These values represent the performance model output from the Service Profiler and we make use of it to properly set the bounded throughput. Specifically, the parameter p, i.e. the *max_threshold*, is set to 0.7. It means that each configuration is allowed to serve requests at a bounded throughput corresponding to the 70% of their maximum. Such value came out by the heuristic of limiting each configuration to a throughput such that average response times over this value start diverging with a high standard deviation. In our setting this response time is 4.5 ms. Maximum and bounded throughput with the obtained p are reported in Table 3.

[11] This is the common value to protect against data loss due to single machine failure once the data has been persisted to at least two machines in modern distributed datastore as recommended for example by DataStax for Cassandra [67], by Apache for HDFS [68] or Amazon for Dynamo [69].

Table 3: Maximum and bounded throughputs with 70% threshold

$conf$	$X_{max}(conf)$ **(tps)**	$X_B(conf)$ **(tps)**
3	92476	64733
4	105869	74108
5	118036	82625
6	131406	91984

Table 4: Add and Remove Times

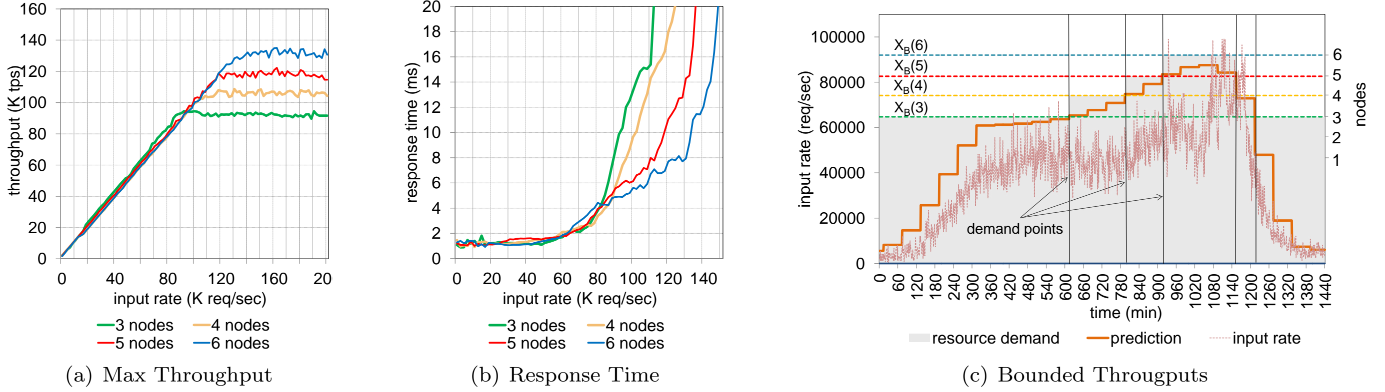
Config. Change	**Time (sec)**
3 to 4	155
4 to 5	155
5 to 6	155
4 to 3	120
5 to 4	145
6 to 5	160

7.2.2. Reconfiguration Overhead

We evaluated the time needed to switch from a configuration to another. Collected times are reported in Table 4, thus we obtain $T_{add} = 155$ sec and $T_{rem} = 160$ sec as maximum values. We then evaluated how the insertion/removal of a new node in the Cassandra cluster would impact the cluster performance, namely throughput, CPU utilization and response times. In particular we studied how the system behaves when, during the *joining* and *decommisioning* process, a constant workload with input rate of 50.000 requests per seconds is submitted to the cluster.

Figure 18 shows the impact of the joining process. We set the cluster starting with 3 nodes and a fourth node is inserted at time 120 seconds. Before and during the joining process, each node of the cluster serves one third of the received requests. After the joining process is completed (second 210), the new node starts serving requests and as result all nodes of the cluster now serve one fourth of the received requests. The cleanup operations are triggered 60 seconds after the time instant in which the fourth node has completed its joining process and does not impact on the cluster throughput.

Figure 19 presents the impact of the decommission process. We set a cluster starting with 4 nodes and the removal of a node is triggered at time 120 seconds. Before and during the decommission process, each node of the four, serves one fourth of received requests. As soon as the decommissioning process is completed, the removed node stops serving user requests, and its portion of handled requests is fairly redistributed among the remaining three nodes, each one serving one third of requests without any impact on the total throughput.



(a) Max Throughput (b) Response Time (c) Bounded Througputs

Figure 21: Performance Model obtained by profiling throughput and response times with different configurations under different input rates. Figure 21(a) shows the max throughput from each configuration. From Figure 21(b) it is possible to see how the response times diverge over the max sustainable input rate. We set a threshold to those throughput levels and we used the $X_B(conf)$ (reported in Table 3) to decide the minimum configuration. Finally, Figure 21(c) shows how we actually decide the configuration from the forecasted workload through the performance model.

7.2.3. Test Results

As we previously discussed, in Figure 15, it is possible to see the generated AWT compared to the real input workload over 4 consecutive days which is fed to the ANN of the Workload Forecaster to learn it. As we evaluated in the previous sections, due to a combination of time-series and date features used as input nodes for the ANN, it is possible to forecast with high accuracy different days. Figure 21(c) reports, in more detail, the real input rate injected into Cassandra with the related workload prediction of a single day. Through the prediction and the bound throughput values, we obtain the resource demand to figure out which is the minimum configuration able to handle the predicted workload. The bounded throughput values are obtained from the performance model by applying the heuristic aforementioned on the maximum throughput and the response time for each configuration (see respectively Figure 21(a) and 21(b)).

To evaluate PASCAL, we compared the performance of our system with an over-provisioning solution and another based on average load. In Figure 22 we show how the throughput sustained from PASCAL is the same of the over-provisioning solution (with static 6 nodes), except for a very short period of system reconfiguration appreciable mostly at minute 780 and 910.

Comparing the performance in term of response times is instead possible to see how PASCAL follows the same behaviour of the over-provisioning solution, while the solution based on the average load (with 3 nodes) after the minute 600, starts increasing its response times. These results are presented in Figure 23.

It is also possible to note between the minutes 390 and 525 as well as over the minute 1173, that PASCAL has a lower response time with respect to the over-provisioning solution. This result makes sense because PASCAL, by using less nodes with respect the over-provisioning solution, allows to have a lower overhead due to less nodes

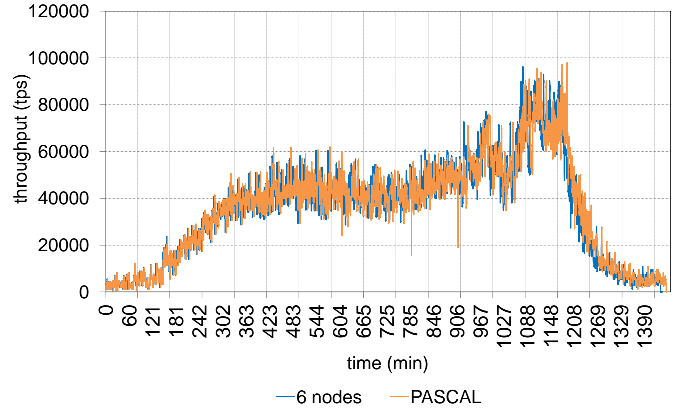


Figure 22: Throughput comparison between 6-nodes config and PASCAL

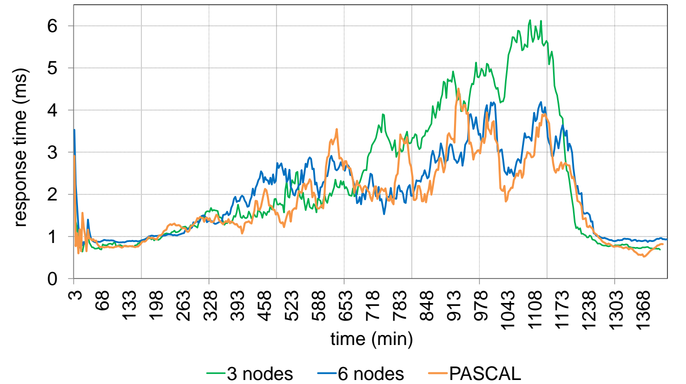


Figure 23: Response time comparison between static configs and PASCAL

inter-communication. Till minute 600 PASCAL provides a configuration with 3 nodes; having a replication factor $\alpha = 3$, when a client request a data to a random node, it always finds a node which has the requested data and it is able to send to the client directly the response. When

scale id	action	TP	RP	DP	duration	Δ
1	+1	7242 s (603 min)	7347 s (612 min)	7380 s (615 min)	105 s	33 s
2	+1	9399 s (783 min)	9492 s (791 min)	9540 s (795 min)	93 s	48 s
3	+1	10848 s (903 min)	10935 s (911 min)	10980 s (915 min)	87 s	45 s
4	-2	13602 s (1134 min)	13833 s (1152 min)	13860 s (1155 min)	231 s	27 s
5	-1	14379 s (1198 min)	14514 s (1210 min)	14580 s (1215 min)	135 s	66 s

Table 5: Scaling Event Times

instead a client request a data to a random node in an over-provisioning configuration having 6 nodes, with certain probability it will not find the data in that node and the latter has to forward the request toward a node which has the requested data.

From Table 5 is possible to see how every scaling action is triggered (TP) so as to have the RP some times before the DP. In the same table are shown the duration of the scaling action and a value δ representing how many seconds before the DP the configuration is ready (i.e. RP). As a result we can see how each scaling action concludes 30-60 seconds before the demand point so as to avoid an excessive period of over-provisioning, and providing an adequate configuration offering good performances as aforementioned by Figure 22 and 23.

8. Conclusions

In this paper we introduced PASCAL, a modular architecture for auto-scaling generic distributed services. The main contributions of the paper regard (i) the definition of general architectural framework for enabling proactive automatic scaling of distributed services, (ii) a solution for proactive auto-scaling a distributed stream processing system and (iii) a solution for proactive auto-scaling a distributed datastore. The problem we addressed in implementing the architectural framework for the two diverse case studies has been respectively (i) a solution to estimate performance of worker nodes even with servers having unbalanced load and (ii) a solution to choose the triggering point of a scaling action in order to reduce as much as possible the difference between the demand point and the reconfiguration point. The modular architecture of PASCAL allows to move from scenario to scenario simply changing the implementation of some modules. In this work we implemented two real prototypes of PASCAL, one dedicated to Apache Storm and one dedicated to Cassandra.

The experimental results show as in both cases PASCAL allows to save up to 40% of resources while handling a variable workload with low latency and performance comparable to the over-provisioning configuration. In the dSPS scenario we showed how PASCAL outperforms a reactive approach. In the datastore case study we instead showed as the system reconfiguration anticipates some seconds the demand point, reducing so the over-provisioning. Furthermore, by employing only resources actually necessary, the instantiation of PASCAL in specific periods performs better than the over-provisioning configuration as a lower overhead due to a less message exchange between nodes allows to reduce the response time while ensuring the same throughput.

9. Future Directions

To extend this work we aim to consider three main directions: (i) a more comprehensive model for autoscaling a target system, (ii) an integration with solutions to provide fault tolerance and finally (iii) an autoscaling benchmark able to compare elasticity and effectiveness provided by different systems. A detailed description of these three directions follows.

Comprehensive Model
As future directions, first we aim to design a more complete model to estimate performances which includes also memory and bandwidth. Furthermore, we aim to extend the model to heterogeneous servers having different computational capability, by building upon the proactive approach based on Q-learning we proposed in [70]. We also want to investigate solutions to integrate load shedding techniques to tackle with load bottlenecks.

Fault Tolerance
Then, we aim to extend the work by integrating scaling and fault tolerance, thus, the scaling policy will be not only related to system overloading, but also in response to a failure. Specifically, we are striving to integrate the failure prediction and anomaly detection system we proposed in [71, 72] as a further module of PASCAL interacting with the Decider module to trigger scaling decisions.

Furthermore, to make the system tolerant to Byzantine attacks, we target to decentralized the architecture in order to provide an elastic Byzantine Fault Tolerant approach similar to CloudBFT [73]. For the stream processing scenario a relevant work to compare with shall be [33], as the authors integrate fault tolerance with operators scale-out, but without proposing a real autoscaling solution as they do not scale-in. Another relevant work to compare with is Drizzle [74] a reactive solution which integrates fault tolerance and scaling. For the datastore

scenario, instead, a good system to compare with can be Replex [75], a scalable datastore which efficiently react to failure, but not providing an autoscaling feature.

Autoscaling Benchmark
As a final direction, we aim to develop a benchmark to evaluate autoscaling systems by standardising, for example, models, metrics and load generation. A good work to start with is APMT (Autoscaling Performance Measurement Tool) [76], a preliminary tool proposed by Jindal et al. to evaluate autoscaling systems. Then, to develop a comprehensive benchmark, we aim to (i) implement the definition suggested by Kuperberg et al. [77] to quantify and compare elasticity, and (ii) integrate novel solutions to detect scalability issues and bottlenecks as Wang et al. proposed with Exalt and Tardis [78] and with semi-automatic tools like PatternMiner [79] by extrapolating workloads. An integration with Exalt/Tardis and PatternMiner may successfully help the autoscaling developer to assess system scalability; indeed, those solutions allow to estimate with good accuracy the scalability of a system having thousands of nodes with less than 10 physical nodes.

References

[1] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[2] A. Ishii and T. Suzumura, "Elastic stream computing with clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 195–202.

[3] L. Aniello, S. Bonomi, F. Lombardi, A. Zelli, and R. Baldoni, "An architecture for automatic scaling of replicated services," in *Networked Systems*. Springer, 2014, pp. 122–137.

[4] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 871–879, Jun. 2011.

[5] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal autoscaling in a iaas cloud," in *Proceedings of the 9th International Conference on Autonomic Computing*, ser. ICAC '12. New York, NY, USA: ACM, 2012, pp. 173–178.

[6] L. R. Moore, K. Bean, and T. Ellahi, "Transforming reactive auto-scaling into proactive auto-scaling," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, ser. CloudDP '13. New York, NY, USA: ACM, 2013, pp. 7–12.

[7] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *11th International Conference on Autonomic Computing (ICAC 14)*, 2014, pp. 57–64.

[8] R. da Rosa Righi, V. F. Rodrigues, C. A. da Costa, G. Galante, L. C. E. de Bona, and T. Ferreto, "Autoelastic: Automatic resource elasticity for high performance applications in the cloud," *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 6–19, 2016.

[9] V. Setty, R. Vitenberg, G. Kreitz, G. Urdaneta, and M. Van Steen, "Cost-effective resource allocation for deploying pub/sub on cloud," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 555–566.

[10] M. R. Jam, L. M. Khanli, M. K. Akbari, E. Hormozi, and M. S. Javan, "Survey on improved autoscaling in hadoop into cloud environments," in *Information and Knowledge Technology (IKT), 2013 5th Conference on*. IEEE, 2013, pp. 19–23.

[11] S. Chaisiri, R. Kaewpuang, B.-S. Lee, and D. Niyato, "Cost minimization for provisioning virtual servers in amazon elastic compute cloud," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2011, pp. 85–95.

[12] F. P. Tso, K. Oikonomou, E. Kavvadia, and D. P. Pezaros, "Scalable traffic-aware virtual machine management for cloud data centers," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 238–247.

[13] I. Georgievski, V. Degeler, G. A. Pagani, T. A. Nguyen, A. Lazovik, and M. Aiello, "Optimizing energy costs for offices connected to the smart grid," *IEEE Transactions on Smart Grid*, vol. 3, no. 4, pp. 2273–2285, 2012.

[14] R. Han, L. Guo, M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, 2012, pp. 644–651.

[15] M. Hasan, E. Magana, A. Clemm, L. Tucker, and S. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012, pp. 1327–1334.

[16] M. Maurer, I. Brandic, and R. Sakellariou, "Enacting slas in clouds using rules," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 455–466.

[17] T. N. T. Blog, "Scryer: Netflix's predictive auto scaling engine," http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html, November 2013.

[18] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *Network and Service Management (CNSM), 2010 International Conference on*, 2010, pp. 9–16.

[19] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 155–162, Jan. 2012.

[20] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011, pp. 500–507.

[21] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:14.

[22] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[23] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow," in *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, Venice/Mestre, Italy, 2011, pp. 67–74.

[24] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "Vconf: A reinforcement learning approach to virtual machines auto-configuration," in *Proceedings of the 6th International Conference on Autonomic Computing*, ser. ICAC '09. New York, NY, USA: ACM, 2009, pp. 137–146.

[25] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1:1–1:39, Mar. 2008.

[26] Q. Zhang, L. Cherkasova, and E. Smirni, "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications," in *Proceedings of the Fourth International Conference on Autonomic Computing*, ser. ICAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 27–.

[27] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and

D. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009.

[28] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012, pp. 204–212.

[29] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 13–26.

[30] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 296–302.

[31] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 13–22.

[32] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 276–287.

[33] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM, 2013, pp. 725–736.

[34] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 572–585, 2018.

[35] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 171 – 185, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17326821

[36] G. Mencagli, M. Torquati, and M. Danelutto, "Elastic-ppq: A two-level autonomic system for spatial preference query processing over dynamic data streams," *Future Generation Computer Systems*, vol. 79, pp. 862 – 877, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X1730938X

[37] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *Journal of Systems and Software*, vol. 127, pp. 205 – 216, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216300796

[38] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buyya, "A stepwise auto-profiling method for performance optimization of streaming applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 12, no. 4, pp. 24:1–24:33, Nov. 2017. [Online]. Available: http://doi.acm.org/10.1145/3132618

[39] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, April 2016, pp. 22–31.

[40] C. Hochreiner, M. Vogler, S. Schulte, and S. Dustdar, "Elastic stream processing for the internet of things," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 100–107.

[41] A. Al-Shishtawy and V. Vlassov, "Elastman: elasticity manager for elastic key-value stores in the cloud," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013, p. 7.

[42] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 69–82.

[43] S. Barker, Y. Chi, H. Hacigümüs, P. Shenoy, and E. Cecchet, "Shuttledb: Database-aware elasticity in the cloud," in *11th International Conference on Autonomic Computing (ICAC 14)*, 2014, pp. 33–43.

[44] C.-W. Huang, W.-H. Hu, C. C. Shih, B.-T. Lin, and C.-W. Cheng, "The improvement of auto-scaling mechanism for distributed database-a case study for mongodb." in *APNOMS*, 2013, pp. 1–3.

[45] E. Casalicchio, L. Lundberg, and S. Shirinbab, "Energy-aware auto-scaling algorithms for Cassandra virtual data centers," *Cluster Computing*, vol. 20, no. 3, pp. 2065–2082, 2017.

[46] M. Kuperberg, N. Herbst, J. von Kistowski, and R. Reussner, "Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms," https://sdqweb.ipd.kit.edu/publications/pdfs/KuHeKiRe2011-ResourceElasticity.pdf, KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association, Tech. Rep., 2011.

[47] G. Zhang, B. E. Patuwo, and M. Y. Hu, "Forecasting With Artificial Neural Networks: the State of the Art," *International Journal of Forecasting*, vol. 14, no. 1, pp. 35 – 62, 1998.

[48] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-based data stream processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 238–245.

[49] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 46, 2014.

[50] S. Schneider, M. Hirzel, and B. Gedik, "Tutorial: stream processing optimizations," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 249–258.

[51] N. Tatbul and S. Zdonik, "Dealing with overload in distributed stream processing systems," in *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*. IEEE, 2006, pp. 24–24.

[52] Y. Zhai and W. Xu, "Efficient bottleneck detection in stream process system using fuzzy logic model," in *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*. IEEE, 2017, pp. 438–445.

[53] A. S. Foundation, "Storm," http://storm.apache.org.

[54] ——, "Storm metrics framework." [Online]. Available: http://storm.apache.org/documentation/Metrics.html

[55] A. S. Fundation, "Apache derby db." [Online]. Available: https://db.apache.org/derby/

[56] H. Research, "Encog machine learning framework." [Online]. Available: http://www.heatonresearch.com/encog/

[57] DataStax, "Cassandra docs - operations - adding nodes to an existing cluster." [Online]. Available: https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_add_node_to_cluster_t.html

[58] A. S. Foundation, "Cassandra." [Online]. Available: http://cassandra.apache.org/

[59] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The rprop algorithm," in *Neural Networks, 1993., IEEE International Conference on*. IEEE, 1993, pp. 586–591.

[60] N. G. Pavlidis, D. K. Tasoulis, M. N. Vrahatis *et al.*, "Time series forecasting methodology for multiple-step-ahead prediction." *Computational Intelligence*, vol. 5, pp. 456–461, 2005.

[61] JBoss, "Hornetq." [Online]. Available: http://hornetq.jboss.org/

[62] Twitter, "Twitter-rolling-top-words topology." [Online]. Available: https://storm.apache.org/javadoc/apidocs/storm/starter/RollingTopWords.html

[63] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[64] A. S. Fundation, "Jmeter," 2016. [Online]. Available: http://jmeter.apache.org/

[65] ——, "Jmeter throughput shaping timer plugin," 2016. [Online]. Available: http://jmeter-plugins.org/wiki/

ThroughputShapingTimer/
[66] M. Stepura, "Cqljmeter plugin." [Online]. Available: http://mishail.github.io/CqlJmeter
[67] DataStax, "Best practices in deploying and managing datastax enterprise," 2014. [Online]. Available: https://www.datastax.com/wp-content/uploads/2014/04/WP-DataStax-Enterprise-Best-Practices.pdf
[68] A. S. Foundation, "Hdfs architecture guide," 2014. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[69] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.
[70] F. Lombardi, "A proactive q-learning approach for autoscaling heterogeneous cloud servers," in *2018 14th European Dependable Computing Conference (EDCC)*. IEEE, 2018, pp. 166–172.
[71] R. Baldoni, A. Cerocchi, C. Ciccotelli, A. Donno, F. Lombardi, and L. Montanari, "Towards a non-intrusive recognition of anomalous system behavior in data centers," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2014, pp. 350–359.
[72] C. Ciccotelli, L. Aniello, F. Lombardi, L. Montanari, L. Querzoni, and R. Baldoni, "Nirvana: A non-intrusive black-box monitoring framework for rack-level fault detection," in *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*. IEEE, 2015, pp. 11–20.
[73] R. Nogueira, F. Araujo, and R. Barbosa, "Cloudbft: elastic byzantine fault tolerance," in *Dependable Computing (PRDC), 2014 IEEE 20th Pacific Rim International Symposium on*. IEEE, 2014, pp. 180–189.
[74] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 374–389.
[75] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi, "Replex: A scalable, highly available multi-index data store." in *USENIX Annual Technical Conference*, 2016, pp. 337–350.
[76] A. Jindal, V. Podolskiy, and M. Gerndt, "Autoscaling performance measurement tool," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 91–92.
[77] M. Kuperberg, N. Herbst, J. Von Kistowski, and R. Reussner, "Defining and quantifying elasticity of resources in cloud computing and scalable platforms," 2011.
[78] Y. Wang, M. Kapritsos, L. Schmidt, L. Alvisi, and M. Dahlin, "Exalt: Empowering researchers to evaluate large-scale storage systems." in *NSDI*, 2014, pp. 129–141.
[79] R. Shi, Y. Gan, and Y. Wang, "Evaluating scalability bottlenecks by workload extrapolation," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 333–347.

Federico Lombardi is a Lecturer in cybersecurity at University of Southampton. He obtained a PhD in Engineering in Computer Science from Sapienza University of Rome and his research mainly copes with scalability and elasticity problems of cloud services, distributed storage, stream processing systems and blockchain. He also works on performance and security evaluations for failure prediction and detection systems, IoT and consensus algorithms. He has been involved in several National and EU projects related to cybersecurity, blockchain and IoT.

Andrea Muti is a MsC In Engineering of Computer Science from Sapienza University of Rome. Currently he works as Software Engineer at Capgemini. His work mainly copes with the integration of IoT devices and Big Data technologies. Specifically he focuses on Stream Processing Systems, Complex Event Processing and scalability of distributed datastore for reliable and multi-tenant solutions.

Leonardo Aniello is a Lecturer at University of Southampton. He obtained a Ph.D. in Engineering in Computer Science from the Sapienza University of Rome. His research studies include several topics in the fields of Big Data in large-scale environments, distributed storages and distributed computation techniques, with focus on the aspects of cyber security, integrity (blockchain-based storage), fault-tolerance, scalability and performance. Leonardo is author of more than 20 papers about these topics, published on international conferences, workshops, journals and books.

Roberto Baldoni Roberto Baldoni is a full professor in the field of Distributed Systems at Sapienza University of Rome. He has been the coordinator of several EU projects, the vice-Chair of the IEEE technical Committee on Fault Tolerant and Dependable Systems and IEEE TDFT Committee Chair. He is a member of the IFIP WG 10.4 and of the steer-

ing committees of ACM DEBS, DSN Conferences and of the Editorial Board of IEEE Transactions on Parallel and Distributed Systems. He is author of around 200 peer-reviewed publications in international scientific conferences and journals in the area.

Silvia Bonomi is a PhD in Computer Science at Sapienza University of Rome. She is member of the Research Center of Cyber Intelligence and Information Security (CIS) in Sapienza. She is doing research on various computer science fields including Byzantine fault-tolerance, dynamic distributed systems, Intrusion Detection Systems and event-based systems. In these research fields, she published several papers in peer reviewed scientific forums. She has been involved in several National and EU-funded project where she addressed problems related to dependability and security of complex distributed systems like smart environment or critical infrastructures.

Leonardo Querzoni is assistant professor at Sapienza University of Rome. His research interests range from distributed systems to computer security and focus, in particular, on topics that include distributed stream processing, dependability and security in distributed systems, large scale and dynamic distributed systems, publish/subscribe middleware services. He regularly serves in the technical program committees of conferences in the field of dependability and event-based systems like DSN and ACM DEBS. He was general chair for the 2014 edition of the OPODIS conference and has been appointed as PhD Symposium co-chair for the 2017 edition of the ACM DEBS conference.