Adaptive Fault-Tolerance for Dynamic Resource Provisioning in Distributed Stream Processing Systems

Paolo Bellavista
Università di Bologna,
Department of Computer Science and Engineering,
Bologna, Italy
paolo.bellavista@unibo.it

Antonio Corradi
Università di Bologna,
Department of Computer Science and Engineering,
Bologna, Italy
antonio.corradi@unibo.it

Spyros Kotoulas
Smarter Cities Technology Centre,
IBM Research,
Dublin, Ireland
spyros.kotoulas@ie.ibm.com

Andrea Reale*
Università di Bologna
Department of Computer Science and Engineering
Bologna, Italy
andrea.reale@unibo.it

ABSTRACT

A growing number of applications require continuous processing of high-throughput data streams, e.g., financial analysis, network traffic monitoring, or Big Data analytics for smart cities. Stream processing applications typically require specific quality-of-service levels to achieve their goals; yet, due to the high time-variability of stream characteristics, it is often inefficient to statically allocate the resources needed to guarantee application Service Level Agreements (SLAs). In this paper, we present LAAR, a novel method for adaptive replication that trades fault tolerance for increased capacity during load spikes. We have implemented and validated LAAR as a middleware layer on top of IBM InfoSphere Streams®. We have performed a wide set of experiments on an industrial-quality 60-core cluster deployment and we show that, under the assumption of only statistical knowledge of streams load distribution, LAAR can reduce resource consumption while guaranteeing an upper-bound on information loss in case of failures.

Keywords

data streams processing, fault-tolerance, dynamic adaptation, service-level agreement, IBM InfoSphere Streams®

1. INTRODUCTION

In recent years, the ability to effectively process Big Data Streams is becoming increasingly important: the vision of smarter cities where data from several physical-world sources are continuously collected, filtered, analyzed, and fed back to administrators and citizens to assist them in their hour-by-hour tasks is just one example of the multitude of novel scenarios where handling large and unbounded flows of information in real-time is a primary requirement.

Experience with Cloud services [31] has shown that the possibility to offload the management of computing infrastructures to third parties represents an attractive opportunity for both developers and cloud providers. However, in a cloud environment, the nature of stream processing applications poses hard challenges to platform providers, including the ability to offer, at the same time, extreme performance *elasticity* in spite of load variations and *resiliency* to failures, while keeping *costs* limited.

From a provider perspective, one major problem lies in the necessity to handle load fluctuations due to sudden and possibly temporary variations in the rates of data streams feeding the hosted applications. If not handled properly, in fact, load peaks can lead to increased processing latency due to data queuing and to data loss due to queue overflows. To avoid these effects, it is necessary to allocate the proper amount of additional resources for the overloaded applications, either statically or dynamically when load variations are detected [4, 8, 22].

Another typical requirement for stream processing applications is the implementation of *fault-tolerance* techniques. In fact, since they usually run for (indefinitely) long time intervals, failures are unavoidable. Many proposals in the literature have investigated possible fault-tolerance approaches — including active replication [9, 28], checkpointing [11, 18], replay logs [6, 16], or hybrid solutions [34] — each providing different trade-offs between runtime cost in absence of failures (*best-case*) and *recovery* cost. Whichever the adopted technique, maintaining some form of replication at some level (software/hardware components, state, or messages) is a significant overhead in terms of computing resources.

In a large class of applications, however, "perfect" fault tolerance is not always required, while it is of primary importance to effectively manage temporary load variations. This is very common, for example, when dealing with Smart City-generated Big Data. In this context, in fact, large data streams are produced by many distributed sources —

*Part of this work has been developed while the author was an intern at IBM Research Dublin.

10.5441/002/edbt.2014.09

e.g., mobile phones, ad-hoc sensing devices, or vehicles — that continuously capture and transmit sensed environmental features. These data need to be analyzed in real-time, and results must be promptly delivered to let appropriate control actions be performed. In this kind of scenarios, controlled information loss is usually tolerable, given the common partial information redundancy or overlap of input streams[1]. Consider, for instance, an application used to control traffic light signals based on periodic reports of vehicles' positions, among other factors. During high traffic conditions (i.e., high system load), it is clearly preferable to compute on incomplete information than delay control decisions, given the high redundancy in reported positions. At the same time, during low traffic conditions, processing events with accuracy is still important.

In this work, we investigate the possibility to trade-off reliability guarantees and execution cost, and use the conserved resources to handle load variations. We propose a novel method, called Load-Adaptive Active Replication (LAAR), that dynamically deactivates and activates redundant replicas of application Processing Elements (PE) in order to claim/release resources and accommodate temporary load variations. Our technique provides a-priori guarantees about the achievable levels of fault-tolerance, expressed in terms of an *internal completeness* metric that captures the maximum amount of information that can be lost in case of failures. We show that LAAR can be suitably implemented as a middleware-level layer on top of existing stream processing platforms, and we present general architectural and design guidelines about how to do it efficiently. As a working proof-of-concept, we describe an implementation of LAAR on top of IBM InfoSphere Streams® [13], an enterprise-level stream processing platform, and we discuss experimental results about the performance of LAAR on a 60-core IBM BladeCenter® cluster deployment.

The remainder of the paper is organized as follows: after reviewing the related literature in Section 2, we present the considered SLA-aware stream processing service model in Section 3. In Section 4, we model our middleware and explain its goals and runtime architecture. Finally, in Section 5, we report a wide set of performance results that quantitatively evaluate the effectiveness of our proposal.

2. RELATED WORK

Techniques for managing load variations have been extensively investigated in the literature, and relevant results have been produced also in the case of distributed stream processing systems. In fact, unless deployments of these systems are over-provisioned with resources (an usually undesired solution because highly cost ineffective), even short variations in the input rate of external data sources can cause increased processing *latency* due to PE queues getting longer, or random tuples drops when queues fill up.

A common and very simple solution is to allocate enough resources to sustain the load for most of the time and then to avoid (or limit) the growth in latency or random data drops by introducing *load shedding* mechanisms [25], which selectively drop tuples at strategic points in the processing flow to maximize some quality measure [29] or to minimize the amount of data lost [30].

More sophisticated solutions try to dynamically adapt to load variations while avoiding to drop any data. A first common approach is to move PEs between processing hosts to re-balance the system and accommodate new load conditions [22, 33, 35]. In [4], the authors develop a dynamic resource allocation algorithm that automatically re-distributes resources among PEs to maximize the expected throughput. More recently, a similar approach has been proposed by [8]. All these solutions effectively manage to handle load variations when the available resources are sufficient to handle the total load or, from another perspective, when there is no hard limit on the runtime cost of the solution.

In [24], the authors propose a dynamic priorities mechanism for the Stream MapReduce system [10] that automatically reduces the execution priority of task replicas during load spikes. Similarly to our proposal, this mechanisms permits to gather the resources necessary to handle the extra load; differently from LAAR, the proposed solution does not provide hard SLA guarantees about the possible information loss in worst-case failure scenarios and is not able to adapt the runtime cost of streams applications to their required fault-tolerance guarantees.

In this paper, we deal with this problem from a different and original perspective. Instead of handling load variation by sacrificing latency (queuing), completeness (load shedding), or increasing cost (resource over-provisioning), we collect the resources needed to cope with changing load conditions by leveraging the flexibility of weaker reliability requirements included in customer–provider SLAs. LAAR guarantees that these requirement are enforced at runtime and minimizes the application execution cost accordingly.

3. SERVICE MODEL

In this section we introduce a PaaS-based [31] stream processing service model for the commercial-relevant scenario where *service providers* host customers' applications according to a set of *service level agreements* (SLAs) that define the expected runtime behavior of hosted applications and the associated costs and pricing plans. By presenting this model, we set up the basic terminology that we use throughout the remainder of this paper, and we state the fundamental assumptions of our LAAR dynamic fault-tolerance approach.

In our model, stream processing services are regulated by customer–provider *contracts* composed of (i) the *stream processing application* to be executed on the platform, (ii) an *application descriptor* that characterizes the application components and the application input (e.g., its statistical properties, see the following), (iii) a *SLA* determining the targeted runtime quality requirements, and (iv) a *pricing plan* that defines the economical conditions under which the provider runs the customer application with the requested quality of service.

The *stream processing application* (or, hereinafter, simply application) consists of a set of software components and an application graph. The software components are one or more *Processing Elements* (PEs), at least one *data source*, and at least one *data sink*. A PE transforms one or more input data streams — theoretically unbounded sequences of structured tuples — into another stream (its output); data sources and data sinks retrieve input from external sources and write

[1]In fact, the content of this type of streams is normally both *temporally* and *spatially* redundant, for example, due to several sources reporting the value of a feature measured in the same geographical area, or due to consecutive measurements of a feature that changes more slowly than the observation period.

tuples to external destinations, respectively. The application graph arranges PEs, data sources, and data sinks as vertices of a directed acyclic graph, connected by edges representing communication channels. This data-flow based processing model is very general and can be mapped on the majority of state-of-the-art data stream management systems from academia [2, 1, 5, 7] and industries [13, 17, 26, 27].

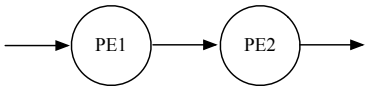
The *application descriptor* is a document summarizing, with a set of concise attributes, the computational behavior of PEs and the expected characteristics of application input streams. Similarly to what has been done in the literature (e.g., [16, 21, 30, 29, 32, 35]), application descriptors summarize PE behavior by using the metrics of *port selectivity* and *per-tuple CPU cost*. To be more specific, we associate every graph edge going into a PE to a selectivity value and a per-tuple CPU cost value: selectivity represents the weight of the contribution of an input data stream on the PE output stream; per-tuple CPU cost is the number of CPU cycles (on a given processing architecture) required on average to process a tuple from the stream associated to the edge. For simplicity of mathematical derivation, we assume a linear load model[2], but, with consideration similar to those in [32], our solution can be extended to nonlinear models as well. In the following discussion, we assume that PE selectivities and per-tuple CPU costs are either provided by the customer or extracted by the service provider through a preliminary *profiling* step [14]. The application descriptor also includes the expected characteristics of the external data sources: for each data source, the descriptor contains the probability distribution function describing the probability of the source to produce data at different tuple rates. We assume that the continuous space of possible tuple rates for each data source has been properly transformed in advance into a finite number of discrete data rates through, e.g., binning techniques [12]. Again, this information is specified by the customer or else inferred from a set of example input traces that she provides.

A *Service Level Agreement* (SLA) is a set of clauses specifying the desired runtime quality characteristics of the application. Two possible examples of SLA clauses are *maximum latency*, putting an upper bound on the time taken to produce an output after all the input data generating it has been received, or *fault-tolerance*, defining a guaranteed application behavior in case of failures.

Finally, the *pricing plan* determines the provider monetary revenue for running the customer application instance. We assume *continuous processing* applications, i.e., applications that run for an indefinite amount of time. As a result, we consider a *time-based, fixed* billing plan, according to which the customer pays a flat fare per billing period T. This fare depends on the characteristics of the application, of its input streams, and on the agreed SLA.

The service provider is expected to deploy and allocate computing resources so that the constraints imposed by SLA clauses are satisfied at runtime as long as, within each billing period T, *the characteristics of the external data streams reflect those specified in the contract*; if they do, the provider has to pay a penalty in case of SLA violations. The provider is interested in satisfying the quality requirements imposed by the SLA, while minimizing resource utilization. In this work, we assume that the service provider does always her best to avoid SLA violations.

[2]i.e., the output rate and the total CPU load of any PE can be expressed as a linear combination of the data source output rates and the PE selectivities and per-tuple CPU costs respectively.



	PE 1	PE 2
Selectivity	1	1
CPU Cost	0.1 s/tuple	0.1 s/tuple

	Low Rate	High Rate
Source	4 tuples/s	8 tuples/s
PE 1	4 tuples/s	8 tuples/s
Prob.	0.8	0.2

Figure 1: A simple processing scenario. Top: the application graph. Bottom: concise characteristics of the application and of its data source. For simplicity, data source and data sink are not shown.

4. LOAD-ADAPTIVE ACTIVE REPLICATION

LAAR is a novel and adaptive active replication method for data streams processing platforms that lets applications adapt to changing load conditions by temporarily trading perfect reliability for computational resources. It can provide *guaranteed* fault-tolerance levels, measured in terms of an upper bound on the information loss in case of failures, called *internal completeness* and defined in Section 4.3.

Similarly to traditional active replication techniques [16], LAAR deploys k replicas of every PE in the application dataflow graph: at any moment, one of the k replicas has the role of *primary*, the others are called *secondary*. Primary and secondary replicas all receive tuples from the primaries of their predecessor PEs, and all process them advancing through the same sequence of internal states. However, only the primary outputs tuples to the replicas of its successors. When a primary fails, one of the secondaries is elected as the new primary; once the failed replica is recovered, its state is synchronized with the non-failed ones before it becomes active again as secondary.

Originally, LAAR monitors the input rate of its application sources, and it dynamically and automatically activates and deactivates replicas in order to satisfy two goals:

1. The application deployment is never *overloaded*.
2. The *internal completeness* constraint expressed in the SLA is satisfied.

An application deployment is said to be overloaded when, for any host, the total CPU cycles per second that would be needed to execute the PEs assigned to it is bigger than the available CPU cycles per second. Note that, in an overloaded system, tuples accumulate at input queues of PEs (increasing latency) and are eventually dropped when the corresponding buffers fill.

4.1 LAAR in a simple application

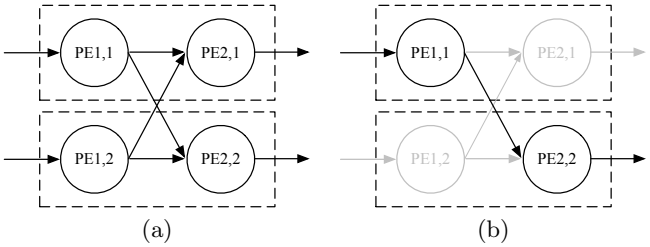


Figure 2: (a) Replicated deployment of the application of Fig. 1 on two hosts. (b) Dynamic deactivation of replicas by LAAR during a "High" input configuration.

Before presenting an in-depth analysis of the LAAR model and its fault-tolerance guarantees, we illustrate the basic intuition behind our approach in a minimal application scenario. Consider the application in Fig. 1: it consists of two PEs connected in a very simple pipeline; PE 1 processes data from a single data source (not reported in the figure for the sake of simplicity) and forwards its output to PE 2, which, in turn, sends the results of its computations to an external data sink (also not depicted in the figure). The selectivity of both PEs is 1, meaning that for every received input tuple they produce one output tuple; moreover, considering the CPU architecture of the deployment hosts, both PEs require 100 milliseconds to process an incoming tuple. The single data source can produce tuples at two different rates: "Low" and "High". The "Low" rate is 4 tuples per second and is active on average for 80% of the time (0.8 probability), while the "High" rate is 8 tuples per second and is active in the remaining time intervals (0.2 probability). The application is replicated and deployed on two hosts, each hosting a copy of each PE, as shown in Fig. 2a. It is straightforward to see that, when the input configuration is "Low", 80% of the CPU time available at both hosts will be occupied for processing tuples. More importantly, when the input configuration is "High", the application would need 160% of the total CPU time available, which — of course — is available only by adding extra resources to the deployment (with an increased cost).

The basic idea behind LAAR is to monitor the data sources and, according to the current data rates, to dynamically deactivate replicas in order to release the resources necessary to face load variations. For example, Fig. 2b shows how LAAR could deactivate two replicas of PE 1 and PE 2 during a load peak so that the total CPU available will become enough to handle the new load.

Fig. 3 shows this behavior in a real deployment. We implemented, deployed, and executed the replicated pipeline application in Fig. 2a on an IBM InfoSphere Stream® deployment consisting of two hosts equipped with a single core CPU. Fig. 3a reports the CPU usage and input/output rates of the application in time when static active replication is used: when the input passes to the "High" configuration (around 50 seconds from the beginning of the experiment), the CPUs of the two hosts saturate, and the application is not able to keep up with the input rate; on the contrary, by temporarily deactivating replicas during the "High" input configuration, it is possible to save enough resources to allow the output stream to follow the input (Fig. 3b).

Obviously, if a failure of one of the active PEs occurred

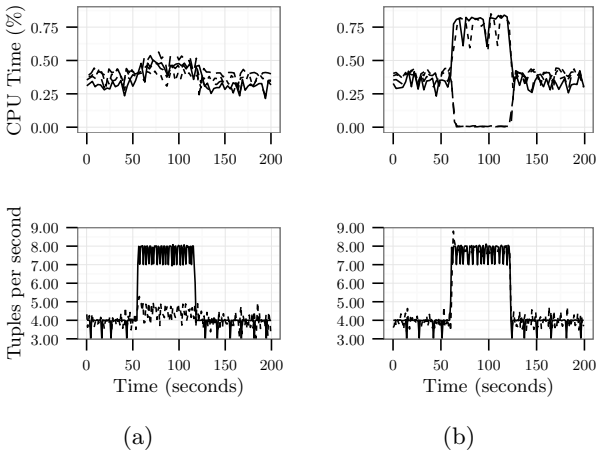


Figure 3: (a) CPU Time used by the replicated PEs — top — and corresponding input and output rate — bottom. (b) CPU time and input/output data rate when PE 1 replica 1 and PE 2 replica 0 are deactivated by LAAR. In the top graphs, different line styles correspond to different PE replicas; in the bottom graphs, the solid line corresponds to the input rate, the dashed one to the output rate.

during a "High" period, part of the input would not be processed as expected. As we will clarify in the remainder of this section, the unique and strong aspect of LAAR is its ability to quantify *a-priori* these effects on the overall application reliability.

4.2 Model and definitions

An application A consists of a set of components: a set I of data sources, a set P of PEs, and a set O of data sinks, which collectively define the set $X = I \cup P \cup O = \{x_i\}$. The components in X are arranged in a directed acyclic application graph $G = (X, E)$. The set of edges E is described by the function:

$$pred : X \mapsto \mathcal{P}(X) \tag{1}$$

which, for each component x_i, identifies the set of predecessor components $\{x_j\}$ so that $x_j \in pred(x_i) \Leftrightarrow (x_j, x_i) \in E$.

The characteristics of PEs are summarized by the selectivity function δ and the per-tuple CPU cost function γ: for each couple (x_i, x_j) so that $x_i \in I \cup P$ and $x_j \in P$ and that $(x_i, x_j) \in E$, $\delta(x_i, x_j)$ is the selectivity of PE x_j with respect to the tuples it receives from x_i, and $\gamma(x_i, x_j)$ is the per-tuple CPU cost for PE x_j to process tuples from x_i.

Every data source $x_i \in I$ can produce output at one rate among a finite set of input rates R_i. The Cartesian product $C = R_1 \times \ldots \times R_t$, where t is the number of sources, is the set of all the possible *input configurations*. As anticipated in Section 3, we assume to know $P_C : C \mapsto [0, 1]$, the probability mass function associated to the probability distribution of different input configurations in time. The output rate of data source $x_i \in I$ in a particular input configuration c is indicated as $\Delta(x_i, c)$. In absence of failures, it is straightforward to derive the expected output rate of each PE in any input configuration c; for uniformity of notation, we also indicate this value as $\Delta(x_i, c), x_i \in P$.

We assume that a PE placement algorithm among the many described in the literature (e.g., in [21] or [32]), computes a *replicated* assignment of k replicas of each of the PEs in P to a set of hosts $H = \{h_i\}$. We indicate the replicated set of PEs as:

$$\widetilde{P} = \{\tilde{x}_{i,j}\} \tag{2}$$

For simplicity of notation, we will use the symbol $\tilde{x}_{i,j}$ to indicate the j-th replica of PE x_i. The assignment is represented by the function:

$$\vartheta : \widetilde{P} \mapsto H \tag{3}$$

which maps every PE replica to the host where it is deployed. For convenience, we also define $\vartheta^{-1} : H \mapsto \mathcal{P}(\widetilde{P})$ such that $\vartheta^{-1}(h) = \{\tilde{x}_{i,j} \in \widetilde{P} : \vartheta(\tilde{x}_{i,j}) = h\}$.

A *replica activation strategy* is a function:

$$s : \widetilde{P} \times C \mapsto \{0, 1\} \tag{4}$$

that associates every PE replica – input configuration pair to one of the two possible active/inactive states.

4.3 The internal completeness (IC) metric

By activating/deactivating PE replicas according to the current input configuration, LAAR dynamically modifies the resilience of applications to failures. In order to measure the effect of LAAR on fault-tolerance guarantees, we define the *internal completeness* (IC) metric. Intuitively, given a failure model that describes how hosts and PEs are expected to fail and a *replica activation strategy* s, internal completeness measures, with respect to the billing period T, the fraction of total tuples that is expected to be processed in case of failures compared to the number of tuples that would be processed in absence of failures.

Let us examine the no-failure scenario (best-case) first: the total number of tuples that is statistically expected to be processed by the application PEs during billing period T is:

$$BIC = T \cdot \sum_{\substack{c \in C, \\ x_i \in P, \\ x_j \in pred(x_i)}} P_C(c) \cdot \Delta(x_j, c) \tag{5}$$

Best-case internal completeness (BIC) is the summation of the contributions of all the application PEs in different input configurations, weighted by the probability of each configuration to occur in T.

Failure internal completeness (FIC) measures the expected number of tuples processed given a failure model ϕ and a replica activation strategy s. It is defined as:

$$FIC(s) = T \cdot \sum_{\substack{c \in C, \\ x_i \in P, \\ x_j \in pred(x_i)}} P_C(c) \cdot \phi(x_i, c, s) \cdot \widehat{\Delta}(x_j, c, s) \tag{6}$$

$$\widehat{\Delta}(x_i, c, s) = \begin{cases} \Delta(x_i, c) & \text{if } x_i \in I \\ \phi(x_i, c, s) \cdot \sum_{x_j \in pred(x_i)} \delta(x_j, x_i) \widehat{\Delta}(x_j, c, s) & \text{if } x_i \in P \end{cases} \tag{7}$$

The function $\phi(x_i, c, s)$ depends on the chosen failure model and describes the probability that at least one replica of PE x_i is alive and active when the input configuration is c and the replica activation strategy is s. $\widehat{\Delta}(x_i, c, s)$, instead, represents the expected output of PE x_i under failure model ϕ, when the input configuration is c and the replica activation strategy is s; note that the definition of $\widehat{\Delta}$ is recursive, as the number of tuples produced by a PE depends not only on its possible failure status (described by ϕ) but also on the number of tuples produced by its predecessor (Equation 7).

Internal completeness (IC) is defined as the ratio between FIC and BIC:

$$IC(s) = \frac{FIC(s)}{BIC} \tag{8}$$

We choose the IC metric over other possible metrics (e.g., output completeness or average replication factor) for two main reasons. First, IC is easy to understand and measure. Second, and most relevant, IC captures not only the completeness of the application output (at the data sinks) but also the divergence, in a scenario with failures, of the state of PEs compared to a failure-free scenario, under the assumption that this divergence is proportional to the amount of tuples that are not processed.

4.4 LAAR replica activation strategies

In LAAR, the information in the application descriptor is used to compute — off-line and before application deployment — a replica activation strategy that fits the application fault tolerance requirements.

The cost minimization problem that is solved to determine the appropriate replica application strategy, given an application descriptor, is defined as follows:

$$\underset{s}{\text{minimize}} \qquad cost(s) \tag{9}$$

subject to:

$$IC(s) \geq SLA\ Constr. \tag{10}$$

$$\sum_{\substack{\tilde{x}_{i,h} \in \vartheta^{-1}(h), \\ x_j \in pred(x_i)}} \gamma(x_j, x_i)\, \Delta(x_j, c) s(\tilde{x}_{i,h}, c) < K \qquad \substack{\forall h \in H, \\ \forall c \in C} \tag{11}$$

$$\sum_{h=1}^{k} s(\tilde{x}_{i,h}, c) \geq 1 \qquad \substack{\forall x_i \in P, \\ \forall c \in C} \tag{12}$$

The *cost* function in the minimization term represents the cost, in terms of resources, for a service provider to run the application using replica activation strategy s and the replicated assignment defined by ϑ. In this work, we model the bandwidth available for cluster-local communication as an abundant resource, and we model our cost function as the total CPU time used by an application in a billing period T. It is defined as follows:

$$cost(s) = T \sum_{\substack{c \in C, \\ \tilde{x}_{i,h} \in \widetilde{P}, \\ x_j \in pred(x_i)}} P_C(c)\, \gamma(x_j, x_i)\, \Delta(x_j, c) s(\tilde{x}_{i,h}, c) \tag{13}$$

and is the summation over all PE replicas $\tilde{x}_{i,h}$ of the CPU time they consume in T.

Equation 10 constraints IC to satisfy the requested SLA value, while Equation 11 states that each host in the deployment should never be overloaded; K is a constant expressing the number of CPU cycles per second available at the deployment hosts. The last constraint, expressed in Equation 12, requires that there is at least one active replica of every PE in every input configuration, and it ensures that the measured IC value is one in absence of failures.

In order to solve the optimization problem, LAAR considers a simplified failure model ϕ, based on the following

assumptions:

1. In any failure scenario, all PE replicas fail except one.
2. Unless all the replicas are active at some point in time, the non-failed replica is chosen among the inactive ones.
3. Once failed, replicas never recover.

or, more formally:

$$\phi(x_i, c, s) = \begin{cases} 0 & \text{if } \sum_{h=1}^{k} s(\tilde{x}_{i,h}, c) < k,\ \tilde{x}_{i,h} \in \widetilde{P} \\ 1 & \text{otherwise} \end{cases} \quad (14)$$

The so defined model will in general overestimate possible failure conditions (it is highly unlikely that all PE replicas fail at the same time) and their consequences (normally failures would be recovered): for these reasons, we also refer to it as *pessimistic failure model*. However, this choice of ϕ provides two fundamental benefits:

- Since it overestimates the likelihood and effects of failures, the IC value computed using this model is a lower bound to the real IC that will be observed on the actual application deployment (see Section 5.3).
- Its mathematical formulation simplifies the computation of IC values for different possible replica activation strategies and hence the optimization complexity.

Note that the solution space of this problem is still very large, as for every application there are $2^{|P|\cdot|C|\cdot k}$ possible replica application strategies. Note also that the IC constraint (Equation 10) and the hosts CPU constraints (Equation 11) that appear in the cost function (Equation 13) depend on $\widehat{\Delta}(x_i, c, s)$ (Equation 7), which is a recursively defined exponential term. Hence, to find algorithms that can find optimal or good enough solutions to this problem is a major technical challenge. In the next section, we will present our FT-Search solution, a constraint-programming -based algorithm that shows how to find, in limited time, good solutions to instances of this problem.

Let us remark again that the optimization phase is performed off-line with respect to the execution of the stream processing application, so its complexity does not cause any direct overhead on the application runtime cost, which is, instead, minimal. In Section 4.6, we describe how LAAR online counterpart can be implemented as a thin middleware layer requiring little modifications to existing data streams processing architectures already supporting (static) active replication.

4.5 FT-Search

FT-Search is a depth-first search algorithm with backtracking that explores the tree of the possible PE activation states for the possible input configurations. Its implementation limits the search space by considering only two-fold replication ($k = 2$), practically restricting its size to $3^{|P|\cdot|C|}$.

While exploring the tree, the algorithm prunes branches that either cannot improve the current best solution (if any) or that would lead to solutions that violate problem constraints. FT-Search uses four pruning strategies to detect such conditions as soon as possible:

1. *Pruning on CPU constraint* (CPU). During the exploration of a branch, FT-Search calculates the current CPU load at any host: if the maximum CPU threshold is violated for any of them, the branch is pruned.
2. *Pruning on IC upper bound* (COMPL). During the exploration of a branch, an upper bound on the possible achievable IC is kept. This value is computed as the IC contributions provided by the PE replicas corresponding to already explored nodes plus the maximum possible IC that PEs corresponding to unexplored nodes could provide. If the IC upper bound is lower then the IC goal, the branch is pruned.
3. *Pruning on cost lower bound* (COST). Once a first feasible solution is found, its cost is saved and updated whenever a better solution is encountered. While exploring other branches, a lower bound on the minimum achievable cost is kept. If this value is greater than the cost of the best solution, the branch is pruned.
4. *Forward domain propagation* (DOM). This strategy is based on the observation that if, in some input configuration, all the predecessors of a PE have only one replica active, then having for it two replicas active would not improve the overall IC value, while it would increase the solution cost. We call this condition *no replication forwarding*. For this reason, during the exploration, whenever some search node is bound to a value corresponding to no replication, a quick routine is started. This routine verifies (if necessary, recursively) whether, for any successor of the PE whose replication values has just been bound, the *no replication forwarding* condition applies. If so, the value corresponding to twofold replication of such successor PE is removed from the domain of its search node.

Note that, in order to compute partial IC components while going down on a branch, the exploration must not violate the topological order [20] of the application graph. Apart from these restrictions, heuristics can be devised to choose the most effective exploration order: according to our experience, exploring nodes corresponding to the most resource hungry configurations first improves execution time by making both the CPU and IC constraints fail faster.

We have developed a highly parallel implementation of FT-Search based on the JSR166 Fork-Join framework [23], and we have tested it on a set of 600 applications to be deployed on 1 to 12 hosts, with 2 to 12 PEs per host. We have executed the algorithm on a machine with a 6-core Intel Xeon® X5690 @3.5Ghz processor. For all the algorithm runs, we have set a hard time limit to 10 minutes: after the deadline, the algorithm terminates and returns the best solution found.

Fig. 4 and 5 show some of the results. In particular, Fig. 4 shows what kind of solutions FT-Search finds within the time limit, with the IC constraint growing from 0.5 to 0.9. Note that, in most cases, the algorithm terminates either by finding an optimal (BST) or feasible (though not necessarily optimal) solution (SOL), or it shows that no feasible solution exists for the problem (NUL); on the contrary, it is not able to find any feasible solution, or to demonstrate that no solution exists for only a small number of instances (TMO). As the IC constraint grows, the number of algorithm instances

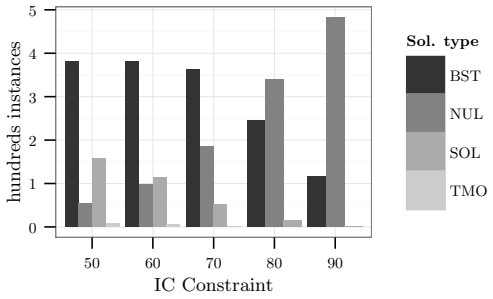


Figure 4: Types of solution found for different IC constraints: (BST) optimal solution, (SOL) feasible solution, (NUL) no solution exists, (TMO) time limit exceeded and no solution found.

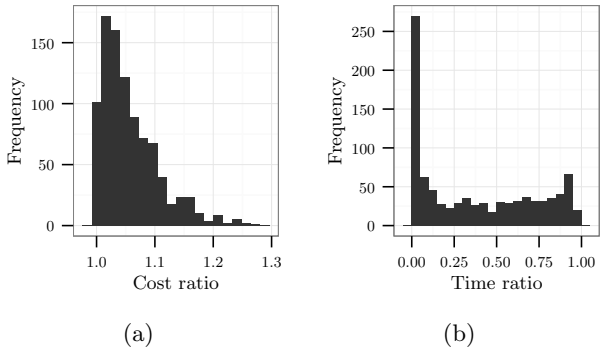


Figure 5: Cost (a) and search time (b) ratios between the first solution and the optimum (when an optimal solution is found).

that terminate demonstrating that there is no feasible solution clearly grows as well, but it is interesting to see that, the stricter the IC constraint, the fewer the situations where the algorithm terminates after finding at least a solution.

For the instances we were able to find an optimal solution for, we have measured the cost ratio between the first solution found and the optimum. A histogram representing the results is shown in Fig. 5a: the distribution is positively skewed, with a mean value of 1.057. At the same time, a first feasible solution is usually found much more quickly than the optimum: the histogram in Fig. 5b shows the distribution of the ratios between the time needed to find the first feasible solution and the optimum, whose mean value is 0.37. This is very important for our scenario, since it means that it is generally possible to accept sub-optimal solutions, thus saving considerable amounts of time at the price of minimal penalties on solution quality.

We also measured how often different pruning strategies were used (in average) and how effective they were. Fig. 6 summarizes the results. The IC-based strategy results to be the most applied pruning method, followed by forward domain propagation. However, CPU-based pruning is generally applied earlier in the search, thus cutting larger search subspaces; the cost-based strategy, finally, is both the least used and the least effective. This is probably due to the fact that the exploration needs to go deep down in the tree before a sufficiently tight cost lower bound can be found.

4.6 Runtime architecture

LAAR has been designed to be integrated with little effort with existing platforms that already offer static active replication and that support the model described in Section 3. The work flow used to deploy a LAAR-enabled application is schematically shown in Fig 7. The application descriptor, the IC SLA requirement, and the application itself (see again Section 3) are fed to two different components. The first implements the FT-Search algorithm described in the previous section and produces a replica activation strategy. The second component, i.e., the *Application Preprocessor*, modifies the original application to produce the *extended application*, which enhances the original user application with LAAR - specific mechanisms. In particular, as shown in Fig. 8, two special PEs are added to the original data-flow graph — the *Rate Monitor* and the *HAController* —, and the behavior of application PEs is extended in order for them to understand and accept *activation/deactivation* commands. The extended application is finally deployed on the actual distributed stream processing system (DSPS).

At runtime, the *Rate Monitor* PE periodically measures the data rates from sources and outputs this measurement result. The High Availability Controller (*HAController*), initialized at startup with the chosen replica activation strategy, receives the sources data rates from the Rate Monitor and, according to their values, it chooses the appropriate replica activation state based on the current input configuration. To achieve that quickly and effectively, it uses an R-Tree [15] -like data structure that selects the input configuration that is spatially closer to the current data rates and whose components are all greater than the corresponding actual rates. This choice guarantees that the chosen replica configuration will never underestimate the actual system load. Whenever a change in the replica configuration occurs, the HAController reliably sends activations or deactivation commands to PE replicas.

Application PEs behavior is also slightly modified to make them accept commands from the HAController. When deactivated, they immediately stop processing their input and transit into an *idle*, resource-saving state. On the contrary, when activated again, they re-synchronize their state with one of the active replicas and restart processing their input. Since this process is almost identical to the recovery

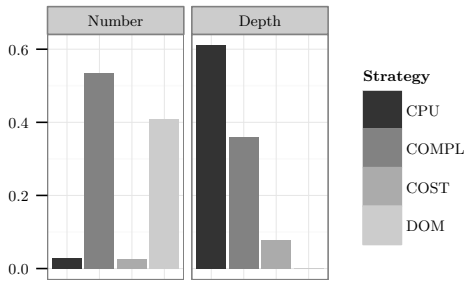


Figure 6: Effectiveness of pruning: relative number of domain values pruned (left), and average height of the pruned search branches (right).

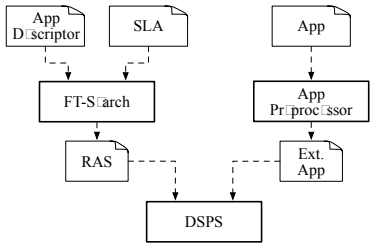


Figure 7: Deployment of a LAAR application.

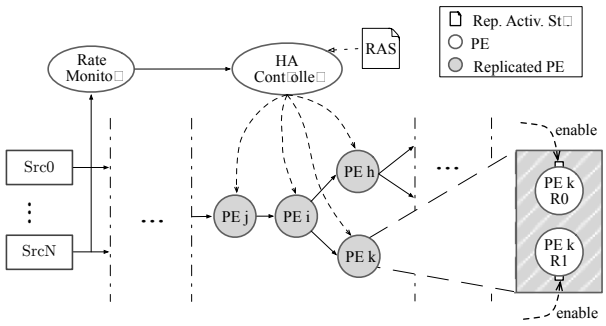


Figure 8: Structure of extended data-flow graphs.

of crashed PEs in traditional static replication systems [16], we will not detail it further in this paper.

Let us emphasize again that, since they do not require any particular platform-dependent functionality, both the Rate Meter and HAController can be implemented transparently on top of existing stream processing platforms as standard PEs. For what concern the enhancements needed on application PEs, they are minimal and can be implemented, for example, by dynamically proxying user provided PEs. In Section 5.1, we describe how we implemented this architecture on IBM InfoSphere Streams®.

5. PERFORMANCE EVALUATION ON IBM INFOSPHERE STREAMS®

We have implemented and deployed LAAR on top of an enterprise deployment of an industrial-strength stream processing system, IBM InfoSphere Streams®, and tested it on a set of artificially generated stream processing applications. In this section, after describing the main aspects of the implementation on Streams and the rationale behind our synthetic application generation process, we will present the performance results obtained by running these applications on a 60-core cluster.

5.1 LAAR on IBM InfoSphere Streams®

IBM InfoSphere Streams® [13] is a distributed stream processing platform evolved from the SPC research project [3]. In Streams, applications are written in an ad-hoc Stream Processing Language (SPL) that is used to describe *operators* and their stream connections. At compilation time, operators are transformed into their runtime counterparts, i.e., PEs, each executed, after application deployment, in its own process on the host system. The mapping from operators to PEs is usually many-to-one, as the Streams compiler can *fuse* [21] several operators into single PEs to minimize context-switching and communication overheads. At the time of writing, the only form of fault-tolerance supported natively by Streams is checkpointing [18].

In order to use LAAR, which leverages active replication support from existing platforms, we implemented a minimal active replication system on top of Streams, based on operator proxying. The same proxying technique is also leveraged to implement the replica activation/deactivation mechanism needed by LAAR at runtime. In more detail, in the application preprocessing step, the application SPL sources are modified by creating two replicas of every operator and by introducing, for each replica, a special *HAProxy* operator. This operator intercepts the input and output streams of the proxied replica and has the following functions:

- Accept *activate/deactivate* commands from the LAAR HAController. When active, HAProxy forwards all the input to the proxied operator replica and all its output to all the replicas of its successors; when inactive, all the input is ignored and no output tuple is forwarded.
- Send periodic heartbeat messages to the proxies of the replica's successors to indicate that the replica is alive.
- Receive heartbeats and input tuples from all the replicas of the proxied operator predecessors and forward only data from the current primary to the proxied operator.

Each proxy and its corresponding operator replica are fused into a single PE using the *partition co-location* setting.

Rate Meter PEs and HAController PEs are also inserted in the operator graph, the latter customized with the path to a JSON file describing the replica activation strategy to be used at runtime.

5.2 Experimental setup

We generated a corpus of different stream processing applications on which to test and validate our LAAR approach. To this purpose, we developed and used a generator that builds synthetic stream processing applications from a set of descriptive parameters. The output of the generation is an application descriptor, which is then transformed into a corresponding Streams application. Every PE in the generated application is mapped onto a deterministic Streams operator that behaves according to its concise attributes[3]. These operators are *stateless*, with no particular semantics associated with their output.

Our deployment environment consists of a 60-core IBM BladeCenter® cluster. Each node is equipped with one Intel Xeon® X5690 processor and 96 gigabytes of primary memory. The cluster runs an instance of InfoSphere Streams® v.2, with one of the servers hosting Streams management services only, and the remaining dedicated to the execution of PEs. In all the experiments, we used applications composed of 24 PEs — 48 PEs, considering the twofold replication (1 PE per logical CPU core) — deployed on the available servers to minimize inter-host communication. During the execution of the experiments, we periodically query Streams

[3]Tuple processing is simulated through busy wait cycles of configured length. Selectivity is implemented by producing an output tuple after receiving, from an input port, a number of tuples equal or greater than an integer multiple of its selectivity.

about the current status of all the PEs and log this information.

In this paper, we present the results of a set of experiments on 100 generated applications. The application graphs have an average outgoing node degree between 1.5 and 3, and the operators are generated with port selectivity values uniformly distributed between 0.5 and 1.5. An external source produces tuples at two possible input rates (labeled "Low" and "High"), both chosen from a uniform distribution between 1 and 20 tuples per second. The PEs' per-tuple CPU cost parameters are randomly generated ensuring that i) the deployment is not overloaded when all replicas are active and the input configuration is "Low" and that ii) it would instead be overloaded when all replicas are active and the input configuration is "High". Every experiment runs on a 5 minute long input trace, with the High input configuration being active for one third of the trace. All the PEs are configured with one queue for each input port, long enough to hold 2 seconds of tuples in the "High" input configuration; once a queue is filled up, new tuples are dropped.

For every application, we run experiments using six different replication approaches, henceforth referred to as *variants*. The first three variants use our LAAR approach: we used FT-Search to obtain replica activation strategies for three different IC requirements of 0.5, 0.6, and 0.7, labeled in the following as L.5, L.6, and L.7, respectively. In order to compare LAAR with other possible static an dynamic replication techniques, we also run experiments using the following other three variants:

- *Non Replicated* (NR). The application is deployed on the available resources with no PE replication. A NR variant is obtained starting from the PE activations for the "High" input rate from the D.5 variant, and modifying them to make sure that only a replica of each PE is ever active. The obtained activations are used for both the possible input configurations. This simple procedure permits to quickly obtain a non replicated deployment over all the cluster resources that guarantees that the system is never overloaded.

- *Static Replication* (SR). For every PE, both replicas are active all the time independently of the current input configuration.

- *Greedy* (GRD). A dynamic replica activation strategy is derived using the following greedy algorithm: starting from a static active replication setting, for every input configuration, redundant PE replicas are iteratively disabled until every host is non overloaded; at each algorithm iteration, an overloaded host is chosen, and the replica that consumes the most CPU is chosen for deactivation. A simple heuristic is used to prefer the deactivation of upstream PEs first.

5.3 Evaluation on BladeCenter® cluster

We present an evaluation of the LAAR approach compared to the different replication variants considering the following failure modes: i) No failure ever occurs (referred to as *best-case* scenario); ii) a replica of each PE is permanently crashed throughout the experiment according to the pessimistic failure model presented in Section 4.4 (referred to as *worst-case* scenario); iii) during the experiment, a random server crashes and is recovered after some time.

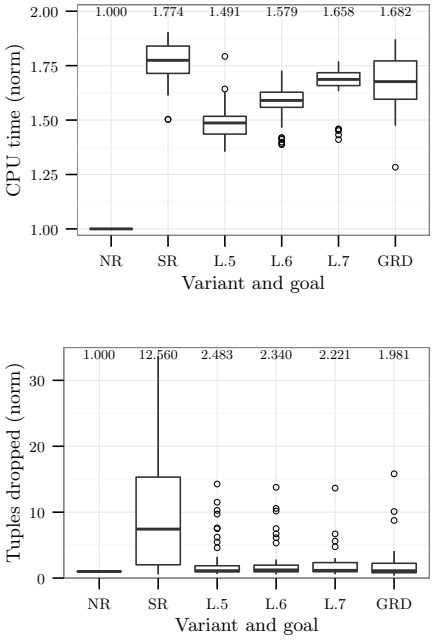


Figure 9: Distributions of the total CPU time used — top – and total number of tuples dropped — bottom — in a best-case experiment scenario, normalized w.r.t. the NR variant. Labels correspond to mean values.

Many of the results in this section are presented through box plots[4], which show how metrics of interest are distributed across executions of different applications when different dynamic replication variants are used. We do not differentiate by other graph parameters (e.g., average node degree) since our experiments have shown that their values do not affect the performance results relevantly.

Fig. 9 (top) shows the distribution of the total CPU time used to process all the input traces when using the considered variants in a best-case scenario. To compare measurements from different applications, the results are normalized with respect to the value measured when a non-replicated deployment is used (NR). As expected, static active replication (SR) is the variant using the highest amount of CPU time to process the same trace, with the overhead due to active replication being between 61% and 90% (note that this is not 100% since the deployment cluster does not have, by design, enough resources to handle the load peak, i.e., twice the CPU time consumed by NR). As expected, the greedy (GRD) variant is the second most expensive one because it deactivates "just enough" replicas for the system not to be overloaded. The three LAAR variants result to be the cheapest solutions in terms of resource use and, most interestingly, the cost of each of them is proportional to the IC value requested. This is a very important feature of our solution because it gives a direct way to correlate the desired reliability level to its runtime cost.

[4]Each box in a box plot shows the 25^{th}, 50^{th} and 75^{th} percentiles of a population of values. The ends of the whiskers represent the smallest (biggest) sample within 1.5 times the inter-quartile range, and circles represent outliers.

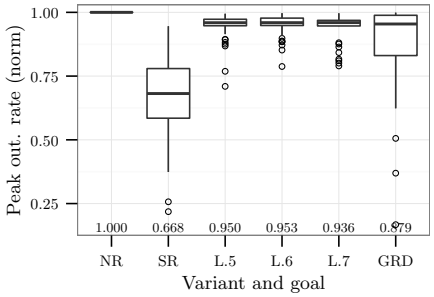


Figure 10: Applications output data rate during load peak, normalized w.r.t. the NR variant. Labels correspond to mean values.

Fig. 9 (bottom) analyzes, in the same best-case scenario, the ability of the studied variants to efficiently use the available resources by adapting to changing input configurations. In particular, we measured the number of tuples dropped due to queues filling up during the experiment. Note that static replication, not having any form of dynamic adaptation to the input rate, can drop up to 33.6 times more tuples compared to the non-replicated deployment, with a very high variance due to the different characteristics of different applications. In all the dynamic variants, instead, the amount of tuples dropped is much smaller, even if it is not zero mainly because of the effects of glitches on the input rate (a phenomenon that could be smoothed by carefully tuning PE queue lengths).

Another important element we looked at for the evaluation of our approach is the output tuple rate of applications during load peaks. In fact, this value directly depends on resource availability and on the ability of the platform to effectively use these resources. For each different application, we measured the average output data rate in the over-provisioned and non-replicated configuration (NR) and used it as reference to evaluate the other variants. Fig. 10 shows the distribution of the output data rate ratios measured for all the applications and variants. When static active replication (SR) is used, the applications output rate is on average 33% slower than the NR variant (up to 63%), while, when LAAR is used, the rate is at most 9% slower. The greedy variant (GRD) is in general better than the static one, but, differently from LAAR, it is not able to provide a consistent behavior across different applications, with an output rate ratio that is from 2% to 38% slower than NR.

We ran the same applications also assuming the pessimistic failure model defined in Section 4.4 in order to verify whether our system was actually able to satisfy the promised IC requirements in real stream processing deployments. The graph at the top of Fig. 11 shows the distribution of the normalized total number of samples produced by PEs in the worst-case scenario for all the replication variants. While the NR variant fails to produce any output (recall that in this failure model one active replica of each PE is permanently failed), the three LAAR variants are able to produce a fraction of output tuples that satisfies their respective IC requirement, except in a very limited number of cases, in which the violation is still never bigger than 4.7%. On the contrary, the GRD variant, while performing well in many

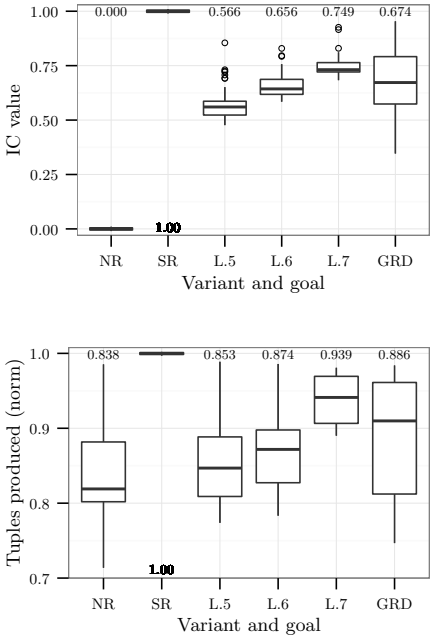


Figure 11: Total number of samples processed simulating the pessimistic (worst-case) failure model — top — and a single server crash model — bottom — normalized w.r.t. the failure-free NR variant (SR is introduced for ease of comparison). Labels correspond to mean values.

cases, is not able to provide a consistent behavior across different application, with measured IC values that can be as big as 0.95 but also as low as 0.35.

Fig. 12 summarizes the presented results showing the average numbers of tuples dropped, the average IC value, and the average cost of different replication strategies compared to the static active replication variant. It is immediate to see that LAAR permits to control application execution costs by tuning the desired IC guarantees, a crucial property in our business application scenario.

Finally, in order to evaluate the system behavior when more realistic failure scenarios are considered, we re-executed a randomly sampled subset of 40 applications, using a *single host crash-failures with recovery* failure model. In practice, during each experiment run, we randomly crash one of the PE hosting servers. The failure lasts for 16 seconds, i.e., the time needed, according to the experiences in [19], for Streams to detect it and migrate failed PEs to another host. We force these failures to occur only during "High" input configurations, because that is the case when LAAR provides the weaker fault-tolerance guarantees (thus disfavoring our solution). Fig. 11 (bottom) shows the IC values measured for the different variants in this scenario. As expected, the IC measured for the LAAR variants is much higher than their guaranteed values, given the less pessimistic failure model. Note that the results for L.5 are similar to those measured for the no-replication variant: recall, in fact, that the NR deployment is derived from L.5 by deactivating the replicas which have still two active replicas in the "High" configuration (which are usually just a few). Once again,

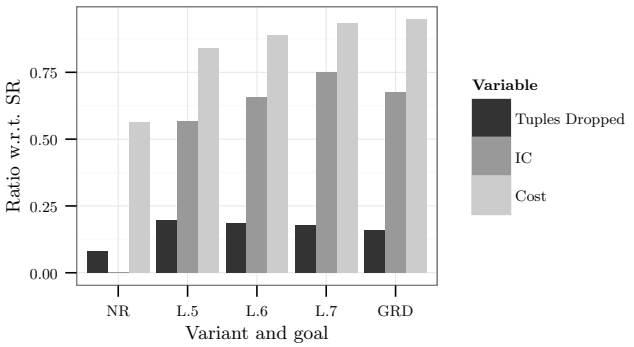


Figure 12: Summary: comparison of the different variants (mean values normalized w.r.t. SR)

the GRD variant confirms its unpredictable behavior in the way it responds to failures.

6. CONCLUSIONS AND FUTURE WORK

Stream processing service providers may need to temporarily provision additional resources to hosted applications during load spikes, if avoiding to drop or delay application tuples is a requirement. In this paper, we have presented LAAR, a novel technique for dynamic active replication that reduces the costs of provisioning these resources by enforcing weaker fault tolerance guarantees for applications that can tolerate them. In particular, LAAR can temporarily gather CPU resources by dynamically activating and deactivating PE replicas according to the current system load. We have implemented LAAR on top of IBM InfoSphere Streams® and deployed it on a 60-cores cluster. Our evaluation shows that LAAR can effectively trade-off runtime costs with availability while still enforcing completely predictable and guaranteed fault-tolerance levels.

We are continuing our work on LAAR in two main directions. First, we are trying to improve the underlying model by i) investigating the use of alternative failure models in the optimization problem with the goal of providing tighter lower bounds on IC values, ii) considering a penalty model associated to IC violations and using IC constraints as minimization terms, and iii) extending the problem formulation by considering the interaction of replica placement with optimal replica activation strategies. Second, we will continue to work on the experimental evaluation of our system by testing it on complex real world applications. Let us finally note that, although we have presented LAAR in the context of stream processing, we deem it applicable to the much larger domain of distributed data flow systems that can tolerate incompleteness.

7. REFERENCES

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proc. of the 2nd Biennial Conference on Innovative Data Systems Research*, CIDR 2005, pages 277–289, Asilomar, CA, USA, 2005. The VLDB Endowment.

[2] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[3] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. Spc: a distributed, scalable platform for data mining. In *Proc. of the 4th International Workshop on Data-Mining and Statistical Science*, DMSS2006, pages 27–37, Sapporo, Japan, 2006. ACM.

[4] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proc. of the 26th IEEE Conference on Distributed Computing Systems*, ICDCS 2006, pages 71–78, Lisboa, Portugal, 2006. IEEE.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical report, Stanford InfoLab, 2004.

[6] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1), 2008.

[7] P. Bellavista, A. Corradi, and A. Reale. Design and implementation of a scalable and qos-aware stream processing framework: the quasit prototype. In *Proc. of the 2012 IEEE International Conference on Cyber, Physical and Social Computing*, CPSCOM 2012, pages 458–467, Besançon, France, 2012. IEEE.

[8] I. Boutsis and V. Kalogeraki. Radar: adaptive rate allocation in distributed stream processing systems under bursty workloads. In *Proc. of the 31st Symposium on Reliable Distributed Systems*, SRDS 2012, pages 285–290, Irvine, CA, USA, 2012. IEEE.

[9] A. Brito, F. C., and P. Felber. Multithreading-enabled active replication for event stream processing operators. In *Proc. of the 28th Symposium on Reliable Distributed Systems*, SRDS2009, pages 22–31, Niagara Falls, NY, USA, 2009. IEEE.

[10] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Proc. of the 3rd International Conference on Cloud Computing Technology and Science*, CloudCom 2011, pages 48–58, Athens, Greece, 2011. IEEE.

[11] Z. Cai, V. Kumar, B. Cooper, G. Eisenhauer, K. Schwan, and R. Strom. Utility-driven proactive management of availability in enterprise-scale information flows. In *Proc. of the ACM/IFIP/USENIX 7h International Middleware Conference*, Melbourne, Australia, 2006. Springer.

[12] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *Proc. of the 12th International Conference on Machine Learning*, ICML '95, pages 194–202, Tahoe City, CA, USA, 1995. Morgan Kaufmann.

[13] B. Gedik and H. Andrade. A model-based framework for building extensible, high performance stream processing middleware and programming language for ibm infosphere streams. *Softw. Pract. Exper*, 42(11):1363–1391, 2012.

[14] B. Gedik, H. Andrade, and K.-L. Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *Proc. of the 18th Conference on Information and Knowledge Management*, CIKM 2009, pages 847–856, Hong Kong, China, 2009. ACM.

[15] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, Boston, Massachusetts, 1984. ACM.

[16] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of the 21st International Conference on Data Engineering*, ICDE 2005, pages 779–790, Tokyo, Japan, 2005. IEEE.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of the 2nd ACM SIGOPS European Conference on Computer Systems*, volume 41 of *EuroSys 2007*, pages 59–72, Lisbon, Portugal, 2007. ACM.

[18] G. Jacques-Silva, B. Gedik, H. Andreade, and K.-L. Wu. Language level checkpointing support for stream processing applications. In *Proc. of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2009, pages 145–154, Estoril, Portugal, 2009. IEEE.

[19] G. Jacques-Silva, B. Gedik, H. Andreade, K.-L. Wu, and R. Iyer. Fault injection-based assessment of partial fault tolerance in stream processing applications. In *Proc. of the 5th International Conference on Distributed Event-based Systems*, DEBS 2011, pages 231–242, New York, NY, USA, 2011. ACM.

[20] A. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

[21] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. Cola: optimizing stream processing applications via graph partitioning. In *Proc. of the ACM/IFIP/USENIX 10th International Middleware Conference*, Urbana Champaign, IL, USA, 2009. Springer.

[22] V. Kumar, B. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Proc. of the 2nd International Conference on Autonomic Computing*, ICAC 2005, pages 3–14, Seattle, WA, USA, 2005. IEEE.

[23] D. Lea. A java fork/join framework. In *Proc. of the ACM 2000 conference on Java Grande*, pages 36–43, San Francisco, CA, USA, 2000. ACM.

[24] A. Martin, C. Fetzer, and A. Brito. Active replication at (almost) no cost. In *Proc. of the 2011 IEEE International Symposium on Reliable Distributed Systems*, SRDS 2011, pages 21–30, Madrid, Spain, 2011. IEEE.

[25] R. Motwani, J. Widom, A. Arasu, B. Babcokc, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proc. of the 1st Biennial Conference on Innovative Data System Research*, CIDR 2003, Asilomar, CA, USA, 2003. The VLDB Endowment.

[26] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proc. of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW 2010, pages 170–177, Sydney, Australia, 2010. IEEE.

[27] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: reliable stream computation in the cloud. In *Proc. of the ACM SIGOPS European Conference on Computer Systems*, EuroSys 2013, Prague, Czech Republic, 2013. ACM.

[28] M. Shah, J. Hellerstein, and E. Brewer. Highly available, fault-tolerant parallel dataflows. In *Proc. of the 2004 ACM International Conference on Management of Data*, SIGMOD 2004, pages 827–838, Paris, France, 2004. ACM.

[29] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *Proc. of the 33rd International Conference on Very Large Data Bases*, VLDB 2007, Vienna, Austria, 2007. The VLDB Endowment.

[30] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniacak, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th International Conference on Very Large Data Bases*, VLDB 2003, pages 309–320, Berlin, Germany, 2003. The VLDB Endowment.

[31] L. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2008.

[32] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proc. of the 32nd International Conference on Very Large Data Bases*, VLDB 2006, Seoul, Korea, 2006. The VLDB Endowment.

[33] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. of the 21st International Conference on Data Engineering*, ICDE 2005, pages 791–802, Tokyo, Japan, 2005. IEEE.

[34] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. A hybrid approach to high availability in stream processing systems. In *Proc. of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS 2010, pages 138–148, Genoa, Italy, 2010. IEEE.

[35] Y. Zhou, B. Ooi, T. Kian-Lee, and W. Ji. Efficient dynamic operator placement in a locally distributed continuous query system. In *Proc. of the 2006 Confederated international conference On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, OTM 2006, Montpellier, France, 2006. Springer.