JetStream: Enabling High Performance Event Streaming across Cloud Data-Centers

Radu Tudoran
IRISA/ENS Rennes, France
radu.tudoran@irisa.fr

Olivier Nano
Microsoft Research – ATL
Europe, Munich, Germany
olivier.nano@microsoft.com

Ivo Santos
Microsoft Research – ATL
Europe, Munich, Germany
ivo.santos@microsoft.com

Alexandru Costan
IRISA/INSA Rennes, France
alexandru.costan@inria.fr

Hakan Soncu
Microsoft Research – ATL
Europe, Munich, Germany
hakan.soncu@microsoft.com

Luc Bougé
ENS Rennes, France
luc.bouge@ens-rennes.fr

Gabriel Antoniu
Inria Rennes-Bretagne
Atlantique, Rennes, France
gabriel.antoniu@inria.fr

ABSTRACT

The easily-accessible computation power offered by cloud infrastructures coupled with the revolution of Big Data are expanding the scale and speed at which data analysis is performed. In their quest for finding the *Value* in the *3 Vs* of Big Data, applications process larger data sets, within and across clouds. Enabling fast data transfers across geographically distributed sites becomes particularly important for applications which manage continuous streams of events in real time. Scientific applications (e.g. the Ocean Observatory Initiative or the ATLAS experiment) as well as commercial ones (e.g. Microsoft's Bing and Office 365 large-scale services) operate on tens of data-centers around the globe and follow similar patterns: they aggregate monitoring data, assess the QoS or run global data mining queries based on inter site event stream processing. In this paper, we propose a set of strategies for efficient transfers of events between cloud data-centers and we introduce JetStream: a prototype implementing these strategies as a high performance batch-based streaming middleware. JetStream is able to self-adapt to the streaming conditions by modeling and monitoring a set of context parameters. It further aggregates the available bandwidth by enabling multi-route streaming across cloud sites. The prototype was validated on tens of nodes from US and Europe data-centers of the Windows Azure cloud using synthetic benchmarks and with application code from the context of the Alice experiment at CERN. The results show an increase in transfer rate of 250 times over individual event streaming. Besides, introducing an adaptive transfer strategy brings an additional 25% gain. Finally, the transfer rate can further be tripled thanks to the use of multi-route streaming.

Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Data sharing; D.2.4 [**Distributed Systems**]: Distributed applications

Keywords

Event Streaming, Cloud Computing, Multi Data-Centers, High Performance Data Management

1. INTRODUCTION

On-demand resource provisioning, coupled with the pay-as-you-go model, have redefined the way data is processed nowadays. Scientific applications, simulations, monitoring, networks of sensors, trading or commercial data mining are all areas which have benefited from and supported the emergence of cloud computing infrastructures. Cloud providers like Microsoft, Amazon, Google, Rackspace have built their cloud solutions at a global scale with data-centers spread across numerous geographical regions and continents. At the same time, the versatility of data analysis has increased with applications running on multiple sites, which have to process larger data sets coming from remote locations and distinct sources. The services dealing with such computations have to face important challenges and issues regarding the management of data across these geographically distributed environments.

Stream data processing is becoming one of the most significant subclass of applications in the world of Big Data. Vast amounts of stream data are collected at increasing rates from multiple sources [20, 29] in many areas: climate monitoring, large-scale sensor-based systems, transactional analysis, financial tickers, monitoring and maintenance systems for web services, large-scale science experiments. Acquiring, processing and managing this data efficiently raise important challenges, especially if these operations are not limited to a single geographical location. There are several scenarios which created the need to geographically distribute the computation on clouds. *The size of the data* can be so big that data have to be stored across multiple data-centers. It is the case of the ATLAS CERN experiment which generates 40 PB of data per year. Furthermore, even incremental processing of such a data set as a stream of events will overpass the capacity of local scientific infrastructure, as it was the case of the Higgs boson discovery which had to extend the computation to the Google cloud infrastructure[2]. Another scenario is given by *the data sources* which can be physically distributed in wide geographical locations as in the Ocean Observatory Initiative [5, 12] in which the collected events are streamed to Nimbus [9] clouds. Finally, *the nature of the analysis*, for an increasing number of services, requires aggregating streams of data from remote application instances. Large-scale services like Microsoft's Bing and Office 365 operate on tens of data-centers around the Globe. Maintenance, monitoring, asserting the QoS of the system or global data mining queries all require (near) real-time inter site event stream processing. All such computations carried on continuous streams of events across resources from different geographical sites are highly sensitive to the efficiency of the data management.

An extended survey over thousands of commercial jobs and millions of machine hours of computation, presented in [23], has revealed that the execution of queries is event-driven. Furthermore the analysis shows that the input data accounts only for 20% of the total IO, the rest corresponding to the replication of data between query services or to the intermediate data passed between them. This emphasizes that the processing services, be they distributed, exchange large amounts of data. Additionally, the analysis highlights the sensitivity of the performance of the processing of streams to the management and transfer of events. This idea is discussed also in [20], in which the authors stress the need for a high performance transfer system for real-time data. A similar conclusion is drawn in [10], where the issues which come from the communication overhead and replication are examined in the context of state-based parallelization. Finally, in [40], the authors emphasize the importance of *data freshness*, which improves the Quality of Service of a stream management system and implicitly the quality of the data (QoD). All these research efforts support and motivate the growing need for a high performance system for event streaming.

In this paper, we address challenges like latency, transfer rate, throughput and performance variability for supporting high performance event transfers in the context of geographically distributed applications. Our solution, called JetStream, is a novel approach for streaming events across cloud data-centers. In order to enable a high performance transfer solution, we leverage batch-based transfers. The size of the batches and the decision on when to stream the events are controlled by modeling the latency based on a set of parameters which characterize the streaming in the context of clouds. To further improve performance, we aggregate inter-data-center bandwidth as we extend our approach to provide multi-route streaming across cloud nodes. Furthermore, we developed a prototype that can be used on clouds, either public or private, which is environment-aware by means of light monitoring. The information gathered and inferred about the environment is coupled with the streaming model, allowing the system to adapt the transfer decisions to all context changes. The system was validated on the Windows Azure [6] cloud using synthetic benchmarks and in a real life scenario using the MonALISA [33] monitoring system of the CERN Alice experiment [1]. The experiments show performance improvements of 250 times over individual event streaming and 25% over static batch streaming, while multi-route streaming can further triple the transfer rate.

The rest of the paper is structured as follows. Section 2 introduces the terminology and present the problems which appear in the context of streaming across cloud data-centers. Section 3 presents our approach for adapting the batch size by modeling the latency for clouds, while considering the cost of multi-routing. We continue with the description of the system architecture and its implementation in Section 4 and discuss the evaluation of the approach across Azure data-centers from different continents (i.e. Europe and America) in Section 5. In Section 6, we present an extensive state of the art in several domains tangent to streaming and data management and position the contributions of our approach with respect to these directions. Finally, Section 7 concludes the paper.

2. BACKGROUND: EVENT STREAMING ON CLOUDS

We start by defining the terminology related to stream processing on clouds and discuss the main problems which need to be addressed in this context.

2.1 Context

Cloud computing is a recent paradigm which enables resources (e.g. compute power, storage capacity, network bandwidth) to be easily leased on-demand, following a pay-as-you-go pricing model. From the infrastructure point of view, clouds are built across several *data-centers* distributed around the globe. The proximity of the geographical repartition of resources to users enables high QoS and fast query responses by means of content delivery networks (CDN). From the user point of view, all these resources are harvested transparently by means of virtualization. A typical cloud usage scenario consists of a user *deployment* through which an application is run within *virtual machines* (VMs) on several cloud nodes from a data-center. Large-scale applications and web services typically run multiple deployments across several data-centers, enabling geographical proximity to users and load balancing. Besides the local computations answering to user requests, several global operations (e.g., monitoring, QoS enhancement, global data mining) require *inter-site transfers* between geographically distant nodes, most often in (near) real-time. The cloud support for such transfers is rather rudimentary: it relies on the cloud storage (e.g., Azure Blobs, Queues, Tables or Amazon S3, SQS) as a global shared point which incurs high transfer latencies [24, 45].

Stream computing refers to the processing of continuous sequences of relatively small data items [20], otherwise referred to as events. According to [17], the characteristics of *data streams* are: 1) a stream is formed of *events*, which are tuples of records; 2) it is a potentially infinite sequence of events ordered by time (e.g., acquisition time, generation time); and 3) the events can be produced by multiple external sources at variable rates, independent of the constraints of the stream processing engine. The *stream*

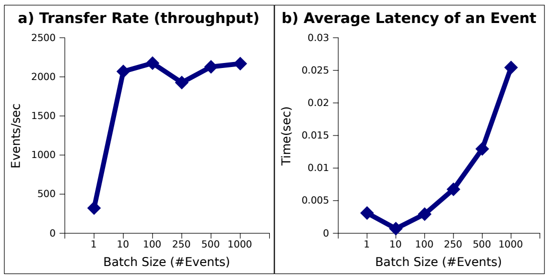

Figure 1: The performance of transferring events in batches between North Europe and North US Azure data-centers. The measurements show the correlation between (a) the transfer rate and (b) the average event latency for increasing batch sizes.

processing engines are systems which treat the events from the stream by applying the user's queries. As it is generally infeasible to store the streams entirely, the *queries* are applied continuously and iteratively over windows of events. The computation is *event-driven*, therefore most of the data management is done at the granularity of an event. Generally, the size of an event is small, in the range of bytes to kilobytes, which allows to handle them in-memory, but also makes their transfer sensitive to any overhead.

2.2 Problem statement

Achieving high performance event streaming across datacenters requires new cloud-based solutions, since currently there are no adequate tools to support data streaming or transfers across sites. Most of the existing stream processing engines only focus on event processing and provide little or no support for efficient event transfers, simply delegating this functionality to the event source. Today, the typical way to transfer events is individually (i.e., *event by event*). This is highly inefficient, especially across WAN, due to the incurred latencies and overheads at various levels (e.g., application, technology tools, virtualization, network).

A better option is to transfer events in *batches*. While this improves the transfer rate, it also introduces a new problem, related to selecting the proper batch size. Figure 1 presents the dependency between the batch size and the transfer rate, and the transfer latency per event, respectively. We notice that the key challenge here is the choice of an optimal batch size and the decision on when to trigger the batch sending. This choice strongly relies on the streaming scenario and on the sensed environment (i.e. the cloud). We aim to tackle these problems by proposing an environment-aware solution, which enables optimum-sized batch streaming of events in the clouds. To achieve this, we model the latency of the event transfer with respect to the cloud environment, dynamically adapt the batch size to the context and enable multi-route streaming across clouds nodes.

3. MODELING THE STREAMING OF DATA IN THE CONTEXT OF CLOUDS

The cloud variability and the fluctuating event generation rates require an appropriate model for event streaming, able to capture the cloud specificities. In this section, we introduce such a model and present the decision mechanisms for selecting the number of events to batch.

3.1 Design Principles

JetStream relies on the following set of design principles:

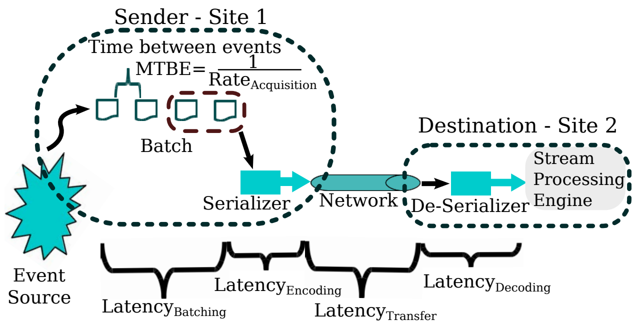
Sender - Site 1
Time between events
MTBE= $\frac{1}{\text{Rate}_{\text{Acquisition}}}$
Batch
Destination - Site 2
Stream Processing Engine
Serializer
Network
De-Serializer
Event Source
$\text{Latency}_{\text{Batching}}$
$\text{Latency}_{\text{Encoding}}$
$\text{Latency}_{\text{Transfer}}$
$\text{Latency}_{\text{Decoding}}$

Figure 2: Breaking down the latency to deliver an event from source to stream processing engine across cloud nodes

Environment awareness - The cloud infrastructures are subject to performance variations due to multi-tenancy and network instability on the communication links within and across sites. Monitoring and detecting such performance changes allows the system to react accordingly and schedule the transfer of events efficiently.

Decoupling the transfer from processing - The event transfer module needs to be designed as a stand-alone component, decoupled from the stream processing engine. We advocate this solution as it allows seamless integration with any engine running in the cloud. At the same time, it provides sustainable and predictable performance, independent on the usage setup.

Self-optimization - User configurations do not guarantee optimal performance, especially in dynamic environments. Moreover, when it comes to large-scale systems, the tasks of configuring and tuning the service tends to become complex and tedious. The alternative is to design autonomic cloud middleware, able to self-optimize. Coupled with an economic model, these systems could also regulate the resource consumption and enforce service-level agreements (SLAs).

Generic solution - Building specific optimizations which target precise applications is efficient, but limits the applicability of the solution. Instead, we propose a set of techniques which can be applied in any cloud context, independent of the application semantics. JetStream does not depend on the nature of the data, nor on the query types.

3.2 Zoom on the event delivery latency

We express the latency of the events based on a set of cloud parameters which can be monitored. Such a technique allows to bind the batch size corresponding to the minimal event latency both to the stream and the environment information. We start by breaking down the end-to-end latency of an event in four components, depicted in Figure 2: forming the batch, encoding (e.g., serializing), transferring the batch and decoding (e.g., de-serializing). The set of parameters able to describe the context and define the latency is: the average acquisition rate ($\text{Rate}_{\text{Acquisition}}$) or mean time between events (MTBE), the event size ($\text{Event}_{\text{SizeMB}}$), the serialization/deserialization technique, the throughput (thr) and the number of events to put in the batch (i.e., batch size - $\text{batch}_{\text{Size}}$). The goal is to determine the latter dynamically. In the following, we will model the latency components using these parameters.

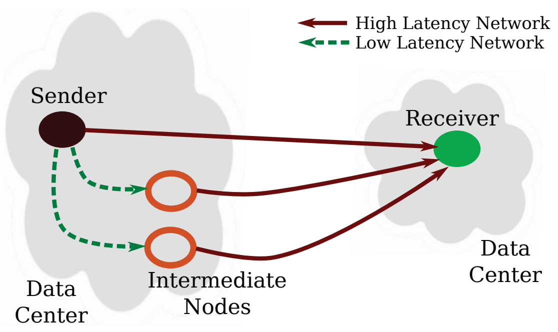

Figure 3: The proposed schema for multi-route streaming across cloud nodes.

The batching latency corresponds to the delay added when an event is waiting in the batch for other events to arrive, before they are all sent as a batch. The parameters which describe this latency are the average acquisition rate of the events and the number of events in the batch. As the delay depends on the position of the event in the batch, we chose to use the average latency per event. This can be computed by averaging the sum of the delays of all events in the batch: $\text{Latency}_{\text{batching}} = \frac{\text{batch}_{\text{Size}}}{2\times\text{Rate}_{\text{Acquisition}}}$. Intuitively, this corresponds to the latency of the middle event.

The transformation latency groups the times to encode and to decode the batch using a specific serialization library. It depends on the used format (e.g. binary, JSON etc.), the number of bytes to convert and the number of events in the batch. To express this, we represent the transformation operation as an affine function (i.e. $f(x) = ax + b$) where a corresponds to the conversion rate (i.e. amount of bytes converted per second - tDs), while the b constant gives the time to write the metadata (tHs). The latency per event can be expressed as: $\text{Latency}_{\text{transformation}} = \frac{tHs + tDs\times\text{batch}_{\text{SizeMB}}}{\text{batch}_{\text{Size}}}$ which holds both for the encoding and decoding operations. This formula can also be applied to express the size of the data to be transferred by simply replacing the time constants with data related constants (i.e. size of the metadata and the compression/extension ratio).

The transfer latency models the time required to transfer an event between cloud nodes across data-centers. To express it, we consider both the amount of data in the batch as well as the overheads introduced by the transfer protocol (e.g. HTTP, TCP) - sOt and the encoding technique - sOe. Due to the potentially small size of data transferred at a given time, the throughput between geographically distant nodes cannot be expressed as a constant value. It is rather a function of the total batch size ($\text{Size}_{\text{Total}} = \text{batch}_{\text{SizeMB}} \times \text{batch}_{\text{Size}}$), since the impact of the high latency between data-centers depends on the batch size. The cloud inter data-center throughput - $thr(\text{Size})$ is discussed in more detail in the following section. The average latency for transferring an event can then be expressed as $\text{Latency}_{\text{transfer}} = \frac{sOt + sOe + \text{batch}_{\text{SizeMB}}}{thr(\text{Size}_{\text{Total}})}$

3.3 Multi-route streaming

The throughput is one of the most peculiar aspects when considering communication in the clouds. On the one hand, *the intra-data-center throughput* (between the nodes within one's single site deployment) is relatively stable and a priori known, as it is specified by the cloud provider within the SLA (e.g., 100 Mbps for Small Azure VMs). On the other hand, there are no guarantees when it comes to *the inter-data-center throughput* (between the data-centers). In fact, the high latency and generally low throughput network connecting the sites is not owned by the cloud provider, as the packet routing is done across multiple internet service providers (ISPs). This makes it more unstable, unknown in advance and subject to poor performance.

In order to address the issue of low inter-data-center throughput, we designed a transfer strategy suitable for a typical cloud setup (i.e., application running across several nodes in different cloud data-centers deployments), which can harvest extra bandwidth by using *additional intermediate nodes*. The idea is to use multiple routes for streaming across sites, as depicted in Figure 3. We build this strategy on two observations: the virtual inter data-center routes do not map to identical physical paths and the latency of intra-site communication is low (less than 10%) compared to inter site communication [45]. These observations allow to aggregate additional bandwidth by sending data from the sender nodes to intermediate nodes within the same deployment (i.e. belonging to the same application space), which will then forward it towards the destination. With this approach, the system aggregates extra physical bandwidth from the different paths and routes established when connecting different nodes across the ISP infrastructures. This generic method of aggregating inter-site bandwidth is further discussed in [45], in the context of bulk multi-hop data transfers across multiple cloud data-centers. To integrate this approach within our model, we extend the set of parameters that are used to characterize the stream context with the number of *channels*.

Independent of the number of routes used for streaming, the value of the throughput function has to be known at runtime. In order to limit the number of network samples that need to be performed to obtain it, and to reduce the monitoring intrusiveness, we approximate this function. In Figure 4 a) we present measurements of the throughput for a number of routes between North Central US and North Europe Azure data-centers. In Figure 4 b) we normalize

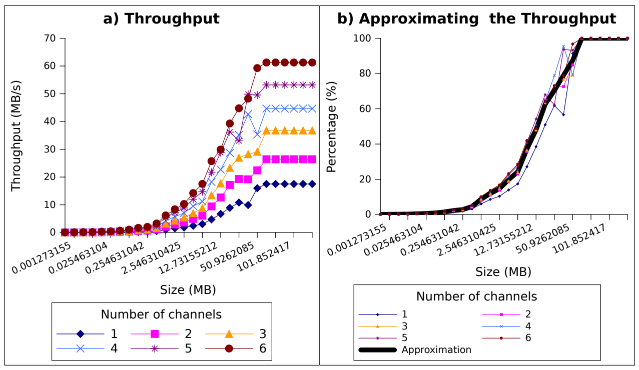

Figure 4: a) The value of the throughput with respect to the number of routes across cloud Small VMs for increasing size of the data chunk. b) Approximating with one pattern the cloud throughput, independent of the number of routes. The approximation is built from the averages of the normalized throughput functions for each number of routes.

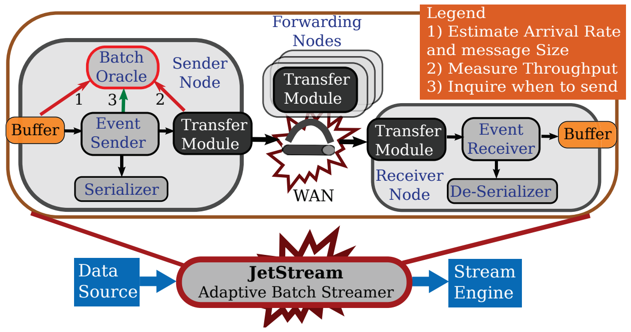

Figure 5: The architecture and the usage setup of the adaptive batch streamer

these values (i.e., % of the corresponding stable throughput) and approximate them using a polynomial function, which we determined empirically that it gives the most accurate approximation. Hence such a model can in fact represent the throughput pattern, with an error introduced by the cloud variability of less than 15%. Using this approximation, the entire function can be extrapolated by measuring only the asymptotic stable throughput. This will be used as the amplitude which multiplied with the normalized estimation will give the throughput for any size.

The downside of using multiple routes for sending batches is that the ordering guarantees offered by the communication protocol for one link do not hold anymore. This translates into batches arriving slightly out of order due to changing conditions on the communication links (e.g., packet drops, congestions etc.). It is mandatory to maintain the integrity of the communication and avoid dropping data just because another link was faster so that batches need to be reordered at the destination. We achieve this by buffering the batches at the destination until their turn to be delivered to the streaming engine arrives. This introduces a new component for the latency: *latency for reordering*. It can be modeled using the Poisson distribution ($\text{Poisson}(k,\lambda) = \frac{\lambda^k \times e^{-\lambda}}{k!}$) to estimate the probability of having k number of batches arriving before the expected batch, which fixes λ to 1, as we consider as reference the time to transfer one batch . This probability can then be correlated with a penalty assigned to each unordered batch, given by its latency. Finally, summing these probabilities over the number of channels and normalizing based on the events will give an estimation of the worst case expected latency: $\text{Latency}_{\text{reordering}} = \frac{\sum_{i=2}^{\text{channels}} \sum_{j}^{L} \text{Poisson}(j,1) \times j \times \text{Latency}_{\times batch}}{\times batch_{\times size} \times L}$. Here, L gives the maximum number of batches (e.g., 10) regarded as potentially arriving before the reference one through a channel, giving the upper limit for iterating the k Poisson parameter.

3.4 Adaptive cloud batching

In Algorithm 1 we wrap up everything and present the decision mechanism in JetStream for selecting the number of events to batch and the number of channels to use. The algorithm estimates the average latency per event for a range of batch sizes and channels, retaining the best one. As an optimization, a simulating annealing technique can be used to perform the search faster, by probing the space with large steps and performing exhaustive searches only around local optima. Depending on the magnitude of the optimal batch size, the maximal end-to-end event latency introduced by batching can be unsatisfactory for a user, as it might violate application constraints, even if the system operates at optimal transfer rates. Hence, the users can set a maximum acceptable delay for an event, which will be converted in a maximum size for the batch (Line 3). As for the number of channels selection, we estimate how many batches can be formed while one is being transferred (Lines 6-8).

Algorithm 1 The selection of the optimal batch size and the number of channels to be used

```
 1: procedure BatchAndChannelsSelection
 2:     getMonitoredContextParameters()
 3:     ESTIMATE MaxBatched from [MaxTimeConstraint]
 4:     while channels < MaxNodesConsidered do
 5:         while $\text{batch}_{\text{size}}$ < MaxBatched do
 6:             ESTIMATE $\text{latency}_{\text{batching}}$ from [$\text{Rate}_{\text{Acquisition}}, \text{batch}_{\text{size}}$]
 7:             ESTIMATE $\text{latency}_{\text{encoding}}$ from [$\text{overheads}, \text{batch}_{\text{sizeMB}}$]
                    ▷ Estimate the transfer latency for 1 channel
 8:             ESTIMATE $\text{latency}_{\text{transfer1}}$ from [$\text{batch}_{\text{sizeMB}}, \text{thrRef}, \text{1channel}$]
 9:             COMPUTE $\text{Ratio}_{CPU}$ from [$\text{latency}_{\text{encoding}}, \text{latency}_{\text{batching}}, VM\_\text{Cores}$]
                    ▷ Prevents idle channels
10:             if $\text{Ratio}_{CPU} * \text{latency}_{\text{batching}} \times \text{channels} < \text{latency}_{\text{transfer1}} + \text{latency}_{\text{encoding}}$ then
11:                 ESTIMATE $\text{latency}_{\text{decoding}}$ from [$\text{overheads}, \text{batch}_{\text{sizeMB}}$]
12:                 ESTIMATE $\text{latency}_{\text{transfer}}$ from [$\text{batch}_{\text{sizeMB}}, \text{thrRef}, \text{channels}$]
13:                 ESTIMATE $\text{latency}_{\text{reordering}}$ from [$\text{channels}, \text{latency}_{\text{transfer}}$]
14:                 $\text{latency}_{\text{perEvent}} = \sum \text{latency}_{*}$
15:                 if $\text{latency}_{\text{perEvent}} < \text{bestLatency}$ then
16:                     UPDATE [bestLatency, Obatch, Ochannels]
17:                 end if
18:             end if
19:         end while
20:     end while
21: end procedure
```

Once the transfer of a batch is completed, adding additional channels to distribute the batches become useless. The condition on Line 10 prevents the system from creating idle routes. Finally, the CPU usage needs to be also integrated in the decision process. Sending frequent small batches will increase the CPU consumption and artificially decrease the overall performance of the cloud node. We therefore assign a penalty based on the ratio between the time to form a batch and the time used by the CPU to encode it, according to the formula $\text{Ratio}_{CPU} = \frac{\text{latency}_{\text{encoding}}}{(\text{latency}_{\text{batching}} + \text{latency}_{\text{encoding}}) \times VM\_Cores}$. To sum up, JetStream collects a set of context parameters (Line 2) and uses them to estimate the latency components according to the formulas presented above. Based on these estimations, it selects the optimal batch size and the number of channels for streaming.

4. SYSTEM OVERVIEW

We designed JetStream as a proof of concept of the previous approach. Its conceptual schema is presented in Figure 5. The events are fed into the system at the sender side, as soon as they are produced, and they are then delivered to the stream processing engine at the destination. The prototype which implements this architecture was developed in

C# using the .NET 4.5 framework. The distributed modules and their functionality are described bellow:

The Buffer is used both as an input and output endpoint for JetStream. The sender application or the event source adds the events to be transferred, as they are produced, while the receiver application (i.e., the stream processing engine) *pops* (synchronously) the events or it is notified (asynchronously) when they have been transferred. The module relies on 2 queues for synchronization purposes in a producer-consumer fashion. The Buffer is also in charge of monitoring the input stream in order to assert in real time the *acquisition rate* of the events and their *sizes*.

The Batch Oracle stays at the core of JetStream, as it enforces the environment-aware decision mechanism for adaptively selecting the batch size and the amount of channels to use. It implements Algorithm 1 and collects the monitoring information from the *Buffer* and the *Transfer Module*. It further controls the monitoring intrusiveness by adjusting the frequency of the monitor samples according to the observed variability.

The Transfer Module performs multi-route streaming. On the intermediate nodes, its role is to forward the packets towards the destination. Currently, the modules is agnostic of any streaming scenario and can integrate any transfer protocol. It provides several implementations on top of TCP: synchronous and asynchronous, single- or multi-threaded. It is also in charge of probing the network and measuring the throughput and its variability. The batches are sent in a round-robin manner across the channels, balancing the load. As future work, the transfer performance can be improved by integrating additional transfer techniques between which the system can automatically switch based on the streaming pattern and the context information.

The Event Sender coordinates the event transfers by managing the interaction between modules. It queries the *Batch Oracle* about when to start the transfer of the batch. Next, it setups the batch by getting the events from the *Buffer* and adding the metadata (e.g., batch ID, streams IDs etc.). The batch is then serialized by the *Serialization* module and the data transferred across data-centers by the *Transfer Module*.

The Event Receiver is the counterpart of *Event Sender* module. The arriving batches are de-serialized, reordered and delivered to the application as a stream of events, in a transparent fashion for the stream processing engine. The module issues acknowledgements to the sender or makes requests for re-sending lost or delayed batches. Alternatively, based on users' policies, it can decide to drop late batches, supporting the progress of the stream processing despite potential cloud-related failures. Users configure when such actions are performed by means of waiting times, number of batches, markers or current time increments (CTI).

Serialization/De-Serialization has the role of converting the batch to raw data, which can be afterwards sent over the network. We integrate in our prototype several libraries: Binary (native), JSON (scientific) or Avro (Microsoft HDInsight), but others modules can be easily integrated. This module can host additional functionalities (e.g., data compression or deduplication) in the future.

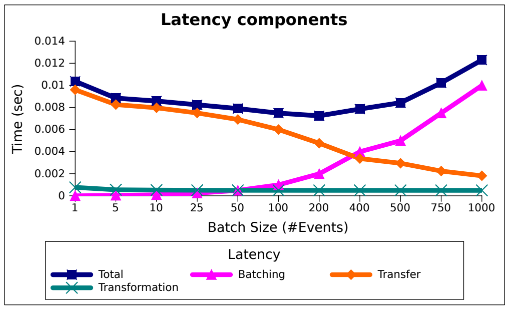

Figure 6: The latency components per event with respect to the batch size for a single streaming channel, for inter-sites event transfers

5. EVALUATION

The goal of the experimental evaluation presented in this section is to validate the JetStream system in a real cloud setup and discuss the main aspects that impact on its performance.

5.1 Experimental setup

The experiments were run in the Microsoft's Windows Azure cloud in the North-Central US and the North Europe data-centers. We chose these sites in order to create a geographical-distributed setup which permits to analyze the ability of the system to adapt to changing factors generated both by the cloud itself (i.e., multi-tenancy, virtualization) and the wide area communication across multiple ISPs. All the experiments were run with Small Web Role VMs having 1 CPU, 1.75 GB of memory, 225 GB local storage and 100 Mbps bandwidth. When using multi-route streaming, up to 5 additional nodes were used within the sender deployment, implementing the transfer scheme discussed in Section 3.3. Each experiment sends between 100,000 and 3.5 million events, which, given the size of an event, translates into a total amount of data ranging from tens of MBs to 3.5 GB. Each sample is computed as the average of at least ten independent runs of the experiment performed at various moments of the day (morning, afternoon and night).

The performance metrics considered are the *transfer rate* and the *average latency of an event*. The transfer rate is computed as the ratio between a number of events and the time it takes to transfer them. More specifically, we measured, at the destination side, the time to transfer a fixed set of events. For the average latency of an event, we measured the number of events in the sender buffer, the transfer time and reported the normalized average per event based on the latency formulas described in Section 3.2.

The evaluation is performed using a synthetic benchmark and a real-life application. We created a synthetic benchmark in order to have full control over the sizes and the generation rate of the events. The generation rates are varied between hundred of events per second to tens of thousands of events per second, as indicated by the scale of the transfer rates. Such a predictable and configurable setup allows us to analyze and to understand the results better when test-

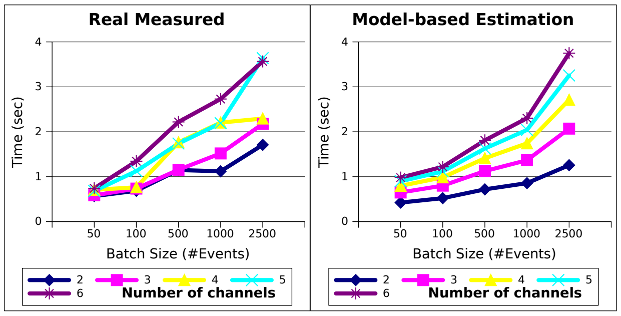

Figure 7: Comparing the estimation of the worst-case latency (i.e., the penalty) of unordered batches with actual measurements, while increasing the number of channels used for the transfers

ing the behavior of JetStream in several scenarios. Finally, we use the events collected by the MonALISA monitoring systems from the Alice LHC experiment to assert the gains brought by our approach in a real setup.

5.2 Accuracy

We depict in Figure 6 the total latency and its components, per event, with respect to the number of batched events. Selecting the optimal size of the batch comes down to finding the value corresponding to the minimal latency (e.g. ~ 200 for the scenario illustrated in Figure 6). The search for the batch size that minimizes the latency per event is at the core of the JetStream algorithm presented in Section 3.4. The selection of the optimal value from this type of representation for the latency is computed from the estimations about the environment (e.g. network throughput, CPU), which may not be exact. Indeed, the cloud variability can lead to deviations around the optimal value in the selection process of the amount of events to batch. Considering that the performance around the optimum is rather flat (as seen in Figure 6), JetStream delivers more than 95% of the optimal performance even in the case of big and unrealistic shifts from the optimum batch size (e.g. selecting a batch size of 150 or 250 instead of 200 in Figure 6 will decrease the performance with 3%). The solution for further increasing the accuracy, when selecting the batch size, is to monitor the cloud more frequently. These observations show that accuracy can be traded for monitoring intrusiveness at the level of cloud resources (e.g. network, memory and CPU).

In order to validate the penalty model proposed for the latency of reordering when using multiple routes, we compare the estimations of our approach with actual measurements. The results are presented in Figure 7. The delay for unordered batches was measured as the time between the moment when an out of order batch (not the one next in sequence) arrives, and the moment when the actual expected one arrives. The experiment shown in Figure 7 presents the average of this maximum unordered delay over tens of independent trials in which a fixed amount of events (i.e. 1 million events of size 2.5 KB) is transferred. We notice that the proposed model gives a good estimation of this delay, which can be used as a penalty latency for multi route streaming. The accuracy in this case is 90%–95%, because the penalty model for multi-routing does not integrate the variability of the cloud. Yet, such errors can be tolerated as they are not determinant when selecting the number of channels to use.

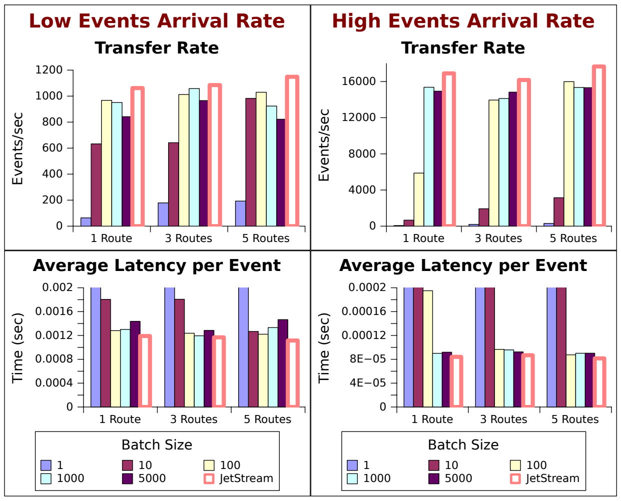

Figure 8: Comparing the performance (transfer rate - top and average latency per event - bottom) of individual event streaming and static batches with the adaptive batch of JetStream for different acquisition rates, while keeping the size of an event fixed (224 bytes). The latency and transfer rates for individual event streaming (i.e., batch of size 1) are between 50 to 250 times worse than the ones of JetStream

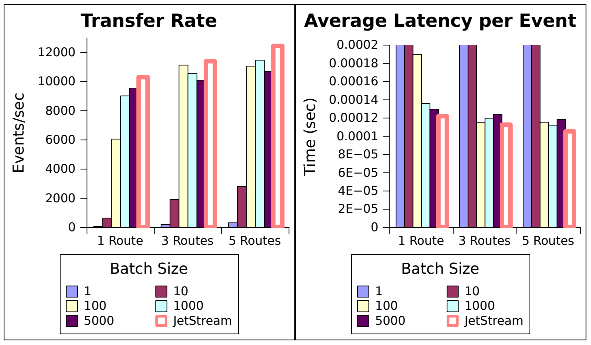

Figure 9: The performance (transfer rate – left and average latency per event – right) of individual event streaming, static batches and JetStream for events of size 800 bytes. The latency and transfer rates for individual event streaming (i.e. batch of size 1) are from 40 up to 150 times worse than JetStream

5.3 Individual vs. batch event transfers

The goal of this set of experiments is to analyze the performance of individual event streaming compared to batch-based streaming, with static and adaptive (i.e., JetStream) sizes. In the static case, the number of events to be batched is fixed a priori, with a batch of size 1 representing independent event streaming. These setups are compared to JetStream, which adapts the batch size to the context at runtime, for varying event sizes or generation rates.

The experiments presented in Figure 8 use an event of size 224 bytes and evaluate the transfer strategies considering low (left) and high (right) event generation rates. The experiments were repeated for different number of routes for streaming: 1, 3 and 5, and measure the transfer rates (top) and average latency per event (bottom). The first observation is that batch-based transfers clearly outperform individual event transfers for all the configurations considered. The results confirm the high overhead and the low throughput with small sizes for the transfers between datacenters. Grouping the events increases the transfer rate tens to hundred of times (up to 250 times for JetStream) while decreasing the average latency per event. Two aspects determine the performance: the size of the batch with respect to the stream context and the performance changes of the cloud. Static batches cannot handle these variations, making certain batch sizes good in one context and bad in others (e.g. batches of size 10 give poor performance for 1 route and high event acquisition rate and good performance for 5 routes and low acquisition rates). Selecting the correct size at runtime brings an additional gain between 25% and 50% over static batch configurations. We repeated the same set of experiments for bigger event sizes, all showing the same behavior. Due to space constraints, we only illustrate the transfer rate and average event latency for an event size of 800 bytes in Figure 9.

5.4 Adapting to the context changes

The data acquisition process in a streaming scenario is not necessarily uniform. Fluctuations in the event rates of an application running in the cloud can appear due to the nature of the data source, the virtualized infrastructure or the cloud performance variability [16]. To analyze the behavior of JetStream in such scenarios, we performed an experiment in which the event generation rate randomly changes in time. For the sake of understanding, we present in Figure 10 a snapshot of the evolution of the transfer rate in which we use fine grain intervals (10 seconds) containing substantial rate changes. JetStream is able to handle these fluctuations by appropriately adapting the batch size. In contrast, static batch transfers either are introducing huge latency from waiting for too many events, especially when the event acquisition rate is low (e.g., batches of size 1000 or 5000 at time moment 9) or are falling behind the acquisition rate which leads to increasing amount of memory used to buffer the untransferred events (e.g., batch of size 100 at moment 5). Reacting fast to such changes is crucial for delivering high performance in the context of clouds; this is also the reason why we chose to consider such sudden rates changes. The number of used resources (e.g. the extra nodes which enable multiple route streaming) is reduced by 30% when taking into account the stream context. These resource savings are justified by the fact that low event acquisition rates do not require multiple route streaming as higher ones do. Additionally, as shown in [23], the fluctuations in the application loads have certain patterns across the week days. Hence, in long running applications, our approach will make substantial savings by detecting these daily or hourly trends and using such knowledge to scale up/down the number of additional nodes used for transfers.

5.5 Benefits of multiple route streaming

Figure 11 shows the gain obtained in transfer rate with respect to the number of routes used for streaming, for JetStream and for a static batch of a relatively small size (i.e. 100 events). Multiple route streaming pays off when increasing the amount of data sent for both strategies (i.e. static and adaptive). In fact, by aggregating extra bandwidth from the intermediate nodes, we are able to decrease the impact of the overhead (batch metadata, communication headers, serialization headers etc.) on smaller batches. More pre-

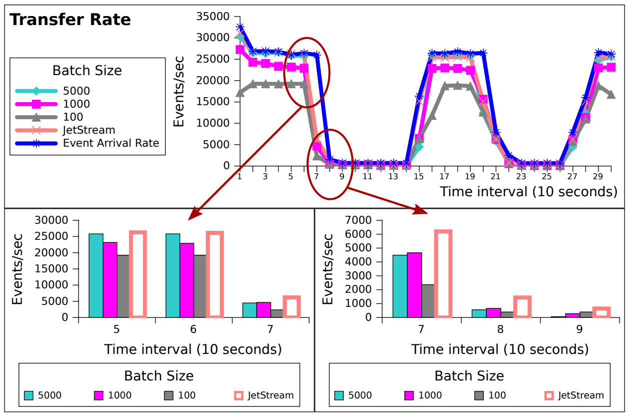

Figure 10: The evolution of the transfer rate in time for variable event rates with JetStream and static batches transfer strategies

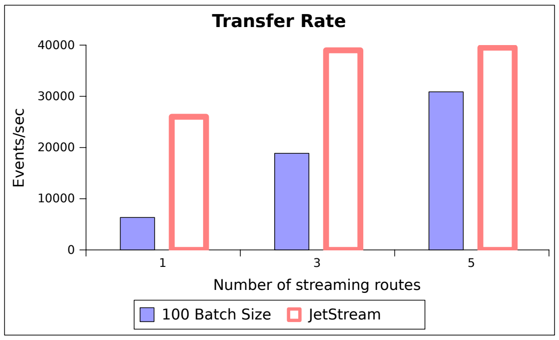

Figure 11: The transfer rate for an increasing number of routes used for streaming

cisely, a larger bandwidth allows to send more data, and implicitly, the additional data carried with each batch does not throttle the network anymore. This brings the transfer rate of smaller, and consequently more frequent, batches closer to the maximum potential event throughput. It is the case of the static batch of size 100 on Figure 11, delivering a throughput close to JetStream for higher number of routes. With higher throughput and a lower overhead, the optimal batch size can be decreased. When selecting it, JetStream is able to decrease the end to end latency. We conclude that sustaining high transfer rates under fixed time constraints is possible by imposing upper bounds for the batch sizes, which enables JetStream to integrate users' time constraints for maximum delay. As discussed in Algorithm 1, it does this by considering a maximum limit on the batch.

5.6 Experimenting with ALICE: a real-life High Energy Physics application

In a second phase, our goal was to assess the impact of JetStream in a real-life application. We opted for ALICE (A Large Ion Collider Experiment) [1], one of four LHC (Large Hadron Collider) experiments at CERN (European Organization for Nuclear Research), as its scale, volume and geographical distribution of data require appropriate tools for efficient processing. Indeed, the ALICE collaboration, consisting of more than 1,000 members from 29 countries and 86 institutes, is strongly dependent on a distributed computing environment to perform its physics program. The experiment collects data at a rate of up to four petabytes per year, produce more than 10^9 data files per year, and require tens of thousands of CPUs to process and analyze them. The CPU and storage capacities are distributed over more than 80 computing centers worldwide. These resources are heterogeneous in all aspects, from CPU model and count to operating system and batch queuing software.

Our focus, in these series of experiments, is on the monitoring information collected in real-time about all ALICE resources. We used the MonALISA [33] service to instrument the huge amount of monitoring data issued from this experiment. More than 350 MonALISA services are running at sites around the world, collecting information about ALICE computing facilities, local and wide area network traffic, and the state and progress of the many thousands of concurrently running jobs. This yields more than 1.1 million parameters published in MonALISA, each with an update frequency of one minute. Using ALICE-specific filters, these raw parameters are aggregated to produce about 35,000 system-overview parameters in real time. These overviews are usually sufficient to identify problems or to take global actions in the system and they are the fuel for the JetStream event streaming platform. The MonALISA framework and its high frequency updates for large volumes of monitoring data matched closely with JetStream's architecture so the pieces fit naturally together, but it also provided us a major opportunity to fullfill the system's initial goal: using the (monitoring) data to take automated decisions in real time (in this context, improve the observed system).

Based on the monitoring data collected and stored by MonALISA as of December 2013, we have replayed a sequence of 1.1 million events considering their creation times at the rate they were generated by Alice. The measurements were performed using 2 additional VMs as intermediate nodes at the sender side (i.e. 3 streaming routes). Initially, the experimental setup considered 5 streaming routes. However, during the transfer of the data using JetStream, we observed that maximum 3 such routes were used, as the system determined that the performance cannot be increased beyond this point. The same number of nodes is recommended if we query the system based on the stream context (i.e. event size and acquisition rate). The accuracy of this decision was in fact validated as our adaptive approach was obtaining the same transfer performance using 3 nodes as the static batch configurations which were using 5. Furthermore, the static configurations also obtained the same transfer performances when switched to 3 nodes, showing that indeed this streaming context was not requiring more than 3 streaming routes. This observation shows the importance of an environment-aware adaptive approach, not subject to arbitrary human configuration choices.

Figure 12 displays the total latency of the events at the sender (Figure 12 a) and the transfer rate (Figure 12 b) when comparing JetStream with static configurations for varying batch sizes. The transfer performances of static batches with more than 100 events are similar with JetStream. Considering that the generation rate of the events varies from low to high, sub-optimal batch sizes will in fact lead to an accumulation of the events in the sender queue during the peak rates. These buffered events will artificially increase the performance, at the expense of extra memory, during the periods when the acquisition rate of events is low. All in all, this behavior will produce a stable transfer performance over a wide range of static batch sizes, as it can be observed in Figure 12 b; but on the other hand, it will increase the latency of the events as depicted in Figure 12 a. As our approach selects the appropriate batch size at each moment, it consequently reduces the amount of events waiting in the sender queue and decreases the overall latency of the events. Compared to the static batch strategies providing constant transfer performance, the latency of JetStream is decreased from 2.2 (100-event batches) down to 17 times (10000-event batches).

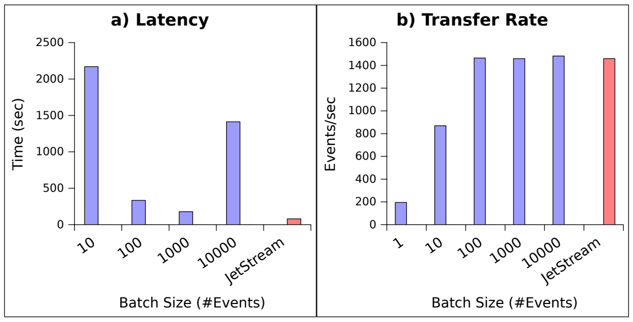

Figure 12: The total latency (a) and the transfer rate (b) measured when transferring 1.1 million events from MonALISA monitoring the Alice experiment. The latency for independent event transfer (i.e. batch of size 1) in a) is not represented because it would modify the scale greatly due its magnitude, having a value of 30900 seconds as opposed to 80 seconds for JetStream

6. RELATED WORK

The existing works can be grouped into 5 categories based on their application domains and features. We discuss them in this section.

Systems for data transfers.

Several solutions have been proposed in the last years for managing data transfers at large scale. StorkCloud [30] integrates multi-protocol transfers in order to optimize the end-to-end throughput based on a set of parameters and policies. It adapts the parallel transfers based on the cluster link capacity, disk rate and CPU capacity, using the algorithm proposed in [47]. The communication between StorkCloud components is done using textual data representation. Similarly, our system optimizes the streaming between data-centers by modeling a set of parameters. However, JetStream remains transparent to the data format and limits the annotation of events or batches with metadata, reducing the overhead of the transfer, which becomes crucial in the context of continuous streams of data. Other systems, like NetSticher [32], target bulk transfers between data-centers, as we do. The authors exploit the day/night pattern peaks of usage of a data-center and leverage the remaining unutilized bandwidth. Though NetSticher is useful for backups and checkpoints, it does not work for real-time systems. GlobusOnline [3] is another system which provides data transfers over WAN, targeting data sharing between different infrastructures (e.g. grids, institution infrastructures, community clouds). Unlike our system which handles streams of events with a light resource fingerprint, GlobusOnline manages only file transfers and has substantial resource requirements. Other recent efforts for enabling parallel data transfers lead to the development of the multi-path

TCP standard [39]. We share common goals with this initiative: aggregating more bandwidth between data-centers and exploiting more links for communication. However, despite the interesting solutions provided at the lower levels of the communication stack for congestion control, robustness, fairness and packet handling, multi-path TCP is not currently available today on the cloud infrastructures. It will need to be set in place by the cloud providers, not by users.

Video streaming.

The work in this area aims to improve the end-to-end user experience by considering mostly video specific optimizations: packet-level correction codes, recovery packets, differentiated redundancy and correction codes based on the frame type [25]. Additionally, such solutions integrate low-level infrastructure optimizations based on network type, cluster knowledge or cross-layer architectures [31, 37]. None of these techniques directly apply to our scenario, as they are strongly dependent on the video format and not adequate for generic cloud processing. On the other hand, ideas like using coordinated or uncoordinated multi-paths to solve congestion, as discussed in [28], are analogous to our approach of using intermediate nodes. However, even the systems which consider the use of TCP for video streaming instead of the traditional UDP, like [46], have some key differences with JetStream. While our goal is to aggregate more bandwidth from the routes between data-centers and to maximize the usage of the network, thereby the extra routes in [46] are used just as a mean to have a wider choice when selecting the fastest path towards the client. Also, the size of the packets is fixed in video streaming, while in our case it adapts to the stream context.

Streaming in peer-to-peer systems.

The peer-to-peer-based systems can be divided in two categories based on how peers that forward the events organize themselves in an overlay network [44]: some use DHT overlays [22, 41] and others group the subscribers in interest groups based on event semantics [42, 43]. While the performance of the former is highly sensitive to the organization of the peers, the latter can improve the performance by sharing common events within the interest group. Further optimizations can be achieved by discovering the network topology which is then matched to the event traffic between the subscribers [44]. This aims to improve the network usage by identifying whether independent overlay paths correspond to common physical paths and by allowing deduplication of events. We share the idea of intermediate nodes forwarding events towards the destination. However, in our case, the virtual topology between the sender, intermediate nodes and destination is fixed and known from the beginning. Moreover, in the context of clouds it is not possible to consider information about physical paths, as all access and knowledge is limited to the virtual space. Unlike these research efforts, our approach focuses on the acquisition of data from remote sources and on the communication between instances of the applications from different cloud data-centers, rather than disseminating information among subscribers. A system close to our work is Stormy[35], which implements concepts from P2P stream processing in the context of clouds. The system is an elastic distributed processing service that enables running a high number of queries on continuous streams of events. The queries are replicated and applied on all the replicas of the events created across the peer nodes. As opposed to JetStream, Stormy delegates the staging-in of the events to the data source which is expected to push the events in the system. It also handles the stream as individual events both in the acquisition and the replication phases.

Data Stream Management Systems (DSMS).

These systems primarily focus on how queries can be executed and only support data transfers as a side effect, usually based on rudimentary mechanisms (e.g., simple event transfer over HTTP, TCP or UDP) or ignore this completely by delegating it to the data source. D-Streams [48] provides tools for scalable stream processing across clusters, building on the idea of handling small batches which can be processed using MapReduce; an idea also discussed in [36]. Here, the data acquisition is event driven: the system simply collects the events from the source. Comet [23] enables batch processing across streams of data. It is built on top of Dryad[26] and its management relies on its channel concept to transfer a finite set of items over shared memory, TCP pipes or files. However, it is designed to work within the same cluster and does not address the issues of sending continuous stream of events between data-centers. In [13], the authors propose a store manager for streams, which exploits access patterns. The streams are cached in memory or disks and shared between the producers and the consumers of events; there is no support for transfers. RIP [10] is another example of DSMS which scales the query processing by partitioning and event distribution. Middleware solutions like System S from IBM [18] were proposed for single cluster processing, with processing elements connected via typed streams. The ElasticStream system [27] migrates this solution to Amazon EC2 taking into consideration cloud-specific issues like SLA, VMs management and the economic aspects of performance. In [14], the authors provide an accurate cost and latency estimation for complex event processing operators, validated with the Microsoft StreamInsight [4]. Other works in the area of cloud-based streaming, like Sphere[21], propose a GPU-like processing on top of a high performance infrastructure. In [38], the authors propose to use web services for the creation of processing pipelines with data passed via the web service endpoints. Other systems like Aurora and Medusa [7] consider exploiting the geographically distributed nature of stream data sources. However, the systems have a series of limitations despite their strong requirements from the underlying infrastructure (e.g., naming schema, message routing): Aurora runs on a single node and Medusa has a single administrator entity. All these cloud-based solutions focus on query processing with no specific improvements or solutions for the cloud streaming itself and, furthermore, they do not support multi-site applications.

Adaptivity for streaming.

Several solutions aim to provide a dynamic behavior at different phases of stream processing. SPC [8] is a distributed platform which supports the execution of many queries on multiple streams. It proposes a new programming model and the corresponding system architecture. The main features are high expressivity and dynamic binding of input with output ports at runtime, based on user format description. From the data management point of view, it provides a hierarchical communication model, adaptively shifting between pointers, shared memory and network transport protocols (TCP, UDP, multicast). Nevertheless, the stream-based optimizations are applicable only at the level of multi-core nodes. Another approach consists in partitioning the input streams in an adaptive way considering information about the stream behavior, queries, load and external metadata [11]. Despite the fact that this solution targets the management of input streams, it has no support for the efficient transmission of data. Other solutions [15] focus on improving performance over limited resources (e.g. disk, memory). This is achieved by moving from row to column-oriented processing and by exploiting the event time information. From the data management point of view, the events are assumed to be transferred event-by-event by the data source. The problem of optimizing the distribution of data in the presence of constrained resources is also discussed in [19]. Their approach identifies delivery graphs with minimal bandwidth consumption on nodes which act both as consumers and sources, while applying filter operations on incoming streams. The benefits of adaptivity in the context of streaming can be also exploited at application level: in [34] the authors show the advantage of adapting the size of the window in the detection of heart attacks. JetStream complements these works by focusing on the transfers and adapting to the stream context.

7. CONCLUSIONS

In this paper, we have proposed a novel approach for high performance streaming across cloud data-centers by adapting the batch size to the transfer context and to the cloud environment. The batching decision is taken at runtime by modeling and minimizing the average latency per event with respect to a set of parameters which characterize the context. To tackle the issue of low bandwidth between data-centers, we propose a multi-route schema. This method uses additional cloud nodes from the user resource pool to aggregate extra bandwidth and to enable multi route streaming.

As a proof of concept, we designed JetStream, which was validated across two data-centers, from Europe and US, within the Azure cloud, using synthetic benchmarks, and monitoring data collected with MonALISA from the Alice experiment at CERN. JetStream increases transfer rates up to 250 times compared to individual event transfers. The adaptive selection of the batch size further increases performance with an additional 25% compared to static batch size configurations. Finally, multi-route streaming triples performance and decreases the end-to-end latency while providing high transfer rates.

Encouraged by these results, we plan to extend the communication module with additional transfer protocols and to automatically switch between them based on the streaming pattern. We are also investigating other transfer strategies for the multi-route streaming across cloud nodes. Finally, we plan to analyze how this approach can be applied for multicast inter site stream traffic.

8. ACKNOWLEDGMENT

The authors would like to thank Costin Grigoras (CERN) for his valuable support and insights on the MonALISA monitoring traces of the ALICE experiment. This work was partially supported by the joint INRIA - Microsoft Research Center in the framework of the ZCloudFlow project.

9. REFERENCES

[1] CERN Alice. `http://alimonitor.cern.ch/map.jsp`.

[2] Cloud Computing and High-Energy Particle Physics: How ATLAS Experiment at CERN Uses Google Compute Engine in the Search for New Physics at LHC. `https://developers.google.com/events/io/sessions/333315382`.

[3] GlobusOnline. `https://www.globus.org`.

[4] Microsoft StreamInsight. `http://technet.microsoft.com/en-us/library/ee362541.aspx`.

[5] Ocean Observatory Initiative. `http://oceanobservatories.org/`.

[6] Windows Azure. `http://windows.azure.com`.

[7] D. J. 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 VLDB Journal*, 12(2):120–139, Aug. 2003.

[8] 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 *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, DMSSP '06, pages 27–37, New York, NY, USA, 2006. ACM.

[9] M. Arrott, A. Clemesha, C. Farcas, E. Farcas, M. Meisinger, D. Raymer, D. LaBissoniere, and K. Keahey. Cloud provisioning environment: Prototype architecture and technologies. *Ocean Observatories Initiative Kick Off Meeting*, September 2009.

[10] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 3–14, New York, NY, USA, 2013.

[11] C. Balkesen, N. Tatbul, and M. T. Özsu. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 15–26, New York, NY, USA, 2013. ACM.

[12] A. Baptista, B. Howe, J. Freire, D. Maier, and C. T. Silva. Scientific exploration in the era of ocean observatories. *Computing in Science and Engg.*, 10(3):53–58, May 2008.

[13] I. Botan, G. Alonso, P. M. Fischer, D. Kossmann, and N. Tatbul. Flexible and scalable storage management for data-intensive stream processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 934–945, New York, NY, USA, 2009. ACM.

[14] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos. Accurate latency estimation in a distributed event processing system. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 255–266, Washington, DC, USA, 2011. IEEE Computer Society.

[15] F. Farag, M. Hammad, and R. Alhajj. Adaptive query processing in data stream management systems under limited memory resources. In *Proceedings of the 3rd Workshop on Ph.D. Students in Information and Knowledge Management*, PIKM '10, pages 9–16, 2010.

[16] I. Foster, A. Chervenak, D. Gunter, K. Keahey, R. Madduri, and R. Kettimuthu. Enabling PETASCALE Data Movement and Analysis. *Scidac Review*, Winter 2009.

[17] Z. Galić, E. Mešković, K. Križanović, and M. Baranović. Oceanus: A spatio-temporal data stream system prototype. In *Proceedings of the Third ACM SIGSPATIAL International Workshop on GeoStreaming*, IWGS '12, pages 109–115, New York, NY, USA, 2012. ACM.

[18] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[19] B. Gedik and L. Liu. Quality-aware dstributed data

delivery for continuous query services. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 419–430, New York, NY, USA, 2006. ACM.

[20] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, June 2003.

[21] Y. Gu and R. L. Grossman. Sector and sphere: The design and implementation of a high performance data cloud. *Philosophical Transactions A Special Issue associated with the UK e-Science All Hands Meeting*, 12.

[22] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[23] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 63–74, New York, NY, USA, 2010. ACM.

[24] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of windows azure. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 367–376, New York, NY, USA, 2010. ACM.

[25] R. Immich, E. Cerqueira, and M. Curado. Adaptive video-aware fec-based mechanism with unequal error protection scheme. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 981–988, New York, NY, USA, 2013. ACM.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

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

[28] P. Key, L. Massoulié, and D. Towsley. Path selection and multipath congestion control. *Commun. ACM*, 54(1):109–116, Jan. 2011.

[29] R. Kienzler, R. Bruggmann, A. Ranganathan, and N. Tatbul. Stream as you go: The case for incremental data access and processing in the cloud. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, ICDEW '12, pages 159–166, Washington, DC, USA, 2012. IEEE Computer Society.

[30] T. Kosar, E. Arslan, B. Ross, and B. Zhang. Storkcloud: Data transfer scheduling and optimization as a service. In *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing*, Science Cloud '13, pages 29–36, 2013.

[31] C. Lal, V. Laxmi, and M. S. Gaur. A rate adaptive and multipath routing protocol to support video streaming in manets. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pages 262–268, New York, NY, USA, 2012.

[32] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with netstitcher. *SIGCOMM Comput. Commun. Rev.*, 41(4):74–85, Aug. 2011.

[33] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, C. Dobre, A. Muraru, A. Costan, M. Dediu, and C. Stratan. Monalisa: An agent based, dynamic service system to monitor, control and optimize distributed systems. *Computer Physics Communications*, 180(12):2472 – 2498, 2009.

[34] M. Lindeberg, V. Goebel, and T. Plagemann. Adaptive sized windows to improve real-time health monitoring: A case study on heart attack prediction. In *Proceedings of the International Conference on Multimedia Information Retrieval*, MIR '10, pages 459–468, 2010.

[35] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann. Stormy: An elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 55–60, New York, NY, USA, 2012. ACM.

[36] K. G. S. Madsen, L. Su, and Y. Zhou. Grand challenge: Mapreduce-style processing of fast sensor data. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 313–318, New York, NY, USA, 2013. ACM.

[37] S. Mao, X. Cheng, Y. Hou, and H. Sherali. Multiple description video multicast in wireless ad hoc networks. *Mobile Networks and Applications*, 11(1):63–73, 2006.

[38] P. N. Martinaitis, C. J. Patten, and A. L. Wendelborn. Component-based stream processing "in the cloud". In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing*, CBHPC '09, pages 16:1–16:12, New York, NY, USA, 2009. ACM.

[39] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[40] M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Tuning qod in stream processing engines. In *Proceedings of the Twenty-First Australasian Conference on Database Technologies - Volume 104*, ADC '10, pages 103–112, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[41] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, Feb. 2003.

[42] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, and K. Rothermel. Meeting subscriber-defined qos constraints in publish/subscribe systems. *Concurr. Comput. : Pract. Exper.*, 23(17):2140–2153, Dec. 2011.

[43] M. A. Tariq, B. Koldehofe, G. G. Koch, and K. Rothermel. Distributed spectral cluster management: A method for building dynamic publish/subscribe systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 213–224, 2012.

[44] M. A. Tariq, B. Koldehofe, and K. Rothermel. Efficient content-based routing with network topology inference. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 51–62, New York, NY, USA, 2013. ACM.

[45] R. Tudoran, A. Costan, R. Wang, L. Bougé, and G. Antoniu. Bridging data in the clouds: An environment-aware system for geographically distributed data transfers. In *Proceedings of the 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2014)*, CCGRID '14. IEEE Computer Society, 2014.

[46] B. Wang, W. Wei, Z. Guo, and D. Towsley. Multipath live streaming via tcp: Scheme, performance and benefits. *ACM Trans. Multimedia Comput. Commun. Appl.*, 5(3):25:1–25:23, Aug. 2009.

[47] E. Yildirim and T. Kosar. Network-aware end-to-end data throughput optimization. In *Proceedings of the First International Workshop on Network-aware Data Management*, NDM '11, pages 21–30, 2011.

[48] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.