JetStream: Enabling high throughput live event streaming on multi-site clouds

Radu Tudoran, Alexandru Costan, Olivier Nano, Ivo Santos, Hakan Soncu, Gabriel Antoniu

Available online at www.sciencedirect.com

ScienceDirect

Future Generation Computer Systems 00 (2014) 1–27

Future Generation Computer Systems

JetStream: Enabling High Throughput Live Event Streaming on Multi-site Clouds

Radu Tudoran[a,*], Alexandru Costan[b], Olivier Nano[c], Ivo Santos[c], Hakan Soncu[c], Gabriel Antoniu[d]

[a] *IRISA/ENS Rennes*
Campus universitaire de Beaulieu, Rennes, France
[b] *IRISA/INSA Rennes*
Avenue des Buttes de Coësmes, Rennes, France
[c] *Microsoft Research – ATL Europe*
Rablstrasse 26, Munich, Germany
[d] *Inria Rennes - Bretagne Atlantique*
Campus universitaire de Beaulieu, Rennes, France

Abstract

Scientific and commercial applications operate nowadays on tens of cloud datacenters around the globe, following similar patterns: they aggregate monitoring or sensor data, assess the QoS or run global data mining queries based on inter-site event stream processing. Enabling fast data transfers across geographically distributed sites allows such applications to manage the continuous streams of events in real time and quickly react to changes. However, traditional event processing engines often consider data resources as second-class citizens and support access to data only as a side-effect of computation (i.e. they are not concerned by the transfer of events from their source to the processing site). This is an efficient approach as long as the processing is executed in a single cluster where nodes are interconnected by low latency networks. In a distributed environment, consisting of multiple datacenters, with orders of magnitude differences in capabilities and connected by a WAN, this will undoubtedly lead to significant latency and performance variations. This is namely the challenge we address in this paper, by proposing JetStream, a high performance batch-based streaming middleware for efficient transfers of events between cloud datacenters. 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, while at the same time optimizing resource utilization and increasing cost efficiency. The prototype was validated on tens of nodes from US and Europe datacenters of the Windows Azure cloud with synthetic benchmarks and a real-life application monitoring the ALICE experiment at CERN. The results show a 3x increase of the transfer rate using the adaptive multi-route streaming, compared to state of the art solutions.

Keywords: cloud computing, Big Data, multi-site, stream processing

1. Introduction

We are at a stage of scientific research development where vast volumes of scientific data (Big Data, in the order of PetaBytes) captured by new instruments on a 24/7 basis are becoming publicly accessible for the purposes of continued

*Corresponding author.
Email addresses: Radu.Tudoran@inria.fr (Radu Tudoran), Alexandru.Costan@inria.fr (Alexandru Costan)

analysis. Mining these data sets will result in the development of many new theories. This is being acknowldged as a major shift towards a new, fourth paradigm of science [1], a successor of the traditional experimental, theoretical and (more recently) computational approaches to science. The basic idea is that our capacity for collecting scientific data has far outstripped our present capacity to analyze it, and so our focus should be on developing technologies that will *make sense of this "data deluge" in almost real-time*. The scale of cross-laboratory experiments and their data rates make the development of tools a formidable challenge. Clouds are already in place to host the Big Data in all their shapes and sizes, as providers like Microsoft, Amazon, Google, Rackspace have built global infrastructures 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. Exploiting the underlying multi-datacenter cloud infrastructures effectively in the presence of Big Data remains critical.

In the past years, a subclass of Big Data, *fast data* (i.e., high-speed real-time and near-real-time data streams) has also exploded in volume and availability. These specific data, often denoted as *events*, are typically characterised by a small unit size (in the order of kilobytes), but overwhelming collection rates. Examples of such data include sensor data streams, social networks feeds (e.g. 4k tweets per second, 35k Facebook likes and comment per second), stock-market updates. Numerous applications must process vast amounts of fast data collected at increasing rates from multiple sources [2, 3], with minimal latency and high scalability. Examples range from industry and business applications (e.g. transactional analysis, financial tickers, monitoring and maintenance systems for web services) to scientific ones (e.g. large-scale sensor-based systems, simulations, climate monitoring, LHC experiments). As the number and sophistication of such applications grow, a natural question arises: *Can we provide a high performance framework for fast data movements, so that event processing engines can quickly react to the sensed changes, independent of the data source or arrival rate?*

Clearly, acquiring, processing and managing this data efficiently is a non-trivial endeavour, 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 datacenters. 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 infrastructures, as it was the case of the Higgs boson discovery which had to extend the computation to the Google cloud infrastructure [4]. Another scenario is given by *the data sources* which can be physically distributed in wide geographical locations as in the Ocean Observatory Initiative [5, 6], in which the collected events are streamed to Nimbus [7] clouds. Finally, *the nature of the analysis* requires aggregating streams of data from remote application instances for an increasing number of services. Large-scale services like Microsoft's Bing and Office 365 operate on tens of datacenters 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.

In this paper, we address the above challenges by proposing JetStream, a novel approach for streaming events across cloud datacenters. JetStream enables low latency fast data processing, high and stable transfer rates in spite of the inherent cloud performance variability for geographically distributed applications executed on clouds. 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 modelling the latency based on a set of parameters which characterize the streaming in the context of clouds. To further improve performance, we aggregate inter-datacenter bandwidth as we extend our approach to provide multi-route streaming across cloud nodes. Furthermore, we develop 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 key innovation is the co-design of an integrated architecture for event streaming from their source to the processing engine, that addresses challenges of heterogeneity in scale, complexity, and diversity for both workloads (high performance and high throughput) and infrastructure (several orders of magnitude difference in capabilities between datacenters). This type of co-design was attempted before only in HPC environments for compute intensive tasks, or only for very small or for very large datacenters. JetStream was validated on the Windows Azure [8] cloud using synthetic benchmarks and in a real life scenario using the MonALISA [9] monitoring system of the CERN ALICE

experiment [10]. 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. JetStream is thus an efficient, fair and cloud agnostic platform for the movements of fast data (events) in the cloud.

The key contributions of this paper are:

- The design of a multi-site cloud streaming platform, able to adapt the event batch size by modeling the cloud latency and considering the cost of multi-routing (Section 3);
- A protocol that routes data streams through multiple parallel paths, aggregating bandwidth from multiple intermediate nodes (Section 3.2 and 3.6);
- An implementation of these building blocks into a prototype, JetStream, used in production on the Azure cloud (Section 4);
- An extensive evaluation of this prototype across Azure data-centers from different continents (i.e. Europe and America) (Section 6);
- A comprehensive survey of the state-of-the-art in several domains tangent to streaming and data management, that supports the positioning of our contributions with respect to these directions (Section 8).

Relationship to previous work. This paper extends several contributions introduced in a previous paper [11] by putting them into a more global perspective and by showing how each of them contributes to the central goal of filling some major technological gaps in the area of cloud-based multi-site streaming, gaps identified as such by the scientific community. We propose a new unified architecture that couples and extends the previous modules into a uniform fast data streaming service for clouds. Beyond this synthetic work, this paper makes a new step further by generalizing the initial multi-path protocol from a many-to-1 scenario to the global many-to-many pattern, useful for a larger spectrum of scientific applications (e.g. broadcast, multicast support). A new scheme for selecting the intermediate hops that exploit the network parallelism within this protocol is also introduced, inspired by the ant colonies movement in biology. Finally, the related work is surveyed in a more comprehensive manner, adding a focus on the competitor cloud based streaming solutions.

2. The Problem of Streaming across Cloud Data Centers

The need for efficient streaming tools. An extended survey over thousands of commercial jobs and millions of machine hours of computation, presented in [12], 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 I/O, the rest corresponding to the replication of data between query services or to the intermediate data passed between them. This emphasizes that the event processing services, be they distributed, exchange large amounts of data. Additionally, the analysis highlights the sensitivity of the performance of the stream processing to the management and transfer of events. This idea is discussed also in [2], where the authors stress the need for a high performance transfer system for real-time data. A similar conclusion is drawn in [13], where the issues which come from the communication overhead and replication are examined in the context of state-based parallelization. Finally, in [14], the authors emphasize the importance of *data freshness*, which improves the QoS of a stream management system and implicitly the quality of the data (QoD). All these research efforts, which complement our evaluation study, support and motivate the growing need for a high performance system for event streaming.

Although not new, these tasks incur unprecedented complexities, which are not addressed by the existing cloud data handling services. Research efforts on cloud stream processing have so far mainly focused on distributing and parallelizing large and complex streaming applications. These approaches lack mechanisms and policies for dynamically coordinating transfers among different datacenters in order to achieve reasonable QoS levels and optimize the cost-performance. The problem is further complicated by the heterogeneity of multiple cloud datacentres: they incur orders of magnitude differences in capability and capacity (from 100 MW scale datacenters for HPC to 10 kW scale datacenters for Big Data), which difference was not exploited due to difficulty and lack of an energy incentive. Being able to effectively use the underlying computing and network resources has thus become critical for wide-area data management as well as for federated cloud settings.

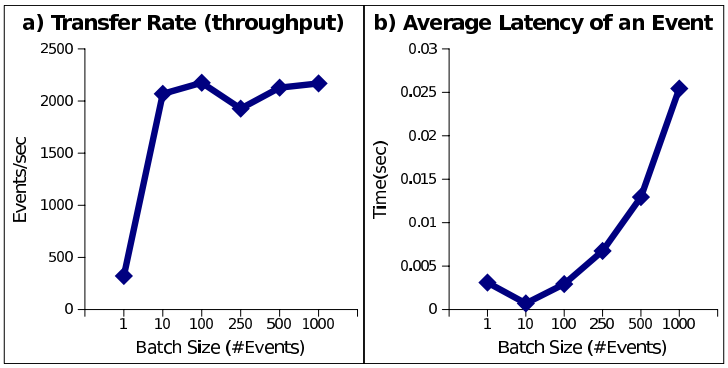

Figure 1. The performance of transferring events in batches between North Europe and North US Azure data-centers. The measurements show the correlation between (a) the transfer rate and (b) the average event latency for increasing batch sizes.

2.1. The challenge of batch streaming

In one of our previous works [15], we performed a performance evaluation study for streaming strategies on clouds. We analyzed two strategies for implementing multi-site cloud streaming scenarios. The first strategy seeks to overlap computation and communication by streaming data directly to the nodes where the computation takes place, in such a way that the rate of data streaming keeps pace with computation; the second strategy relies on first copying data to the cloud and then using it for computation. We evaluate these strategies in the context of a CERN LHC application [16, 17]. The results indicate that current real-time communication in the cloud can significantly interfere in computation and reduce the overall application performance. In fact, deploying state-of-the-art event based transfers, contrary to one's expectation, reduces the stream performance by up to 4 times compared to the second strategy which involves intermediately copying data to storage. The main lesson learned is that achieving high performance for geographically-distributed stream processing requires to minimize the latency while accounting for the resource usage: CPU and bandwidth.

Moving to batch-based streaming. Achieving high performance event streaming requires new cloud-based solutions, since current tools do not provide adequate support. Most of the existing stream processing engines, as the one used by CERN's Atlas applications, only focus on event processing and provide little or no support for efficient event transfers. Others even delegate this functionality to the event source. Today, the typical way to transfer events is individually (i.e., *event by event*) or in small, fixed groups. 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 the selection of the proper batch size (i.e., *how many events to batch?*). 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, the resource usage and on the sensed environment (i.e., the cloud). We 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 environment, dynamically adapt the batch size to the context and enable multi-route streaming across clouds nodes.

2.2. The challenge of multicast streaming

Many streaming scenarios involve several participants. For example the publish/subscriber architectures [18] are being deployed more and more often due to their design capability of supporting large-scale systems. Moreover, many Big Data applications (e.g., virtual observatories, monitoring large web services, scientific workflow etc.) comply to a pattern in which different operations are being applied to a stream of data. Accounting for the topology overlay that exists between the event source and the consumers is equally important for achieving high performance streaming in the cloud. Consequently, the previous question concerning the number of events to be batched extends to how to adapt the number of events in the batch with respect to the streaming topology. This is the main question that

we address in this paper as we focus to generalize the previous point-to-point streaming model introduced in [11] to multicast streaming. Although the primary goal of minimizing latency remains the same, moving from single to multiple destinations introduces additional challenges concerning costs. Replicating data across destinations can incur different charges levels depending on the replication strategy. We recall that the cloud providers charge based on the amount of outbound data from the cloud sites. Hence, finding the right architecture, striking the good balance between cost and performance, and able to dynamically adapt the streaming strategy accordingly is critical for providing high performance cloud streaming.

3. Towards a Multi-Site Cloud Streaming Model

3.1. Initial observations

We start the design of our approach from the following observations concerning streaming across data centers:

Cost of in-site vs. cross-site transfers. Transferring and replicating data within a cloud datacenter is free of charge. Things change for inter-site data exchanges. Cloud providers such as Amazon or Azure charge for the price of outbound data, while keeping inbound data costs to zero. Generally, several cost plan levels are offered based on the amount of outbound data. Hence, cost-wise, it is profitable to perform data replication within the site on the one hand, and to de-duplicate the data that is transferred between sites, on the other hand. Nevertheless, performing such optimizations is at odds with the primary constraint of minimizing latency for real time communication. Hence, different strategies need to be analyzed (and adopted) in order to find the right balance between price and performance.

Latency and Transfer Rate. Inter-site communication is a critical operation not only from the cost point of view but also for perspective. Clouds are composed of several datacenters distributed across geographical regions. This global deployment naturally inherits a high communication latency. Per packet latency can reach tens of milliseconds between data centers located in different areas of a continent to hundreds of milliseconds between datacenters located in different continents. The network bandwidth available between the application instances across datacenters is also typically low. Therefore, the transferring rate (e.g., of the live transferred events) achieved by applications is low and represents a major challenge for real time data.

CPU and I/O. Users rent cloud VMs primary for supporting the application computation logic. But computation depends on the data, and for many application scenarios the two distinct but inter-dependent phases are interleaved. Consequently, two undesirable situation emerge. First, applications are stalled to perform geographically-remote I/O operations and synchronize application instances to progress in the next computation phases (e.g., successive phases of iterative algorithms, synchronizing reduce phase with map phase, MPI checkpoint etc.). Second, as for the study mentioned in Section 2.1, having the I/O phase competing with the computation phase for the CPU can significantly reduce the overall performance. Therefore, separating the communication and providing a fine-grain access of the I/O modules over the compute resources are key milestones.

Low Event Sizes. Being able to handle data in large chunks, in the order of tens of MB or more, improves performance and eases the management process as it minimizes (or renders insignificantly) the impact of the communication overhead (e.g., protocol headers, network congestion, packets resend, etc.). Nevertheless, having such sizing guarantees is rarely the case for live streaming, where data is generated, represented and handled in the form of small tuplets, called events. For example, a whether/climate monitoring application will handle a set of events aggregating as much as tens of small parameters (e.g., temperature, wind speed/direction, humidity, station id, etc.) with sizes of up to KB order. High energy physics simulations as CERN ATLAS [16] have a significantly higher number (i.e., hundreds) of parameters captured in real-time but with sizes usually less than MBs. Finally, monitoring data of large web services or warehouses conform to a similar pattern, aggregating tens of parameters in the size of KB–MB order. Due to the small sizes, it is not possible to apply out-of-the-box the existing cloud data management techniques. Therefore, the main challenge is to provide new and dedicated solutions for such small, live data streams.

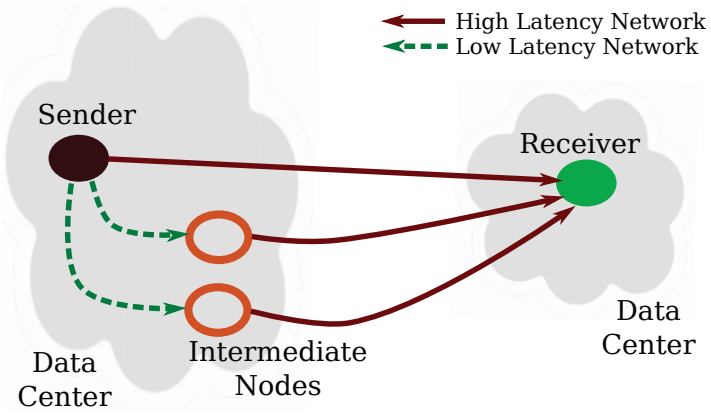

Figure 2. The proposed schema for multi-route streaming across cloud nodes.

3.2. Design principles

We introduce the following design principles to tackle the challenges of cloud streaming in the clouds:

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 of 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.

Cost – performance tradeoffs - Not all applications value performance equally. Therefore, we consider and adopt in our architecture design different strategies that would enable applications to set cost and performance tradeoffs. Hence, we adopt transfer schemes that enable elastic scaling of resources. In this way, critical operations are supported by acquiring more resources while low priority operations are optimized for costs at the expense of lower performance levels.

Multi-route streaming - To tackle the low interconnecting bandwidth between sites, we designed a cloud transfer strategy for harvesting extra bandwidth by using *additional intermediate nodes*. The idea is not to rely on a single communication channel between sites (i.e., the channel between sender and destination) but on multiple routes for streaming, as shown in Figure 2. The strategy is built on 2 observations: the virtual routes between sites created when using extra nodes are not mapped to the same physical paths. Secondly, the latency of intra-site communication is low (less than 10%) compared to inter-site communication [19]. These observations allow to aggregate inter-site bandwidth by sending passing data to destination also through intermediate nodes within the same deployment (i.e. belonging to the same application space). This generic method of aggregating inter-site bandwidth is further discussed in [19], in the context of bulk multi-hop data transfers across multiple cloud data-centers.

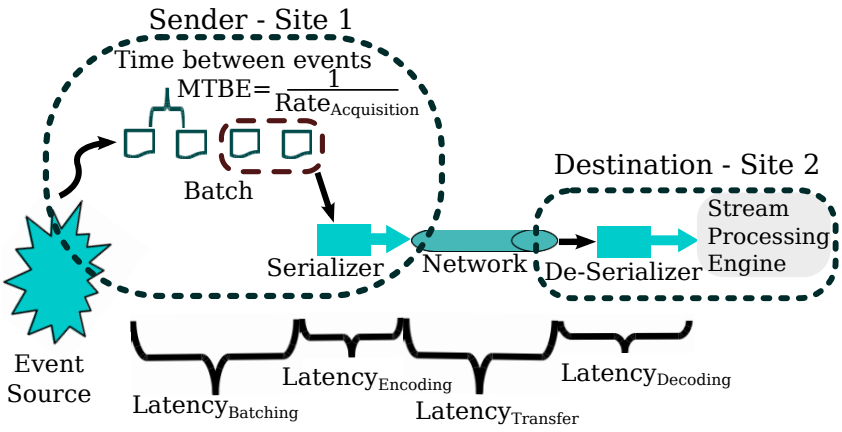

Figure 3. Breaking down the latency to deliver an event from source to stream processing engine across cloud nodes.

3.3. Point-to-Point streaming

To apply the previous design principles and provide a high performance streaming mechanism, we propose a streaming model for the event latency. We express this streaming latency based on a set of cloud parameters which can be monitored at runtime. Such a technique allows to correlate the batch size corresponding to the minimal event latency both to stream context and to environment information. We start by breaking down the latency between the source and the destination of an event in four components, depicted on Figure 3: creating the batch, encoding it (e.g., serializing, compression, etc.), transferring the batch and decoding it. 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/de-serialization technology, the throughput (thr) with the corresponding parallelism of multi-route streaming, expressed by the number of routes *routes*, and the number of events to put in the batch (i.e., batch size - $\text{batch}_{\text{Size}}$). The goal is to determine dynamically the size of the batch, based on the latency model defined based on the other parameters.

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 together. 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 (i.e., position $\times \frac{1}{\text{Rate}_{\text{Acquisition}}}$, we chose to express it as the average latency of an event. This can be computed by averaging the sum of the delays of all events in the batch:

$$\text{Latency}_{\text{batching}} = \frac{\text{batch}_{\text{Size}}}{2} \times \frac{1}{\text{Rate}_{\text{Acquisition}}}$$

Intuitively, this corresponds to the latency of the event in the middle of the sequence.

The transformation latency gathers the times to encode and to decode the batch. This applies to any serialization library/technology. The latency 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 - time for data encoding tDe), while the b constant gives the time to write the metadata (time for header encoding tHe). The latency per event can be expressed as:

$$\text{Latency}_{\text{transformation}} = \frac{tHe + tDe \times \text{batch}_{\text{SizeMB}}}{\text{batch}_{\text{Size}}}$$

which holds both for the encoding and decoding operations. The formula can be used also to express the size of the data after the encoding operations. It only requires to replace the time-related constants with data-related ones (i.e., size of the metadata after encoding and the compression ratio). Moreover, it can be applied to other data pair transformations: compression, deduplication, encryption, etc.

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) — size overhead for transport *sOt* and the encoding technique — size overhead for encoding *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-site throughput - *thr*(Size) is discussed in more detail in the following section. The average latency for transferring an event can then be expressed as:

$$\text{Latency}_{\text{transfer}} = \frac{sOt + sOe + \text{batch}_{\text{SizeMB}}}{thr(\text{Size}_{\text{Total}}, \text{routes})}$$

We define the $thr(\text{Size}_{\text{Total}}, \text{routes})$ function both in terms of the number of routes used for streaming (i.e., the network parallelism used) and the size of the batch. We need to account for the size and not rely only a stable throughput value because of the potential small sizes of the events and batches. Sending small chunks of data will prevent any system to reach the potential throughput of the communication link. In fact, this is one of the issues of streaming and a motivation factor for grouping the events in batches. We model this throughput function by empirically determining a polynomial function that defines the relation between the throughput obtain for a certain size of the data and the stable throughput that can be reached with a certain number of parallel routes. Using such an approximation, the entire function can be extrapolated by measuring only the stable throughput. This stable through is used as the amplitude, which multiplied with the polynomial estimation, will give the throughput for any size.

Batch reordering is the downside of using multiple routes. The ordering guarantees offered by the communication protocol for one route does not hold anymore when batches are sent via independent routes. This translates into batches arriving out of order due to changing conditions on the physical communication routes (e.g., packet drops, congestion, etc.). Nevertheless, it is mandatory to maintain the integrity of the communication and avoid dropping data just because another route was faster. Hence, batches need to be reordered at the destination and the corresponding delay (i.e., *latency for reordering*) needs to be accounted for within the model. The reordering is done by buffering the batches at the destination until their turn to be delivered to the streaming engine arrives. We model the introduced latency by 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. As we take as reference the transfer of the next expected batch, λ parameter becomes 1. This probability can then be correlated with a penalty assigned to each unordered batch. We use as penalty the latency (i.e., $\text{Latency}_{\times batch}$) incurred by having a number of batches (j) arriving out of order. This gives $\text{Poisson}(j, 1) \times j \times \text{Latency}_{\times batch}$, over which we sum in order to account for all potential number of batches arriving out of order. We denote L the maximum number of batches (e.g., 10) regarded as potentially arriving before the reference one through a channel, giving the upper limit for the summation. Finally, we sum these penalties over the number of channels, as each channel can incur its own number of unordered batches, and normalizing based on the events, as our model express everything as latency per event. The final equation that models the unordered batches arriving through all channels is:

$$\text{Latency}_{\text{reordering}} = \frac{\sum_{i=2}^{\text{channels}} \sum_{j}^{L} \text{Poisson}(j, 1) \times j \times \text{Latency}_{\times batch}}{\text{batch}_{\text{size}} \times L}$$

3.4. Beyond Point-to-Point: Multicast streaming

So far we have considered streaming the data from the source, across the data centers, towards one destination. However, in many scenarios, the data stream is required by several applications or machines. Examples of such scenarios range from partitioning the computation in distinct queries, executed across distinct machines, to interconnected processes which manipulate distinct processes of the stream. In what follows, we will denote the number of destinations with the *Destinations* parameter.

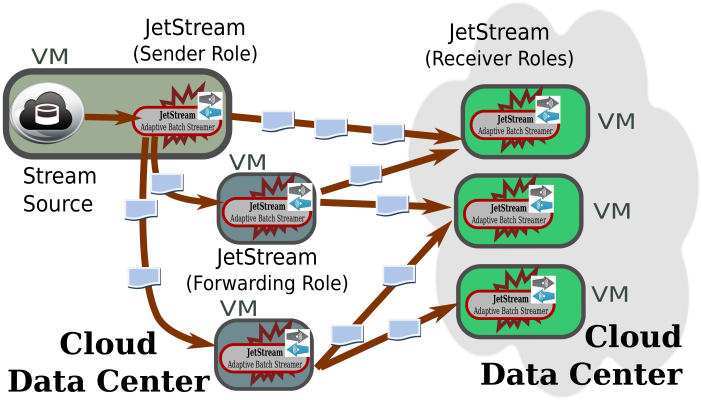

Figure 4. A direct multicast streaming strategy across datacenters, towards all destinations.

Direct Multicast Streaming Strategy. The primary goal is to deliver high performance while minimizing the latency of the events towards the destinations. To achieve this goal, using all the available routes, aggregated through the multi-route streaming towards all destinations, seems the right thing to do. The scheme of this transfer strategy is depicted in Figure 4.

In this way, no destination has a higher delay than the others (i.e., no priority is imposed) and no additional latency components are introduced, compared to point-to-point streaming. Nevertheless, small adjustments to the initial stream model are required to account for the number of destinations. As data is sent directly towards all destinations, the amount of bandwidth required increases proportionally with the number of destinations. Consequently, the model will increase the batch size with the number of destinations. The intuition is that, larger batches need to be formed compared to a point-to-point streaming, to provide sufficient time to transfer the batches in order not to throttle the network. Hence, the latency transfer component, which accounts for this strategy direct multicast streaming (i.e., labeled "DMS") is:

$$\text{Latency}_{\text{transferDMS}} = \frac{(sOt + sOe + \text{batch}_{\text{SizeMB}}) \times \text{Destinations}}{thr(\text{Size}_{\text{Total}}, \text{routes})}$$

The tradeoff for using this direct multicast strategy is the incurred costs. The typical cloud pricing model charges based on the amount of outbound data from a data center (i.e., the inbound data is usually free). Therefore, as each batch of events will be replicated with the number of destinations, before being sent to the remote data centers, the price will increase with a factor equal to the number of destinations.

Intermediate-Destination Multicast Streaming Strategy.. In order to control the price of multicast streaming we design a second strategy, presented in Figure 5. Instead of replicating the batches in the sender data center, these are sent only once to the destination site, where they are locally replicated. Clearly, this strategy reduces the cost over the Direct Multicast Streaming, having an equivalent outbound cost with a point-to-point scenario.

The first issue with this strategy, and the reason why we do not rely solely on it, is that at least one destination node needs to know and to be able to connect to all the other destinations. Because of that, we do not consider a strategy in which batches are sent once to a destination (i.e., not necessary the same) and then replicated by that destination node to the others. Although this might be one's default option, it would impose higher constraints on the system, by requiring open access between all destinations. Additionally, it would not provide lower latencies than the strategy proposed in Figure 5, as despite balancing the load across nodes, the theoretical average batch latency towards all destinations is the same. The second issue is the replication latency that is introduced. Despite being significantly smaller than the inter-site communication, this new latency component needs to be accounted for and added to the other latency components. The formula depicted bellow to model this is similar to the transfer latency component in

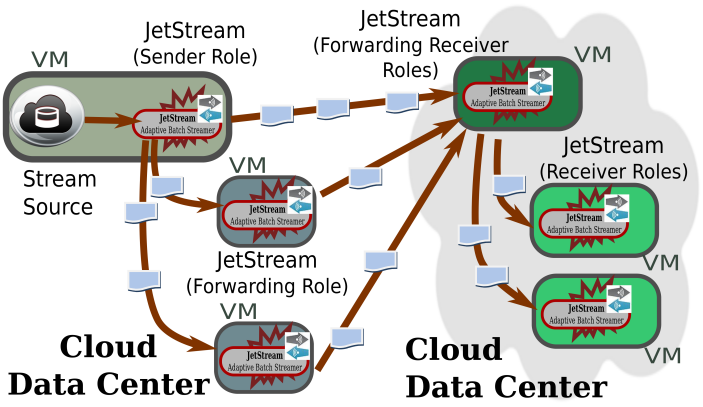

Figure 5. A multicast streaming strategy with a stream replication at destination side.

point-to-point streaming or in direct multicast streaming. The only difference is the throughput which corresponds to the intra-site communication.

$$\text{Latency}_{\text{destinationreplication}} = \frac{(sOt + sOe + \text{batch}_{\text{SizeMB}}) \times \text{Destinations}}{thr_{Intra-Site}(\text{Size}_{\text{Total}})}$$

3.5. Zoom on the selection of intermediate destinations

Choosing the right number of nodes to leverage multi-paths on each site and their optimal configuration among the available VMs has a great impact on the end-to-end performance. To this end, we designed an algorithm inspired by the behavior of ant colonies, studied in biology. These classes of algorithms are a good fit for finding which nodes to use for multi-paths as they are mainly used to solve shortest path problems, by relying on the ant mechanism of leaving information on the paths it has traversed, information called pheromone. The ants tend to follow the pheromone trails. Within a fixed period, shorter paths between nest and food can be traversed more often than longer paths, therefore the shorter paths are impregnated with a higher amount of pheromone, ergo a larger number of ants will choose them and thereby will reinforce the pheromone on the paths again.

To implement this approach, a set of agents, called artificial ants, are adopted to represent the behavior of real ant colonies. These ants work cooperatively and communicate indirectly through artificial pheromone trails. An ant constructs a problem solution iteratively by traveling a construction graph. Each edge of the construction graph represents the possible partial solution that the ant can take according to a probabilistic state transition rule. This rule provides a positive feedback mechanism to accelerate convergence and also keeps premature solution stagnation from happening.

In our case, we applied a modified version of this ant colony algorithm to find the best intermediary paths within a datacenter. We defined the weight of the path as the number of events read per second by each receiver. If the receiver does not reply after a period of time, the ant agent eliminates the intermediary node from the path by adding an infinite cost for that path and a null pheromone trail. We place an Ant Agent in each node from the same datacenter and start by applying a greedy algorithm to construct the first solution. The ants then travel from one node to another, leaving a pheromone trail on the edges.

In Algorithm 1 one can notice that the main characteristic of this approach is that the pheromone values are updated by all the ants that have completed the tour. Solution components c_{ij} are the edges of the graph, and the pheromone update for τ_{ij}, that is, for the pheromone associated to the edge joining nodes i and j, is performed as follows:

$$\tau_{ij} \leftarrow (1 - \rho) * \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

where $\rho \in (0, 1]$ is the evaporation rate, m is the number of ants and $\Delta\tau_{ij}^{k}$ is the quantity of pheromone laid on edge (i, j) by the k-th ant.

Algorithm 1. Ant Colony algorithm for finding the best multi-paths at each site

procedure AntNodeSelection($NumberOfNodes, NumberOfAnts, m, \rho, \alpha, \beta, JetStreamBest, BestCost$)
  P_{best} ←CreateHeuristicSolution(Number Of Nodes)
  $P_{best_{cost}} \leftarrow Cost(S_h)$
  $Pheromone \leftarrow InitializePheromone(P_{best_{cost}})$
  while ¬$StopCondition()$ **do** $Candidates \leftarrow \oslash$
    for i = 1 to m **do** $S_i \leftarrow ProbabilisticStepwiseConstruction(Pheromone, NumberOfNodes, \alpha, \beta)$ $S_{i_{cost}} \leftarrow Cost(S_i)$
      if ($S_{i_{cost}} \leq P_{best_{cost}}$) **then** $P_{best_{cost}} \leftarrow S_{i_{cost}}$ $P_{best_{cost}} \leftarrow S_i$
      end if
      if ($P_{best_{cost}} \leq BestCost$) **then** $BestCost \leftarrow P_{best_{cost}}$
      end if
      $Candidates \leftarrow S_i$
    end for
    $DecayPheromone(Pheromone, \rho)$
    for $S_i \in Candidates$ **do** $UpdatePheromone(Pheromone, S_i, S_{i_{cost}})$
    end for
    for $S_i \in Candidates$ **do** $\epsilon_i \leftarrow |BestJetStream_i - BestCost|$
      if ($S_{i_{cost}} \leq 0 || JetStreamBest + \epsilon_i < S_{i_{cost}}$) **then** $UpdatePheromone(Pheromone, S_i, S_{i_{cost}})$
      end if
    end for
  end while
  Return(P_{best})
end procedure

$$\Delta\tau_{ij}^k = \begin{cases} \frac{1}{L_k} & \text{if ant k used edge (i,j) in its tour,} \\ 0 & \text{otherwise.} \end{cases}$$

where L_k is the tour length of the k^{th} ant.

When constructing the solutions, the ants traverse the construction graph and make a probabilistic decision at each vertex. The transition probability $p(c_{ij}|s_p^k)$ of the k^{th} ant moving from city i to city j is given by:

$$p(c_{ij}|s_p^k) = \begin{cases} \frac{\tau_{ij}^{\alpha} * \eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s_p^k)} \tau_{il}^{\alpha} * \eta_{il}^{\beta}} & \text{, if } j \in N(s_p^k), \\ 0 & \text{otherwise.} \end{cases}$$

where $N(s_p^k)$ is the set of components that do not belong yet to the partial solution s_p^k of ant k , and α and β are parameters that control the relative importance of the pheromone versus the heuristic information $\eta_{ij} = \frac{1}{d_{ij}}$, where d_{ij} is the length of component c_{ij}. In the JetStream context, the pheromone update is done based on the read rate of each receiver. It represents the normalized value of the read rate in the interval [0, 1].Then, the pheromone update equation is: $\tau_{ij} = (1 - \rho) * \tau_{ij} + \Delta\tau_{ij}^{best}$

$$\tau_{ij}^{\alpha} = \begin{cases} \frac{1}{L_{best}} & \text{if the best ant used edge (i, j) in its tour} \\ 0 & \text{otherwise.} \end{cases}$$

The pheromone values are constrained between $\tau_{min} = 0$ and $\tau_{max} = JetStreamBest + \epsilon$ by verifying, after they have been updated by the ants, that all pheromone values are within the imposed limits. The minimum value τ_{min} is most often experimentally chosen (however, some theory about how to define its value analytically has been developed). In our context we set $\tau_{min} = 0$. The maximum value τ_{max} may be calculated analytically provided that the optimum ant tour length is known. Before applying ant, we evaluate the system with JetStream to determine the initial parameters, therefore τ_{max} is set to the read rate obtained and ϵ_i is set to the difference between the best read rate for $node_i$ and the $JetStreamBest_i$.

The distance between each two nodes is represented by the first read rate we obtain from JetStream without Ant, normalized in the interval $[1, 10]$. The ant agents stop when they exceed a time limit or when they constructed a solution where the amount of events read by each receiver is the same or higher than the best read rate provided by JetStream, having the same parameters such as batch size and event generation rate.

Algorithm 2. 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 batch_size < MaxBatched do
6:              ESTIMATE latency_batching from [Rate_Acquisition, batch_size]
7:              ESTIMATE latency_encoding from [overheads, batch_sizeMB]
                                          ▷ Estimate the transfer latency for 1 channel
8:              ESTIMATE latency_transfer1 from [batch_sizeMB, thrRef, 1channel]
9:              COMPUTE Ratio_CPU from [latency_encoding, latency_batching, VM_Cores]
                                                          ▷ Prevents idle channels
10:             if Ratio_CPU * latency_batching × channels < latency_transfer1 + latency_encoding then
11:                 ESTIMATE latency_decoding from [overheads, batch_sizeMB]
12:                 ESTIMATE latency_transfer from [batch_sizeMB, thrRef, channels]
13:                 ESTIMATE latency_reordering from [channels, latency_transfer]
14:                 latency_perEvent = ∑ latency_*
15:                 if latency_perEvent < bestLatency then
16:                     UPDATE [bestLatency, Obatch, Ochannels]
17:                 end if
18:             end if
19:         end while
20:     end while
21: end procedure
```

3.6. Adaptive cloud batch-streaming

In Algorithm 2, we present the decision mechanism for selecting the number of events to batch and the parallelism degree (i.e., channels/routes) to use. The algorithm successively estimates the average latency per event, using the formulas presented previously, for a range of batch sizes and channels, retaining the best one. As an implementation optimization, instead of exhaustively searching in the whole space we apply a simulating annealing technique, 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). Imposing an upper bound for the batch size limits the latency to form the batch (i.e., fewer events are needed) and thus the end-to-end delay.

The selection of the number of channels is done by estimating how many batches can be formed while one is being transferred (Lines 6-8). Beyond this point, adding new channels leads to idle resources and therefore decreases the cost efficiency. The condition on Line 10 prevents the system from creating such idle channels. Finally, the CPU usage needs to be accounted in the decision process, being an important factor which determines the transfer rate, as revealed by our performance study. Sending frequent small batches will increase the CPU consumption and artificially decrease the overall performance of the cloud node. We therefore assign a penalty for the CPU usage, based on the ratio between the time to form a batch (a period with a low CPU footprint) and the time used by the CPU to encode it (a CPU intensive operation), according to the formula:

$$\text{Ratio}_{CPU} = \frac{\text{latency}_{\text{encoding}}}{(\text{latency}_{\text{batching}} + \text{latency}_{\text{encoding}}) \times VM\_\text{Cores}}$$

When computing the ratio of intense CPU usage, we account also for the number of cores available per VM. Having a higher number of cores prevents CPU interferences from overlapping computation and I/O and therefore does not require to prevent using small batches. With this model, the batch decision mechanism is aware of the CPU usage and the intrusiveness of the transfer on the computation, and therefore adapts the batch size accordingly. To sum up,

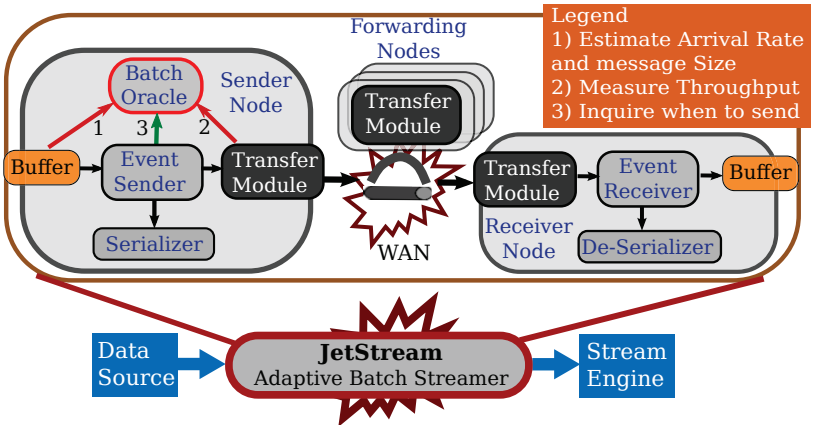

Figure 6. The architecture and the usage setup of the adaptive batch streamer.

JetStream collects a set of context parameters (Line 2) and uses them to estimate the latency components according to the formulas presented previously. Based on these estimations, it selects the optimal batch size and the number of channels for streaming.

4. JetStream Architecture Overview

We designed JetStream as a high-performance cloud streaming middleware. The system is easily adoptable and ready to be used in any scenario, as it does not require changes in application semantics nor modifications or elevated privileges to the cloud hypervisor or processing engines. Its conceptual scheme is presented in Figure 6. 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. Hence, the adaptive-batch approach remains transparent to the system and user. The distributed modules composing the system and their functionality are described below.

The Buffer is used both as an event input and output endpoint for JetStream, present at the sender and at the receiver. The sender application or the event source simply 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 is notified (asynchronously) when they are available. The Buffer entity at the sender side is also in charge of monitoring the input stream in order to assert and report the *acquisition rate* of the events and their *sizes* in real time.

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 2 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 the multi-route streaming, using the approach presented in [19]. Batches are sent in a round-robin manner through the channels assigned for the transfer. On the intermediate nodes, the role of this component is to forward the batches towards the destination. Currently, the system offers 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 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 and the acknowledgment-related mechanism proposed for multi-route transfers, which is discussed in more details in [19]. 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, buffered and reordered, and delivered to the application as a stream of events, in a transparent fashion for the stream processing engine. The module issues acknowledgments 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 are 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. Moreover, this module can be extended to host additional functionality: data compression, deduplication, etc.

5. Implementation

We implemented JetStream in C# using the .NET 4.5 framework as a modular system that would enable easy future extensions. To this end, the components shown in Figure 6 are designed using several software design patterns such as abstract factory (e.g., for the transfer module), publish-subscriber (e.g., for the receiver), facade (e.g., sender) or flyweight (e.g., serializer libraries). Additionally, we adopted a number of implementation optimizations as follows.

The *Buffer module*, used both at the sender and receiver side, is implemented using 2 queues, which are used in a producer-consumer fashion. Using a separate queue for input operations to the buffer and another one for the output ones allows to reduce the number of blocked read/write operations. In this way locking is required only to swap the queues when the output (or consumer) one gets empty. At the receiver side, the component enables sequential access, for which we built a dedicated iterator. For the *Transfer module* we had as option to implement the communication threads as an object pool. In this way, each channel used for multi-route streaming is rapidly assigned to a thread to handle the sending or receiving of batches. Nevertheless, we observed that this option can lead to performance degradations particularly for single core VMs. This happens due to the increase in context switches. The alternative we chose is to implement single-thread communication and multiplexing the management of the channels. Although we have tried different strategies to implement the multiplexing (e.g., round-robin, bag of tasks, priority queues) we did not observe significant performance differences. For the *Event Receiver* reordering, we use insertion sort, exploiting the observation that batches arrive as a time sequence. Finally, the *Batch Oracle* is implemented as a singleton component per (sender) node.

6. 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 its performance.

6.1. Experimental setup

The experiments were run in the Microsoft's Azure cloud in the North-Central US and the North EU data centers, using Small Web Role VMs (1 CPU, 1.75 GB of memory, 225 GB local storage). For multi-route streaming, up to 5 additional nodes were used within the sender deployment. Each experiment sends between 100,000 and 3.5 million events, which, depending on the size of the event to be used, 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).

6.2. Experimental methodology

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. The evaluation is performed using a synthetic benchmark, that

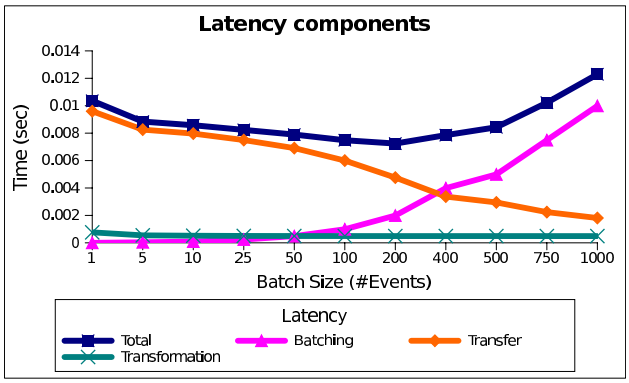

Figure 7. The latency components per event with respect to the batch size for a single streaming channel, for inter-site event transfers

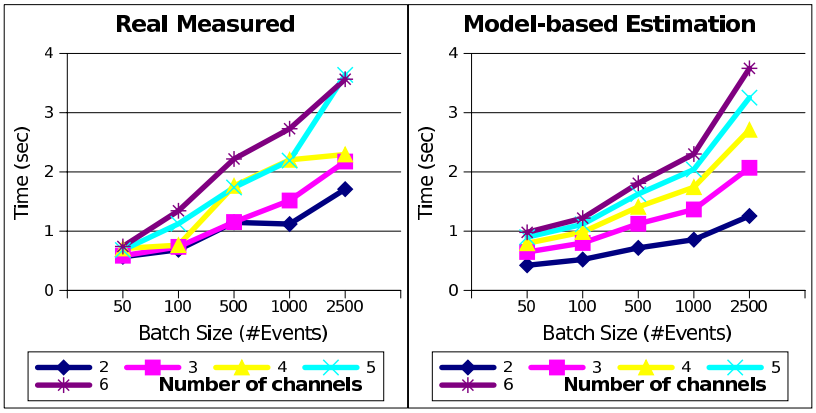

Figure 8. Comparing the estimation for the unordered batch latency (i.e., the penalty) with actual measurements, while increasing the number of channels used for the transfers

we created 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. Additionally, we use the monitoring readings collected by the MonALISA monitoring system from the ALICE LHC experiment to assert the gains brought by our approach in a real geographically distributed setup.

6.3. Accuracy

We depict on Figure 7 the total latency per event and its components, as defined in Section 3.3, 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 illustrated scenario). The search for the batch size that minimizes the latency per event is at the core of the JetStream algorithm presented in Section 3.6. However, the selection is computed based on estimations about the environment (e.g., network throughput, CPU usage), 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. However, considering that the performance of using batches with sizes around the optimal value is roughly similar (as seen in Figure 7), JetStream delivers more than 95 % of the optimal performance even in the case of large and unrealistic shifts from the optimum batch size (e.g., by mis-selection of a batch size of 150 or 250 instead of 200 in Figure 7, the performance will decrease with 3 %). The solution for further increasing the accuracy, when selecting the batch size, is to monitor the cloud more frequently, which comes at the price of higher intrusiveness and resource usage levels.

To validate the penalty model proposed for the latency of reordering batches when using multiple routes, we compare the estimations computed by our approach with actual measurements. The results are presented in Figure 8. The delay for unordered batches was measured as the time between the moment when an out of order batch (not

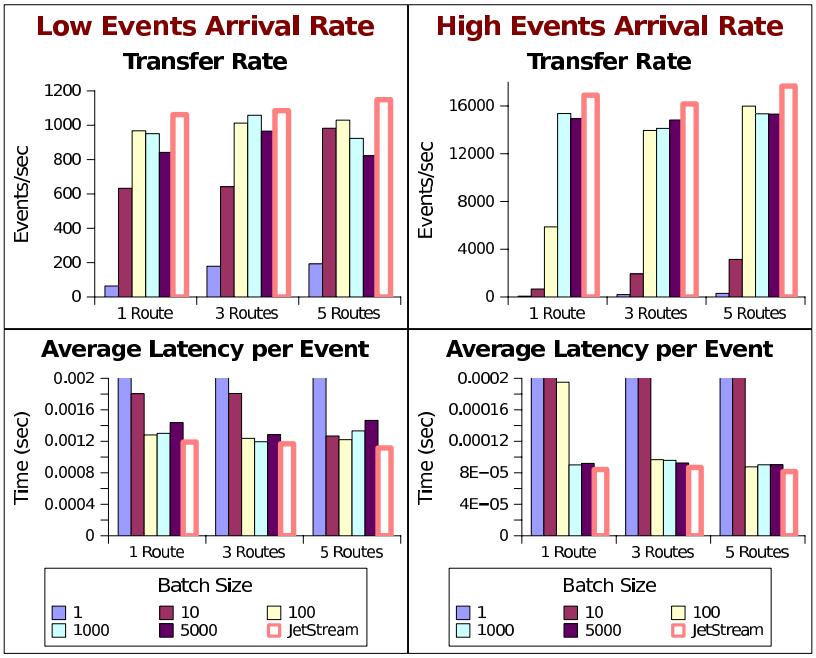

Figure 9. Comparing the performance (transfer rate - top and average latency per event - bottom) of individual event streaming and static batches with JetStream for different acquisition rates, while keeping the size of an event fixed (224 bytes). The performance of individual event streaming (i.e., batch of size 1) are between 50 to 250 times worse than the ones of JetStream.

the one next in sequence) arrives, and the moment when the actual expected one arrives. The experiment 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, having an accuracy of 90–95 %. Errors appear because the model for the reordering delay introduced by multi-route streaming 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.

6.4. 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 between cloud data centers. For the later approach we consider both static batch sizes as well as the adaptive batch selection of JetStream. In the case of static sizes, the number of events to be batched is fixed a priori. A batch of size 1 represents event by event streaming. These setups are compared to JetStream, which implements the proposed model for adapting the batch size to the context at runtime. To this end, the evaluation is performed with different event sizes and generation rates. The experiments were repeated for different number of routes for streaming: 1, 3 and 5. We measured the transfer rates (top) and average latency per event (bottom).

The experiments presented on Figure 9 use an event of size 224 bytes and evaluate the transfer strategies considering low (left) and high (right) event generation rates. The first observation is that batch-based transfers clearly outperform individual event transfers for all the configurations considered. The results confirm the impact of the streaming overheads on small chunk sizes and the resulting low throughput achieved for inter-site transfers. Grouping the events increases the transfer rate tens to hundreds 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 variations of the cloud resources (e.g., nodes, network links). Static batches cannot provide the solution, as certain batch sizes are good in one context and bad in others. For example batches of size 10 deliver 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 % for the event transfer rate over static batch configurations (for good batch sizes not for values far off the optimal). To confirm the results, we

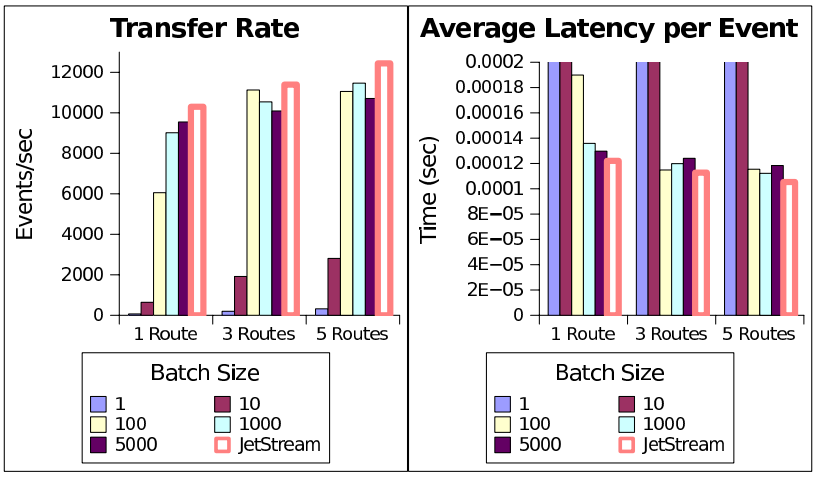

Figure 10. The performance (transfer rate - top and average latency per event - bottom) of individual event streaming, static batches and JetStream for events of size 800 bytes. The performance of individual event streaming (i.e., batch of size 1) are from 40 to 150 times worse than JetStream.

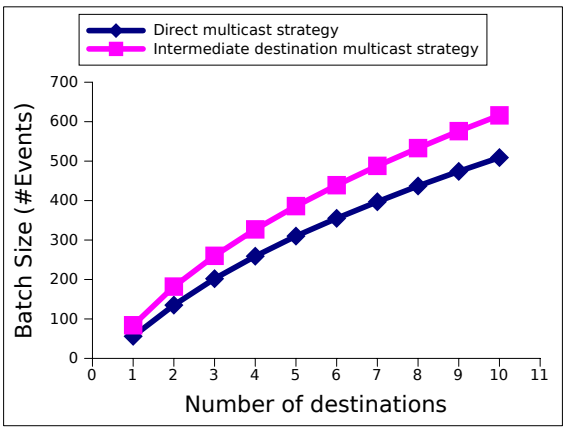

Figure 11. The correlation between the number of events to batch and the number of multicast destinations, for the 2 strategies proposed.

repeated the same set of experiments for larger event sizes. Figure 10 illustrates the measurements obtained for event sizes of 800 bytes. The results support our conclusions, showing that JetStream is able to increase the performance with up to 2 orders of magnitude over current streaming strategies.

6.5. Multicast streaming strategies

In Figure 11, we show the impact that the number of destinations have on the number of batched events, for each of the 2 proposed strategies. As discussed in Section 3.4, a direct approach for implementing the multicast will lead to lower transfer latencies compared to a strategy that uses additional intermediate hops. This can be indirectly observed by comparing the batch sizes, with larger batches leading to larger latencies. The second observation is that the batch sizes increase with the number of destinations, for both strategies. This is explained by the fact that having more destinations requires more network I/O and therefore more CPU cycles to be allocated by the nodes for the transfer. Large batches favor such periods as the CPU is idle (i.e., from the point of view of batch creation) most of the time during the forming of the batch and the CPU-intensive encoding operations are more sparse. This decreases the CPU interferences which leads to an increase of the overall performance, as discussed also in [15].

6.6. Adapting to the context changes

The event acquisition process in streaming scenarios 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

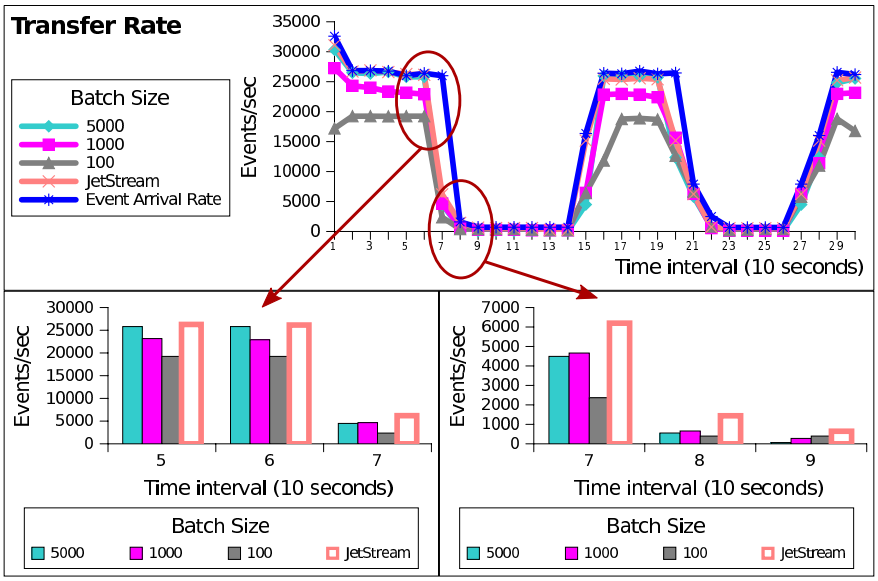

Figure 12. The evolution of the transfer rate in time for variable event rates with JetStream and static batches transfer strategies

or the cloud performance variability [20]. 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 12 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 and number of routes. In contrast, static batch transfers either are introducing huge latencies 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 events not transferred yet (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.

JetStream reduces the number of used resources by 30 % in such scenarios (e.g., the extra nodes which enable multiple route streaming), as it takes into account the stream context when scheduling the transfers. These resource savings are explained by the fact that low event acquisition rates do not require multiple route streaming as higher ones do. Additionally, as shown in [12], the fluctuations in application load have certain patterns across the week days. Hence, in long running applications, our approach will make substantial savings by scaling up/down the number of additional nodes used for transfers to these daily or hourly trends.

6.7. Benefits of multiple route streaming

Figure 13 shows the gains 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). When increasing the amount of data to be sent, multi-route streaming pays off for both strategies. This validates our decision to apply the transfer scheme proposed in [19] for bulk data transfers to streaming. By aggregating extra bandwidth from the intermediate nodes, we are able to decrease the impact of the overhead on smaller batches: batch metadata, communication and serialization headers. More precisely, a larger bandwidth allows to send more data, and implicitly, the additional data carried with each batch does not throttle the inter-site network anymore. This brings the transfer rate of smaller, and consequently more frequent, batches closer to the maximum potential event throughput. This can be observed for the static batch of size 100 on Figure 13, which delivers a throughput close to JetStream for a high number of routes.

With higher throughput and a lower overhead impact, the optimal batch size can be decreased. In fact this is leveraged by JetStream, which is able to decrease the end-to-end latency by selecting lower batch sizes. Hence, we conclude that sustaining high transfer rates under fixed time constraints is possible by imposing upper bounds for the batch sizes and compensating with additional streaming routes. This enables JetStream to integrate users' time

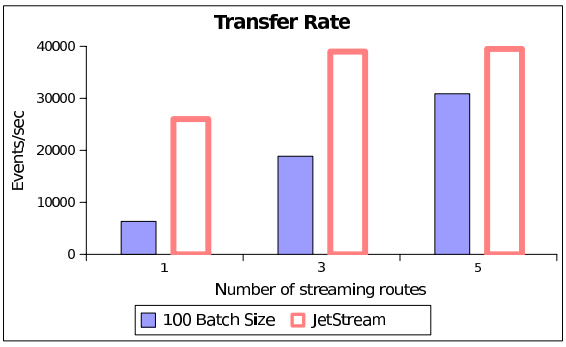

Figure 13. The transfer rate for an increasing number of routes used for streaming, when the batch size is fixed or adaptively chosen using JetStream.

constraints for maximal delay, which are integrated in the streaming decision shown in Algorithm 2 by considering a limit on the batch size.

6.8. 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) [10], one of four LHC (Large Hadron Collider) experiments at CERN, 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, 86 institutes and more than 80 computing centers worldwide, 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. Our focus, in these series of experiments, is on the monitoring information collected in real-time about all ALICE resources. We used the MonALISA [9] service to instrument and replay 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. The MonALISA framework and its high-frequency updates for large volumes of monitoring data matched closely with JetStream's architecture purposes, being the reason why we chose this as a real application scenario. Based on the monitoring data collected 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.

Streaming monitoring data of the ALICE CERN experiment with JetStream. Figure 14 a) shows the total latency of the events at the sender and the transfer rate, Figure 14 b), when comparing JetStream with static configurations for various batch sizes. The transfer performance 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, these 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 14 b). But on the other hand, it will increase the latency of the events as depicted in Figure 14 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, the latency obtained with JetStream is reduced from 2.2 (100-event batches) down to 17 times (10,000-event batches).

7. Discussion

We now discuss our experience running JetStream in production on the Azure cloud, where it is primarily used to to handle monitoring data from the ALICE experiment. We also introduce several ongoing extensions.

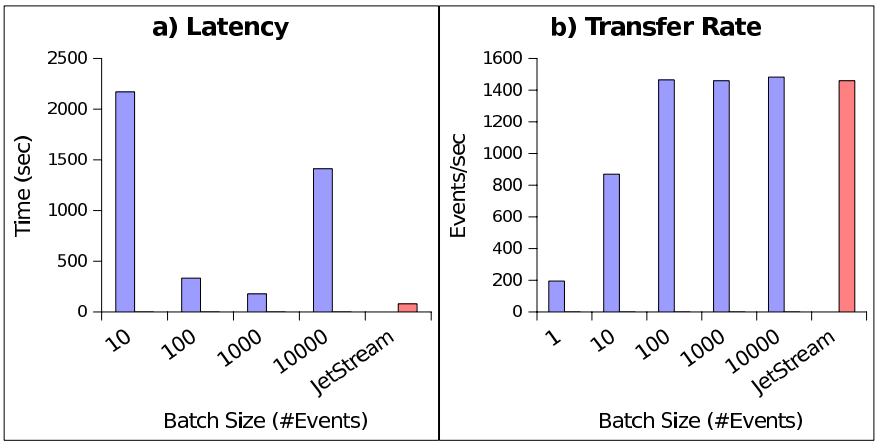

Figure 14. The total latency (a) and the transfer rate (b) measured when transferring 1.1 million events from MonALISA monitoring the Alice experiment. The latency for independent event transfer (batch of size 1) in a) is not represented because it would modify the scale, having a value of 30900 seconds as opposed to 80 seconds for JetStream.

Streaming as a Service. The multi-path and multi-hop transfer strategies leveraged within JetStream allow to accommodate many different users at the same time, by means of network parallelism (i.e. scaling up the number of streams). The system processes any number of streams the more nodes are added to the underlying paths, and thus can support an arbitrary amount of users. This feature makes JetStream an ideal candidate for enabling the Streaming as a Service (SaaS) concept. The paradigm is supported by seamlessly allowing users to subscribe to a set of events they are interested in and that are published by producers through our simple API. This SaaS approach can help ”democratise” the access to fast data for a large spectrum of non-specialist users, as it would be offered transparently by the cloud provider. This shift of perspective comes naturally: instead of letting users optimise their event transfers by making (possible false) assumptions about the underlying network topology a and performance through intrusive monitoring, JetStream SaaS delegates this task to the cloud provider. Indeed, the cloud owner has extensive knowledge about its network resources, which it can exploit to optimise (e.g., by grouping) user event transfers, through this dedicated service. We argue that the adoption of such a SaaS approach brings several benefits for both users and the cloud providers who propose it. For users of multi-site or federated clouds, this proposal is able to decrease the variability of event transfers and increase the throughput, while benefiting from the well-known high availability of cloud-provided services. For cloud providers, such a service can decrease the energy consumption within a datacenter by means of more effective use of resources.

Self-optimization vs. bad user configurations. The JetStream deployment on the Azure cloud of for processing MonALISA data used 2 intermediate nodes located at the sender side (i.e., resulting in 3 streaming routes). Initially, the setup considered 5 streaming routes. However, during the transfer of the data using JetStream, we observed that at most 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 offline 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 self-adaptive approach, not subject to arbitrary human configuration choices. Furthermore, long running experiments allow JetStream to exploit other observations about its execution context, like the time patterns (e.g. daily usage models with spikes during working hours and idle periods at night and weekends).

Strong vs. weak consistency. JetStream enables strong consistency by design. All transferred events are totally ordered and cannot be lost within the service. As seen in the previous section, even in the presence of such a strong constraint, the system achieves high throughput and is able to seamlessly scale up. However, number of applications

don't actually need such strong consistency requirements. For instance, all applications handling events with timestamps (e.g. monitoring, wireless sensors) typically order them when storing into databases. Hence, the ordering at transfer time might be useful but not always needed. This is the reason why the consistency level in JetStream can be adjusted by users at runtime, according to their application needs. A weak level eliminates the ordering stage and has the potential to further increase throughput, especially when traversing links with small or variable bandwidth.

Cost effectiveness. Efficiency can have multiple declinations depending on application context and user requirements. In clouds, the most critical among these are the transfer time and the monetary cost. Our model estimates these metrics for transferring events between the source and destination sites, with respect to the transfer setting (used resources, geographical location etc.).

- The transfer time (Tt) is estimated considering the number of nodes (n) that are used in parallel to stream events and the predicted transfer throughput (thr_{model}), obtained from monitoring the respective cloud link:

$$Tt = \frac{Size}{thr_{model}} * \frac{1}{1 + (n-1) * gain} \tag{1}$$

  where *gain* is the time reduction due to parallel transfers (determined empirically, with values less than 1).

- The cost of a geographical transfer is split into 3 components. The first corresponds to the cost charged by the cloud provider for outbound data ($outbound_{Cost}$), as usually inbound data is free. The other two components are derived from the cost of the VMs (n - number of VMs) that are leased: the network bandwidth (VMC_{Band}) and the CPU (VMC_{CPU}) costs. The ratio of the VM resources used for the transfer is given by the intrusiveness parameter ($Intr$). The final cost equation is:

$$Cost = n * (Tt * VMC_{CPU} * Intr + \frac{\frac{\frac{Size}{n}}{Tt} * Intr}{VMC_{Band}}) + outbound_{Cost} * Size \tag{2}$$

  where for simplicity we considered that each of the n nodes sends the same amount of data ($\frac{Size}{n}$).

This model captures the correlation between performance (time) and cost (money) and is able to adjust the tradeoff between them dynamically during transfers. An example of such a tradeoff is setting a maximum cost for the live event streaming, based on which our system is able to infer the amount of resources to use. Although the network or end-system performance can drop, the system rapidly detects the new reality and adapts to it in order to satisfy the budget constraint. The proposed model is rather simple, relying on a small set of parameters, easily collected by monitoring tools (e.g. sample history depends on the cloud variability) or set by users based on the application type (e.g. the CPU-intrusiveness tolerated by compute-intensive applications could be smaller: 5%, while I/O-intensive applications would tolerate a bigger one: 10%). These parameters are used to calibrate a general methodology based on component monitoring, application side feedback and behavior/pattern analysis to discover and characterize situations leading to fluctuations of data access throughput. Hence, this model is not tailored for specific configurations of resources/services, and can be transported from one application to another. This very simplicity allows the model to be more general, but at the expense of becoming less accurate for some precise, application dependent, settings. It was our design choice to trade accuracy for generality. Moreover, using knowledge discovery techniques for calibrating these parameters reduces the possible impact of biased analysis manually performed by administrators.

Integration with stream processing engines. Thanks to its simple API, JetStream can be used as an event transfer backend by any stream processing engine (SPE). One potential benefit of such an integration would be the possibility to leverage the applications / queries semantics in order to minimise the data movements. For instance, knowing the query a priori allows to transfer only the relevant events from their distant source to the destination. This can be achieved as modern production sites (e.g. wireless sensors) are typically enhanced with some light data processing facilities (e.g. filtering), while events are expressed as records, easy to identify and map to some SQL queries. With this approach, JetStream enables a movement of the query from the destination to the event source, complementing the traditional movement of events from their source to the processing destination. An interesting use-case for this approach would be an application streaming events from a source, which is a query applied by another SPE on a different stream.

8. Related Work

There is a rich eco-system of research on building high performance and highly available event streaming services. Most of the works focus on optimising the query processing in real-time over streams, assuming that the events were already collected and are close to the processing engine. We zoom out from this perspective, focusing on *how the events are sent from their geographically distributed production sites to the processing datacenters*. Similar to JetStream, several works distribute these *fast data* movements across several nodes, in an attempt to scale with the number of queries. Although basic functions to move events between nodes are provided, these systems are not designed to run in the dynamic environment of the Cloud. There have been first attempts to support more dynamism in distributed stream processing engines, however none of them takes into account the challenges related to low latencies and high performance heterogeneity of the multi-datacenter clouds. The remainder of this section presents a taxonomy of the current state-of-the-art in event streaming communication.

Cloud based streaming. Streaming solutions for cloud have just started to emerge. ElasticStream [21] support a cost efficient scheme to transfer incoming events to the cloud when local nodes cannot accommodate it. Based on data rates predictions, the minimum number of required cloud nodes is computed according to latency limits in SLA. Although, this approach costs less than reserving a fixed number of cloud nodes, it does not provide true elasticity, in the JetStream sense, as nodes are added based on periodic predictions rather than real-time load. Yahoo!'s S4 [22] cloud streaming service relies on a symmetric design in which every processing element is configured to handle key/value based events. Communication among nodes and automatic failover is managed by ZooKeeper. One issue with S4 is its fixed allocation of processing elements on the cloud nodes, hardcoded into the application code. This impacts the load balancing and prevents exploiting the cloud elasticity. Some works rely on the MapReduce model to process events [23]. These approaches do not yet meet the needs of a general streaming system either because they still hardly rely on persistent storage of data or they still do not meet the ultra-low latency required in streaming applications. In [24], the authors provide an accurate cost and latency estimation for complex event processing operators, validated with the Microsoft StreamInsight [25]. Sphere[26], proposes a GPU-like processing on top of a high performance infrastructure. In [27], 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 [28] consider exploiting the geographically distributed nature of stream data sources. However, these 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.

Streaming in peer-to-peer systems. Many current concepts used for cloud event streaming are inherited from P2P systems. These can be divided in two categories based on how peers that forward the events organize themselves in an overlay network [29]: some use DHT overlays [30, 31] and others group the subscribers in interest groups based on event semantics [32, 33]. 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 [29]. 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[34], 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.

Systems for data transfers. Several solutions have been proposed in the last years for managing data transfers at large scale. StorkCloud [35] 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 [36]. 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 [37], 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 [38] 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 [40]. Additionally, such solutions integrate low-level infrastructure optimizations based on network type, cluster knowledge or cross-layer architectures [41, 42]. 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 [43], 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 [44], 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 [44] 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.

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 [23] 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 [45]. Here, the data acquisition is event driven: the system simply collects the events from the source. Comet [12] enables batch processing across streams of data. It is built on top of Dryad[46] 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 [47], 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 [13] is another example of DSMS which scales the query processing by partitioning and event distribution. Middleware solutions like System S from IBM [48] were proposed for single cluster processing, with processing elements connected via typed streams.

Dynamic adaptation for streaming. Several solutions aim to provide a dynamic behavior at different phases of stream processing. SPC [49] 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 [50]. Despite the fact that this solution targets the management of input streams, it has no support for the efficient transmission of data. Other solutions [51] 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 [52]. 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 [53] 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.

9. Summary and Future Work

We have motivated the need for a framework for *fast data* movements across geographically distributed datancenters. In this paper, we have described such a framework, JetStream, which leverages the idea of adapting the batch size to the transfer context to enable high performance event streaming in the cloud. 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. The nodes to select and their optimal configuration are chosen by a biology inspired algorithm, that mimics the movement of ant colonies in their quest for food.

We have also reported on our experience using JetStream on large deployments across several Europe and US datacenters of the Azure Cloud. The system was validated using synthetic benchmarks, and real-life monitoring data collected with MonALISA from the ALICE experiment at CERN. JetStream increases the transfer rates up to 250 times compared to individual event transfers. The adaptive selection of the batch size further increases the performance with an additional 25% compared to static batch size configurations. Finally, the multi-route streaming triples the performances and decrease the end to end latency while providing high transfer rates. Overall, we conclude that a framework to process fast data, such as implemented in JetStream, is feasible and highly promising, in terms of allowing developers to quickly write fast-data applications, and to achieve low latency and high scalability in those applications.

Learning from our experience, we are currently deploying JetStream to more applications and are extending it in several important directions. We plan to enhance the communication module with additional transfer protocols and to automatically switch between them based on the streaming pattern. We are also investigating several competitive pricing schemes for the event transfers in order to make the Streaming as a Service approach of JetStream more appealing to cloud providers. In particular, we would like to propose a dynamic cost model for the service usage, which enables the cloud providers to regulate and encourage data exchanges via a data transfer market. We are also interested by the advantages brought by the integration of JetStream with existing stream processing engines, namely the potential data movement reduction when moving the queries from the processing to the production sites.

10. Acknowledgments

The authors would like to thank Andreea Pintilie for her valuable input related to the ant-based algorithm for the intermediate node selection. This work was partially supported by the joint INRIA - Microsoft Research Center in the framework of the ZCloudFlow project.

References

[1] T. Hey, S. Tansley, K. M. Tolle (Eds.), The Fourth Paradigm: Data-Intensive Scientific Discovery, Microsoft Research, 2009.
[2] L. Golab, M. T. Özsu, Issues in data stream management, SIGMOD Rec. 32 (2) (2003) 5–14. doi:10.1145/776985.776986.
URL http://doi.acm.org/10.1145/776985.776986

[3] R. Kienzler, R. Bruggmann, A. Ranganathan, 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, IEEE Computer Society, Washington, DC, USA, 2012, pp. 159–166.
[4] 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.
[5] Ocean Observatory Initiative, http://oceanobservatories.org/.
[6] A. Baptista, B. Howe, J. Freire, D. Maier, C. T. Silva, Scientific exploration in the era of ocean observatories, Computing in Science and Engg. 10 (3) (2008) 53–58. doi:10.1109/MCSE.2008.83.
URL http://dx.doi.org/10.1109/MCSE.2008.83
[7] M. Arrott, A. Clemesha, C. Farcas, E. Farcas, M. Meisinger, D. Raymer, D. LaBissoniere, K. Keahey, Cloud provisioning environment: Prototype architecture and technologies, Ocean Observatories Initiative Kick Off Meeting.
[8] Microsoft azure, http://azure.microsoft.com/en-us/.
[9] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, C. Dobre, A. Muraru, A. Costan, M. Dediu, C. Stratan, Monalisa: An agent based, dynamic service system to monitor, control and optimize distributed systems, Computer Physics Communications 180 (12) (2009) 2472 – 2498, 40 {YEARS} {OF} CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures. doi:http://dx.doi.org/10.1016/j.cpc.2009.08.003.
URL http://www.sciencedirect.com/science/article/pii/S0010465509002410
[10] CERN Alice, http://alimonitor.cern.ch/map.jsp.
[11] R. Tudoran, O. Nano, I. Santos, A. Costan, H. Soncu, L. Bougé, G. Antoniu, Jetstream: Enabling high performance event streaming across cloud data-centers, in: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, ACM, New York, NY, USA, 2014, pp. 23–34. doi:10.1145/2611286.2611298.
URL http://doi.acm.org/10.1145/2611286.2611298
[12] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, L. Zhou, Comet: Batched stream processing for data intensive distributed computing, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, ACM, New York, NY, USA, 2010, pp. 63–74. doi:10.1145/1807128.1807139.
URL http://doi.acm.org/10.1145/1807128.1807139
[13] C. Balkesen, N. Dindar, M. Wetter, 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, ACM, New York, NY, USA, 2013, pp. 3–14. doi:10.1145/2488222.2488257.
URL http://doi.acm.org/10.1145/2488222.2488257
[14] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, Tuning qod in stream processing engines, in: Proceedings of the Twenty-First Australasian Conference on Database Technologies - Volume 104, ADC '10, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2010, pp. 103–112.
URL http://dl.acm.org/citation.cfm?id=1862242.1862257
[15] R. Tudoran, K. Keahey, P. Riteau, S. Panitkin, G. Antoniu, Evaluating streaming strategies for event processing across infrastructure clouds, in: Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, 2014, pp. 151–159. doi:10.1109/CCGrid.2014.89.
[16] ATLAS, http://home.web.cern.ch/fr/about/experiments/atlas.
[17] ATLAS Applications, https://twiki.cern.ch/twiki/bin/viewauth/AtlasProtected/PhysicsAnalysisWorkBookRel16D3PDAnalysisExample.
[18] C. Esposito, M. Ficco, F. Palmieri, A. Castiglione, Interconnecting federated clouds by using publish-subscribe service, Cluster Computing 16 (4) (2013) 887–903. doi:10.1007/s10586-013-0261-z.
URL http://dx.doi.org/10.1007/s10586-013-0261-z
[19] R. Tudoran, A. Costan, R. Wang, L. Bouge, G. Antoniu, Bridging data in the clouds: An environment-aware system for geographically distributed data transfers, in: Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, 2014, pp. 92–101. doi:10.1109/CCGrid.2014.86.
[20] I. Foster, A. Chervenak, D. Gunter, K. Keahey, R. Madduri, R. Kettimuthu, Enabling PETASCALE Data Movement and Analysis, Scidac Review.
URL http://www.scidacreview.org/0905/pdf/cedps.pdf
[21] A. Ishii, T. Suzumura, Elastic stream computing with clouds, in: Cloud Computing (CLOUD), 2011 IEEE International Conference on, 2011, pp. 195–202. doi:10.1109/CLOUD.2011.11.
[22] Yahoo! S4, http://incubator.apache.org/s4/.
[23] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: Fault-tolerant streaming computation at scale, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, ACM, New York, NY, USA, 2013, pp. 423–438. doi:10.1145/2517349.2522737.
URL http://doi.acm.org/10.1145/2517349.2522737
[24] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, 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, IEEE Computer Society, Washington, DC, USA, 2011, pp. 255–266. doi:10.1109/ICDE.2011.5767926.
URL http://dx.doi.org/10.1109/ICDE.2011.5767926
[25] Microsoft streaminsight, http://technet.microsoft.com/en-us/library/ee362541(v=sql.111).aspx.
[26] Y. Gu, R. L. Grossman, Sector and sphere: The design and implementation of a high-performance data cloud, Philosophical Transactions: Mathematical, Physical and Engineering Sciences 367 (1897) (2009) pp. 2429–2445.
URL http://www.jstor.org/stable/40485591
[27] P. N. Martinaitis, C. J. Patten, A. L. Wendelborn, Component-based stream processing "in the cloud", in: Proceedings of the 2009 Workshop on Component-Based High Performance Computing, CBHPC '09, ACM, New York, NY, USA, 2009, pp. 16:1–16:12.

doi:10.1145/1687774.1687790.
URL http://doi.acm.org/10.1145/1687774.1687790

[28] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: A new model and architecture for data stream management, The VLDB Journal 12 (2) (2003) 120–139. doi:10.1007/s00778-003-0095-z.
URL http://dx.doi.org/10.1007/s00778-003-0095-z

[29] M. A. Tariq, B. Koldehofe, 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, ACM, New York, NY, USA, 2013, pp. 51–62. doi:10.1145/2488222.2488262.
URL http://doi.acm.org/10.1145/2488222.2488262

[30] A. Gupta, O. D. Sahin, D. Agrawal, 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, Springer-Verlag New York, Inc., New York, NY, USA, 2004, pp. 254–273.
URL http://dl.acm.org/citation.cfm?id=1045658.1045677

[31] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup protocol for internet applications, IEEE/ACM Trans. Netw. 11 (1) (2003) 17–32. doi:10.1109/TNET.2002.808407.
URL http://dx.doi.org/10.1109/TNET.2002.808407

[32] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, K. Rothermel, Meeting subscriber-defined qos constraints in publish/subscribe systems, Concurr. Comput. : Pract. Exper. 23 (17) (2011) 2140–2153. doi:10.1002/cpe.1751.
URL http://dx.doi.org/10.1002/cpe.1751

[33] M. A. Tariq, B. Koldehofe, G. G. Koch, 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, ACM, New York, NY, USA, 2012, pp. 213–224. doi:10.1145/2335484.2335508.
URL http://doi.acm.org/10.1145/2335484.2335508

[34] S. Loesing, M. Hentschel, T. Kraska, 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, ACM, New York, NY, USA, 2012, pp. 55–60. doi:10.1145/2320765.2320789.
URL http://doi.acm.org/10.1145/2320765.2320789

[35] T. Kosar, E. Arslan, B. Ross, 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, ACM, New York, NY, USA, 2013, pp. 29–36. doi:10.1145/2465848.2465855.

[36] E. Yildirim, T. Kosar, Network-aware end-to-end data throughput optimization, in: Proceedings of the First International Workshop on Network-aware Data Management, NDM '11, 2011, pp. 21–30. doi:10.1145/2110217.2110221.
URL http://doi.acm.org/10.1145/2110217.2110221

[37] N. Laoutaris, M. Sirivianos, X. Yang, P. Rodriguez, Inter-datacenter bulk transfers with netstitcher, in: Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM '11, ACM, New York, NY, USA, 2011, pp. 74–85. doi:10.1145/2018436.2018446.

[38] I. Foster, R. Kettimuthu, S. Martin, S. Tuecke, D. Milroy, B. Palen, T. Hauser, J. Braden, Campus bridging made easy via globus services, in: Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond, XSEDE '12, ACM, New York, NY, USA, 2012, pp. 50:1–50:8. doi:10.1145/2335755.2335847.

[39] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, M. Handley, Data center networking with multipath tcp, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX, ACM, New York, NY, USA, 2010, pp. 10:1–10:6.

[40] R. Immich, E. Cerqueira, 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, ACM, New York, NY, USA, 2013, pp. 981–988. doi:10.1145/2480362.2480550.
URL http://doi.acm.org/10.1145/2480362.2480550

[41] C. Lal, V. Laxmi, 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, ACM, New York, NY, USA, 2012, pp. 262–268. doi:10.1145/2345396.2345440.
URL http://doi.acm.org/10.1145/2345396.2345440

[42] S. Mao, X. Cheng, Y. Hou, H. Sherali, Multiple description video multicast in wireless ad hoc networks, Mobile Networks and Applications 11 (1) (2006) 63–73. doi:10.1007/s11036-005-4461-5.
URL http://dx.doi.org/10.1007/s11036-005-4461-5

[43] P. Key, L. Massoulié, D. Towsley, Path selection and multipath congestion control, Commun. ACM 54 (1) (2011) 109–116. doi:10.1145/1866739.1866762.
URL http://doi.acm.org/10.1145/1866739.1866762

[44] B. Wang, W. Wei, Z. Guo, D. Towsley, Multipath live streaming via tcp: Scheme, performance and benefits, ACM Trans. Multimedia Comput. Commun. Appl. 5 (3) (2009) 25:1–25:23. doi:10.1145/1556134.1556142.
URL http://doi.acm.org/10.1145/1556134.1556142

[45] K. G. S. Madsen, L. Su, 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, ACM, New York, NY, USA, 2013, pp. 313–318. doi:10.1145/2488222.2488281.
URL http://doi.acm.org/10.1145/2488222.2488281

[46] M. Isard, M. Budiu, Y. Yu, A. Birrell, 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, ACM, New York, NY, USA, 2007, pp. 59–72. doi:10.1145/1272996.1273005.

[47] I. Botan, G. Alonso, P. M. Fischer, D. Kossmann, 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, ACM, New York, NY, USA, 2009, pp. 934–945. doi:10.1145/1516360.1516467.

URL http://doi.acm.org/10.1145/1516360.1516467

[48] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, 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, ACM, New York, NY, USA, 2008, pp. 1123–1134. doi:10.1145/1376616.1376729.
URL http://doi.acm.org/10.1145/1376616.1376729

[49] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, 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, ACM, New York, NY, USA, 2006, pp. 27–37. doi:10.1145/1289612.1289615.
URL http://doi.acm.org/10.1145/1289612.1289615

[50] C. Balkesen, N. Tatbul, 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, ACM, New York, NY, USA, 2013, pp. 15–26. doi:10.1145/2488222.2488258.
URL http://doi.acm.org/10.1145/2488222.2488258

[51] F. Farag, M. Hammad, 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, 2010, pp. 9–16. doi:10.1145/1871902.1871905.
URL http://doi.acm.org/10.1145/1871902.1871905

[52] B. Gedik, 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, ACM, New York, NY, USA, 2006, pp. 419–430. doi:10.1145/1142473.1142521.

[53] M. Lindeberg, V. Goebel, 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, 2010, pp. 459–468. doi:10.1145/1743384.1743466.
URL http://doi.acm.org/10.1145/1743384.1743466