2016 IEEE/ACM Symposium on Edge Computing

SpanEdge: Towards Unifying Stream Processing over Central and Near-the-Edge Data Centers

Hooman Peiro Sajjad*, Ken Danniswara*, Ahmad Al-Shishtawy†, and Vladimir Vlassov*
*KTH Royal Institute of Technology
Stockholm, Sweden
{shps,kend,vladv}@kth.se
†Swedish Institute of Computer Science (SICS)
Stockholm, Sweden
ahmad@sics.se

***Abstract*—In stream processing, data is streamed as a continuous flow of data items, which are generated from multiple sources and geographical locations. The common approach for stream processing is to transfer raw data streams to a central data center that entails communication over the wide-area network (WAN). However, this approach is inefficient and falls short for two main reasons: i) the burst in the amount of data generated at the network edge by an increasing number of connected devices, ii) the emergence of applications with predictable and low latency requirements. In this paper, we propose SpanEdge, a novel approach that unifies stream processing across a geo-distributed infrastructure, including the central and near-the-edge data centers. SpanEdge reduces or eliminates the latency incurred by WAN links by distributing stream processing applications across the central and the near-the-edge data centers. Furthermore, SpanEdge provides a programming environment, which allows programmers to specify parts of their applications that need to be close to the data source. Programmers can develop a stream processing application, regardless of the number of data sources and their geographical distributions. As a proof of concept, we implemented and evaluated a prototype of SpanEdge. Our results show that SpanEdge can optimally deploy the stream processing applications in a geo-distributed infrastructure, which significantly reduces the bandwidth consumption and the response latency.**

***Keywords*—geo-distributed stream processing; geo-distributed infrastructure; edge computing; edge-based analytics**

I. Introduction

Stream processing plays an important role in the area of Big Data analytics. It enables the analysis of large volumes of data as soon as the data is available, and supports real-time decision-making. In stream processing, data is streamed as a continuous flow of data items that is generated from multiple sources and geographical locations. A simple example of a data stream is the sequence of temperature values emitted from a weather sensor at a fixed rate. Other examples include motion detection information sent from surveillance cameras, traffic information, server logs, user clicks, and data generated from Internet of Things. A stream processing application can continuously analyse and extract useful information from the data streams emitted from one or several sources. For example, detecting trending topics in a social network, analyzing stock market, intrusion detection, and traffic management in transportation. A stream processing application is usually presented as a graph of operators (e.g., aggregations or filters), called stream processing graph.

Several stream processing systems have been developed, in both academia and industry [1]–[3], in order to provide scalable solutions for stream processing. These systems provide sophisticated application development and run-time environments. Their programming model enables to implement complex applications [4]. Their run-time systems provide a scalable execution environment by distributing the application among a cluster of machines. The existing stream processing systems, such as Apache Spark [5] and Flink [3], are designed and optimized for running in a single data center. Therefore, the common approach for the stream processing is to transfer raw data streams to a central location.

The aforementioned approach for the stream processing entails communication over wide-area network (WAN) between data sources on the network edge and stream processing applications hosted in a central data center. However, this approach is inefficient and falls short for several reasons: 1) the WAN bandwidth is expensive and can be scarce [6], 2) there has been a huge increase in the number of connected devices in the network edge (50 billion devices by 2020 [7]), which increase the network traffic dramatically, 3) applications that require predictable and low network latency can not tolerate the long communication delay in the WAN links (e.g., sensor networks that monitor the environment, smart grid, and smart urban traffic management), and 4) some data have legal constraints to a certain jurisdiction and can not be moved to another geographical area. Therefore, there is a need to avoid the data movement, to reduce the network latency and the communication over WAN as much as possible. In several distributed applications, it is known that placing applications closer to data/users can effectively decrease the network cost and the latency [8], [9]. Despite the fact that stream processing can benefit from placing the stream processing applications closer to the data sources, this approach is not well explored.

In this paper, we consider stream processing in a geo-distributed setting and at the edge that we believe will improve existing and enable novel real-time stream processing applications, including mission-critical applications. We propose SpanEdge, a novel approach that unifies stream processing over a geo-distributed infrastructure, i.e., multiple data centers located in different geographical locations. In addition to the central data centers, SpanEdge leverages the state-of-the-art solutions for hosting applications close to the network edge, such as carrier clouds [10], cloudlets [11], and Fog [12]. In SpanEdge, the data centers are categorized into two tiers, where the central data centers come in the first tier and the near-the-edge data centers are in the second tier. In SpanEdge, programmers can develop a stream processing application, regardless of the number of data sources and their geographical distribution. SpanEdge provides a programming environment, which allows programmers to specify parts of their applications that need to be close to the data sources. SpanEdge executes the application on the data centers across the two tiers in order to reduce the network latency and the cost. SpanEdge, by placing the stream processing applications close to the network edge reduces or eliminates the latency incurred by the WAN links, thus speeding up the analysis and the decision-making. In order to aggregate the results from several near-the-edge data centers, SpanEdge utilizes the central data centers to optimally place the application components in order to reduce the network latency and the cost. As a proof of concept, we implemented a prototype of SpanEdge using Apache Storm [1], an open source stream processing system.

Our contributions in this paper are as follows:

- We propose a novel approach to distribute stream processing applications across central and near-the-edge data centers in order to reduce the response latency and bandwidth consumption and thus improving the performance of stream processing applications.
- We introduce two new groupings for stream processing graphs that facilitate programmers to define the logic of stream processing applications in a complex geo-distributed environment.
- We provide a scheduler to optimally distribute the applications among central and near-the-edge data centers.
- We implement a prototype of our system on Apache Storm, an open source stream processing system, and evaluate it in an emulated environment.

The rest of this paper is organized as follows. In section II, we provide the required background information. In section III, we explain SpanEdge, the task groupings, and the scheduler. In section IV, we explain our implementation of the system on Apache Storm followed by evaluation in section V. In section VI, we discuss the related work. Finally, in section VII, we present conclusions and future work.

II. Background

A. Geo-distributed Infrastructure

Recent advancement in the geo-distributed infrastructure advocates a hierarchical infrastructure model, with at least two tiers. The *first tier* is built of typical *central data centers* and the *second tier* includes *near-the-edge* data centers such as cloudlets [11], central [13] and distributed [14] micro data centers, carrier clouds [10] and Fog [12]. The difference between the data centers in the first tier and second tier are two-fold: first, in their amount of compute and storage resources and second, in their network latency and cost to access the

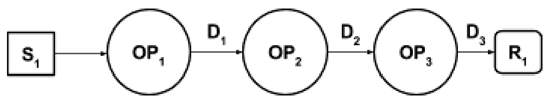

Fig. 1: An example of a stream processing graph.

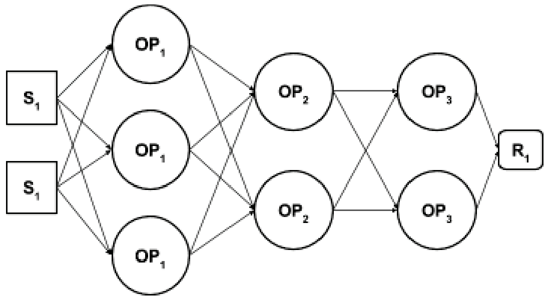

Fig. 2: An example of an execution graph.

edge. Usually, the central data centers have more compute and storage than the second tier data centers. On the other side, the second tier data centers have lower latency and network cost for accessing the network edge.

More tiers can be considered for a geo-distributed infrastructure. However, as we move toward the lower tiers, we have more limitations in the amount of computing and storage resources. In this paper, we leverage a geo-distributed infrastructure with two-tiers of central and near-the-edge data centers. We assume that near-the-edge data centers have enough resources to host stream processing applications.

B. Stream Processing

Stream processing is about processing *data streams* in real-time. A data stream is made of atomic data items each called a *tuple*. A data stream is generated from a *streaming data source* [4]. A *stream processing application* is an application that is developed for processing data stream. For the development and execution of stream processing applications, many *stream processing systems* have emerged [1]–[3], [15]–[17]. They provide sophisticated application development and runtime environments. In the rest of this section, we explain the common stream processing ecosystem, methods for the development of stream processing applications, the runtime environment for the execution of stream processing applications and categories of the applications based on their requirements.

Ecosystem: A common ecosystem for stream processing is built of a large number of heterogeneous streaming data sources that generate millions of tuples and send them to stream processing applications. However, the streaming data sources may not directly send the tuples to the stream processing applications. For example, IoT devices generate the tuples and send them through a *gateway*, which is responsible for protocol translation. Then, the gateway redirect the tuples to a *message broker* inside a data center. Finally, the broker sends the tuples to the right application. In addition, the broker buffers the incoming tuples so they can alleviate spikes on the tuple production rate for the stream processing applications. There has been several open-source [18], [19] and commercial message brokers [20], [21]. There are some trade-offs on the latency, scalability and fault tolerance among the existing message brokers.

Application Development: Stream processing systems provide a development environment that enables programmers to express the logic of their stream processing applications. A stream processing application is expressed as a graph, which is called a *stream processing graph*. Nodes of the graph include *operators*, *sources* and *sinks*. The operators can be some basic function units or some complex logic to be applied on incoming tuples. The source nodes are for reading tuples from an external system, e.g., from a message broker. The sink nodes are to send the results to an external service, such as triggering an alarm or storing the results in a database. The connections between nodes of the stream processing graph, depending on the stream processing system, can be defined explicitly or implicitly. In the explicit model, programmers connect the operators directly. In the implicit model, programmers define the operators' dependency on the types of data stream and then the stream processing system generates the graph on runtime. For example, Storm [1] has the explicit model and Flink [3] and Spark [2] support the implicit model. An example of a stream processing graph is shown in Fig. 1. In this figure, there are three operators. S_1 is a source that generates the tuples. The operator OP_1 outputs the tuple of the type D_1 and the operator OP_2 receives the tuple D_1 and transmits the tuple D_2 to the operator OP_3 and this operator outputs its results to the sink R_1.

Runtime environment: Every stream processing system provides a runtime environment in order to execute stream processing applications. A common architecture for the runtime environment is a *master-worker* architecture [1], [3], [15], [22], where the primary task of the *master* is to execute stream processing applications across the *workers*. To execute a stream processing application, the master, through a *runtime engine*, converts a given stream processing graph to an *execution graph* (Fig. 2). Each node of the execution graph is a *task* corresponding to an operator in a stream processing graph. Multiple tasks may be created for an operator in order to run in parallel. The runtime engine has a *scheduler* that allocates the tasks to run on the workers. The runtime engine gives the execution graph and the available resources on the workers to the scheduler. The scheduler, depending on the scheduling policy, allocates the tasks to different processes on the workers. The runtime engine may get the resources dynamically through a *resource allocator* such as Mesos [23] and Yarn [24]. The resource allocators enable to share the resources of a cluster among multiple frameworks. A scheduler may assign multiple tasks to one process, multiple processes to one machine or may distribute multiple processes among several machines. Different placements of the tasks depend on the trade-offs between the level of fault-tolerance, load balancing, network cost, and throughput [25]. After the runtime engine schedules the tasks, they will run on the workers infinitely.

Applications: Each stream processing application has a set of requirements. Some stream processing applications require to preserve the order of events with respect to the time they occurred not the time they are processed. Some stream processing applications require to process very large data being streamed. Stream processing applications could require to process streams from multiple streaming data sources of different types. The streams may come from different geographical locations. Some stream processing applications could be highly latency sensitive. Some data have privacy restrictions to a specific geographical boundary and that should be considered in the development of a stream processing application.

III. SpanEdge Overview

SpanEdge is a novel approach that unifies the stream processing across a geo-distributed infrastructure, including the central and near the edge data centers. In the rest of this section, first, we describe the architecture of SpanEdge and how it works in a geo-distributed infrastructure. Second, we present the two new groupings of the operators for defining a stream processing graph. Finally, we describe the scheduler of SpanEdge, which can manage the execution of stream processing graphs across a geo-distributed infrastructure.

A. Architecture

SpanEdge has a master-worker architecture, which consists of a *manager* as the master and several *workers*. The architecture is demonstrated in Fig. 3. The manager receives the stream processing applications and schedules them among the workers as tasks (Section III-C). A worker consists of a cluster of *compute nodes* and executes tasks assigned to it by the manager. We define two types of workers: the *hub-worker* and the *spoke-worker*, where a hub-worker resides in a data center in the first tier and a spoke-worker resides in a data center in the second tier near the edge. We name the hub-worker and the spoke-worker after their conceptual similarity with the hub and spoke model. The data centers, which host the workers, are connected through Wide Area Network (WAN). Since the data centers are geographically distributed, the network communication cost and the latency between them are correlated with their geographical locations and their underlying WAN connection [13]. Therefore, the geographical proximity of the data centers is a good estimate for the network latency and the cost among the workers. In this paper, there is no functional difference between a hub-worker and a spoke-worker. They only differ in their proximity to the network edge, which SpanEdge exploits in the deployment of the stream processing applications.

In SpanEdge, there are two types of communications: i) the *system management* communication (worker-manager), ii) the *data transfer* communication (worker-worker). The system management communication between the workers and the manager is for scheduling the tasks and ensuring that the tasks are running as expected. The data transfer communication is the actual data streams that the tasks dispatch to each other for processing.

An agent runs within each worker for doing the system management operations. The agent's role is to send/receive the management data to/from the manager. The agent constantly monitors the compute nodes and ensures that the given tasks run properly. The agent periodically sends the worker's status to the manager as heartbeat messages. SpanEdge uses the heartbeat messages for the failure detection of the workers. In case a compute node fails, the agent can detect it and restart the tasks in another compute node.

We designed SpanEdge for an ecosystem with many heterogeneous streaming data sources that are geographically dispersed, i.e., i) the stream sources can differ in type, e.g., weather or traffic sensors, and ii) there can be several number

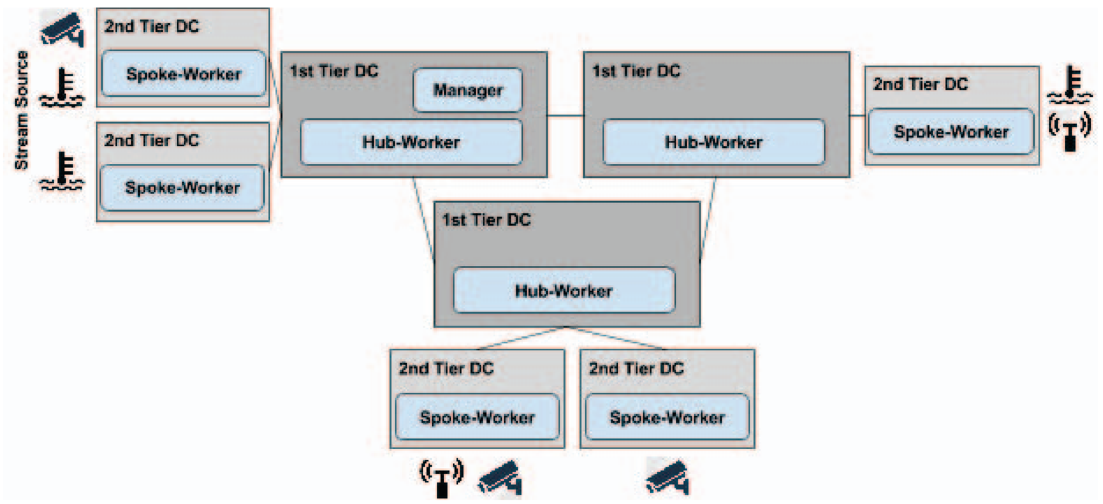

Fig. 3: The architecture of SpanEdge across a two-tier geo-distributed infrastructure.

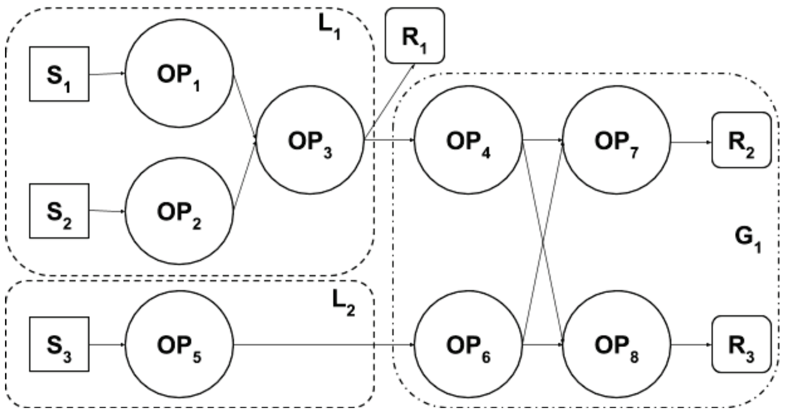

Fig. 4: An example of the local-task and the global-task.

of instances of each source type in different geographical areas. We assume that the streams are redirected from their sources to the nearest second tier data centers.

B. Task Grouping

SpanEdge enables programmers to group the operators of a stream processing graph either as a *local-task* or a *global-task*, where a local-task refers to the operators that require to be close to the streaming data sources and a global-task refers to the operators that process the results of a group of local-tasks.

The operators grouped in a local-task are placed close to the sources from which they consume data. SpanEdge creates a replica of a local-task at each spoke-worker with the corresponding data source types. Note that, SpanEdge assigns the tasks to the spoke-workers only if all the sources in a local-task are available. Otherwise, it will assign the tasks to a hub-worker. In Fig. 4, we demonstrate a high level view of an example stream processing graph. In this example, there are three types of the streaming data sources S_1, S_2 and S_3. Every local-task should contain at least one streaming data source. In this example, we group OP_1, OP_2 and OP_3 as a local-task L_1 accordingly with the sources S_1 and S_2. As it can be seen, OP_3 generates a partial result and transmits it to the sink R_1. This indicates that R_1 receives the results as soon as OP_3 produces a tuple. In Fig. 5, we show how SpanEdge schedules and deploys this stream processing graph in a particular infrastructure. We explain the scheduler in detail in Section III-C.

Grouping a set of operators as a global-task indicates that the run-time must schedule the operators on a single hub-worker. The scheduler selects a hub-worker optimally in order to decrease the network latency and the cost (Section III-C). Programmers can use the global-task grouping for processing the data generated from the local-tasks. For example, it can be an aggregation over the data generated by some local-tasks. In Fig. 4, the global operators OP_4 and OP_6 receive the data from the local operators OP_3 and OP_5. These operators receive the results from all replicas of the local operators.

The operators grouping provides a general and an extensible model in order to develop any arbitrary operations. The notions of local and global groupings enable programmers to instruct the scheduler with respect to the partial and the aggregated results. Programmers can consider that by selecting

Algorithm 1 Main logic of the scheduler

```
INPUT:
graph                                        ▷ Stream Processing Graph
sMap             ▷ Map of streaming data sources to spoke-workers
topology                                       ▷ Topology of workers

1: procedure SCHEDULE(Graph graph, Map sMap, Topology topology)
2:     lwMap ← assignLocalTasks(graph, sMap);
3:     gwMap ← assignGlobalTasks(lwMap, graph, topology);
4:     twMap.add(lwMap);
5:     twMap.add(gwMap);
6: return twMap;
```

Algorithm 2 Assign the local-tasks to the spoke-workers.

```
INPUT:
graph                                        ▷ Stream Processing Graph
sMap             ▷ Map of streaming data sources to spoke-workers

 1: procedure ASSIGNLOCALTASKS(Graph graph, Map sMap)
 2:     lTasks ← graph.getLocalTasks();
 3:     for each l ∈ lTasks do
 4:         sNodes ← l.getSourceNodes();
 5:         for each s ∈ sNodes do
 6:             workers ← findWorkersWithSource(sMap, s);
 7:             workerSet.add(workers);
 8:         end for
 9:         lwMap.put(l, workerSet);
10:     end for
11: return lwMap;
```

a group of operators as a local-task, those operators exploit the proximity of the sources to generate low latency results. However, the (*partial*) results are based on the data generated from the sources close to a specific geographical location. In order to create the aggregated results based on the whole data processed in different geographical locations, programmers can select a group of the operators as the global-task in order to aggregate the partial results.

C. Scheduler

In SpanEdge, the manager receives a stream processing graph to be executed in a geo-distributed infrastructure. The manager employs the scheduler, which converts the stream processing graph to an execution graph by assigning the tasks to the hub-workers and the spoke-workers. Each node of the execution graph represents a task and each connection indicates the flow of data among the tasks.

The scheduler requires some information in order to assign the tasks to the workers: i) a stream processing graph, ii) a map of the streaming data sources to the spoke-workers, iii) the network topology between the workers. The scheduler leverages the map of streaming data sources to the spoke-workers in order to deploy the local-tasks in all the spoke-workers with the required data source types. We assume that the map is provided by a *source discovery service*. The network topology of workers can be either dynamically generated based on a network monitoring service or it can be based on the geographical location of their host data centers.

The scheduler assigns the tasks to the workers in two steps (Algorithm 1). First, it assigns the local-tasks to the spoke-workers and second, it assigns the global-tasks to the hub-workers. In Algorithm 2, we demonstrate the assignment

Algorithm 3 Assign the global-tasks to the hub-workers

```
INPUT:
graph                                        ▷ Stream Processing Graph
lwMap                       ▷ Map of local-tasks to spoke-workers
topology                                       ▷ Topology of workers

 1: procedure ASSIGNGLOBALTASKS(Map lwMap, Graph graph, Topology
    topology)
 2:     gTasks ← graph.getGlobalTasks();
 3:     for each g ∈ gTasks do
 4:         lTasks ← g.getLocalTasks();
 5:         for each l ∈ lTasks do
 6:             workers ← lwMap.get(l);
 7:             workerSet.add(workers);
 8:         end for
 9:         hWorker ← findClosestHubWorker(workerSet, topology);
10:         w ← assignTaskToWorker(g, hWorker);
11:         gwMap.put(g, w);
12:     end for
13: return gwMap;
```

of the local-tasks to the spoke-workers. In this algorithm, the scheduler gets all the local-tasks from the graph. It goes through each local-task and retrieves the source nodes. Each source node represents the type of a streaming data source. The scheduler uses the type of source nodes to list a set of spoke-workers that are closest to streaming data sources. To do this, the scheduler finds each source node in the map of stream sources to spoke-workers. In the end of this step, the scheduler assigns the local-task to all the workers in the created set. In other words, the scheduler replicates a local-task in all the data centers that are close to the streaming data sources. By assigning all the local-tasks, the scheduler has a map of the local-task to the spoke-workers. This enables the scheduler to place the global-tasks more wisely. The scheduler chooses a hub-worker that is closest to the corresponding spoke-workers in the given topology of workers (line 9 in Algorithm 3). As we present in Algorithm 3, the scheduler iterates through all the global-tasks inside the graph. For each global-task, it iterates through all its adjacent local-tasks and creates a set of their assigned spoke-workers. The scheduler can find the nearest hub-worker to the set of spoke-workers by having a set of assigned spoke-workers and the network topology of workers. Finally, the scheduler assigns the global-task to the selected hub-worker.

After assigning all the local-tasks and the global-tasks, the scheduler returns a map of the tasks to the workers. SpanEdge, using its runtime engine, distributes the tasks (the binaries from compiling the source codes) among the workers according to the output generated from the scheduler.

IV. IMPLEMENTATION

We implemented a prototype of SpanEdge[1] by extending Apache Storm [1]. We choose Apache Storm because its low-level API for stream processing makes it more flexible and intuitive to evaluate our prototype. In the Storm's terminology, the term *spout* is used to refer to a source and the term *bolt* is used to refer to an operator or a sink.

Apache Storm has a master-slave architecture. The master node (called Nimbus), schedules the tasks over the slave

[1]The source code is available at https://github.com/Telolets/StormOnEdge

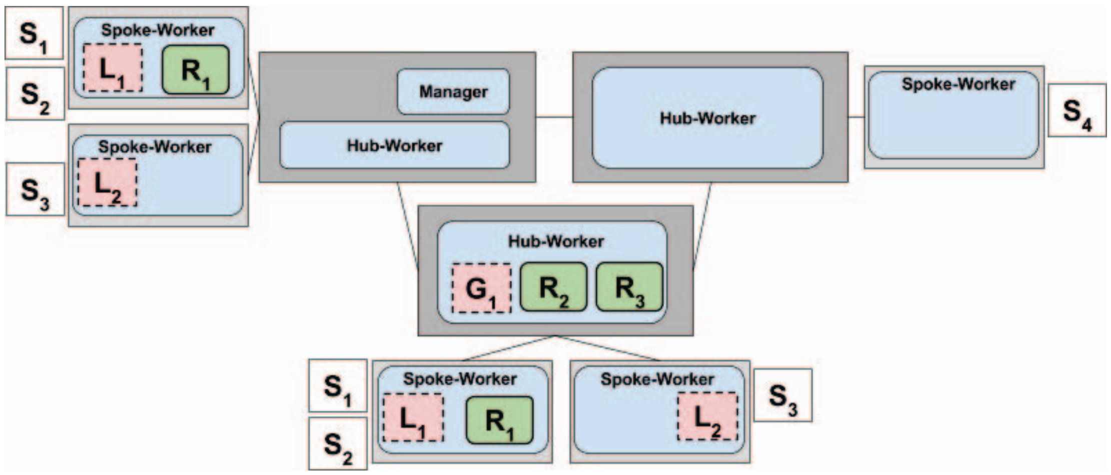

Fig. 5: A deployment of the example application in SpanEdge.

nodes (called the worker nodes). Apache Storm provides fault-tolerance and reliability guarantees through sending heartbeats and ack messages. We implement the prototype by having one Storm cluster across a geo-distributed infrastructure. We evaluated Apache Storm's performance when distributed over a set of machines connected through a high latency network [26]. Our evaluation shows that the heartbeat and ack messages have no significant network cost. However, increasing the network latency delays the detection of the workers failure. Another approach to implement SpanEdge is by having multiple instances of Apache Storm cluster each deployed in a data center. Each instance has its own local manager, which are all coordinated by a higher level manager. Also, each local manager can handle the failure detection in their cluster. Discussion over the fault-tolerance and reliability guarantees are beyond the scope of this paper and we aim it as a future work.

We implemented the scheduler of SpanEdge as a plugin in Apache Storm. Creating a custom scheduler in Storm is based on implementing the IScheduler interface available in the storm-core library. The scheduler is executed by Nimbus. The topology is defined in Storm by creating an instance of the class TopologyBuilder. An instance of TopologyBuilder is given to the scheduler as a stream processing graph. We define the local-tasks and the global-tasks by the addConfiguration method. This method receives a key and a value as the input parameters. The key specifies the type of task grouping, local-task or global-task. The value is an arbitrary name to refer to a specific task group. All the spouts and bolts under the same group name will be considered as one local-task or global-task. Storm can create a configurable number of parallel tasks for each bolt and spout. The way that the data is distributed among the parallel tasks are defined by the stream groupings. We implemented a new grouping called *zone-grouping*, in order to avoid the data transfer among the parallel tasks of two different data centers. We implement the zone-grouping by extending the CustomStreamGrouping abstract class available in the Storm's library.

An example topology is shown in Listing 1. In this example, we want to process the data generated from some temperature sensors. We create a spout ($tSpout$) as a data source and add it to the local-task $L1$. We want to have two operators, one to process the temperature data in order to detect anomalies and the other to aggregate the anomalies statistics. The anomalies in the temperature can be detected by placing the operator close to the data source. Therefore, we create a bolt $lBolt$ and add it to the same local-task as $tSpout$. For the aggregation, we define a bolt $aBolt$ and add it to a global-task $G1$. The integer numbers in lines 3, 6 and 10 specify the number of parallel instances of the spouts and the bolts. For example, in line 3, we specify 4 parallel instances of the spout $tSpout$. The shuffleGrouping in line 7 indicates that the data from the spout instances should be uniformly distributed among the instances of $lBolt$.

```
1  TopologyBuilder builder = new TopologyBuilder();
2  . . .
3  builder.setSpout("temperatureSpout", tSpout, 4)
4      .addConfiguration("local-task", "L1");
5  ...
6  builder.setBolt("localTempBolt", lBolt, 2)
7      .shuffleGrouping("temperatureSpout")
8      .addConfiguration("local-task", "L1");
9  . . .
10 builder.setBolt("aggregateBolt", aBolt, 4)
11     .shuffleGrouping("localTempBolt")
12     .addConfiguration("global-task", "G1");
```

Listing 1: An example of defining a local-task and a global-task in the SpanEdge prototype.

V. Evaluation

In this section, we evaluate the performance of SpanEdge based on the prototype implementation. We start by describing the experimental setup followed by the evaluation criteria and the evaluation results. We compare SpanEdge against a central data center deployment architecture.

A. Experimental Setup

In order to evaluate SpanEdge in different deployment scenarios, we use the CORE [27] network emulator to emulate a geo-distributed infrastructure with both central and near-the-edge data centers. The setup we use in our experiments is depicted in Fig. 6. The setup consists of 2 central data centers (shown as squares in Fig. 6) and 9 near-the-edge data centers

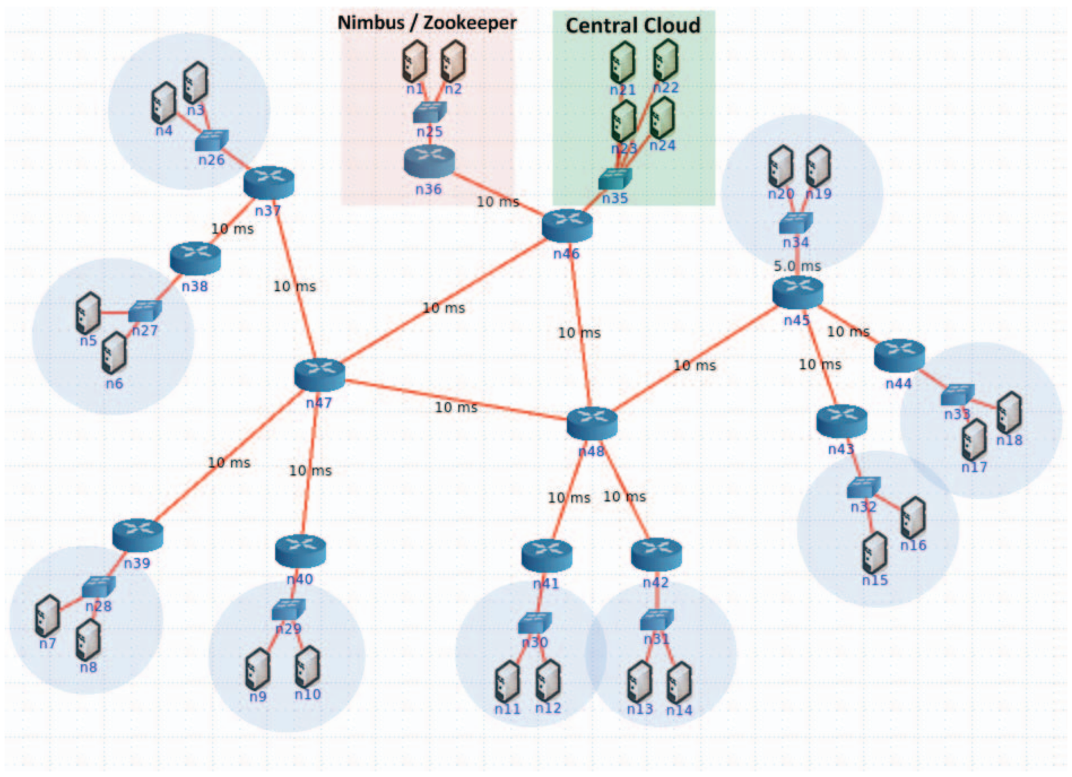

Fig. 6: The geo-distributed infrastructure used in the experiments implemented in the CORE network emulator

(shown as circles in Fig. 6). For the streaming data sources, we use two types of data streams (type A and type B). Each near-the-edge data center can provide none, one, or both types of data streams. We vary the total number of data stream sources as detailed in the experiments. Each instance of a data source type is set to generate a 500 bytes tuple with a constant rate of 450 tuples per second.

The CORE network emulator can be used to emulate various network components, such as routers, switches, servers, and network links between them. The CORE network emulator uses Linux containers to run the same software used in a real deployment of such routing protocols and server software. It also emulates the network latency and bandwidth for the links between the different components.

We deploy the SpanEdge prototype in the emulated environment by deploying the various software components inside the server containers provided by the CORE network emulator. The main components include the master (Nimbus and ZooKeeper), which we deploy in one of the central data centers, and the workers, which we deploy in both the central and the near-the-edge data centers. Our proposed geo-aware scheduler is deployed as a plugin for Nimbus (see Section IV). For each experiment, we provide a map (see Section III-C) that contains the data streams available at each of the near-the-edge data centers.

We compare SpanEdge against a standard central deployment architecture for stream processing systems. In a central deployment, both the master and the worker reside in the same central data center. However, the streaming data sources are still geo-distributed and the data streams need to be transferred to the central data center for processing.

Our evaluation is inspired by Yahoo's Storm performance test [28]. The graph of the stream processing application used throughout the experiments is depicted in Fig. 7. The stream processing application requires two types of data streams (type A and type B), produces two local results (the local result A and the local result B) from the two local tasks, and one global result (the global result AB) from the global task. Since preforming computationally intensive processing of the streams is not feasible in an emulated environment and would impact the latency in an unrealistic manner, we emulated the stream processing by having each bolt randomly dropping 40% of the incoming tuples. This emulates a typical stream processing operator such as filtering or aggregation where the output stream rates are usually smaller than the input stream rates. Our geo-aware scheduler uses the stream processing graph together with the map of available data stream sources to generate an execution graph that will assign the tasks to the workers. The local tasks will be replicated and assigned to the workers in the near-the-edge data centers with the required data stream sources. The global task will be assigned to a worker in the best central data center in terms of the latency to the other workers.

B. *Performance Evaluation Experiments*

To evaluate the performance of SpanEdge, we measure the overall bandwidth consumption and the average response

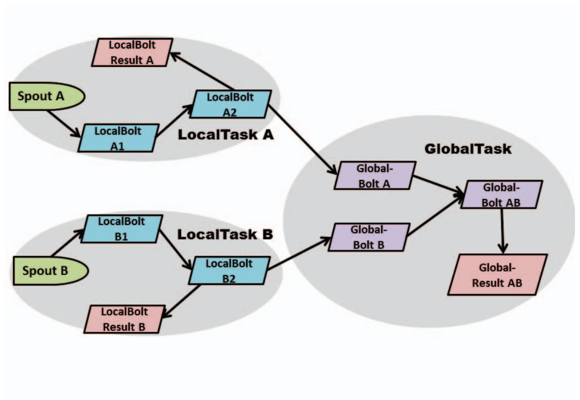

Fig. 7: The stream processing graph representing the logic of application used in the experiments.

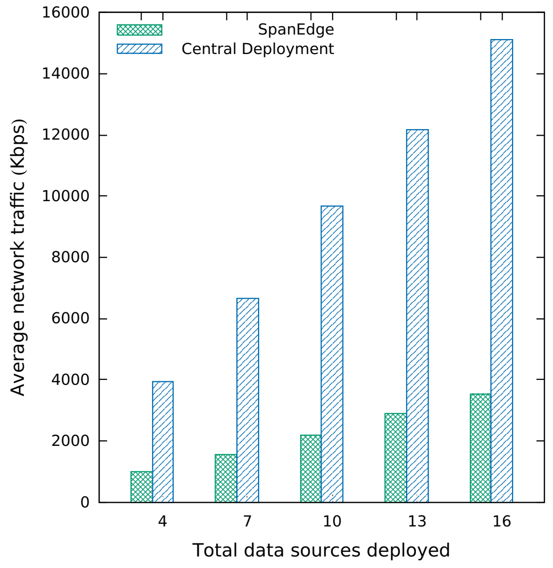

Fig. 8: The overall bandwidth consumption.

latency (tuple processing latency). The overall bandwidth consumption is the sum of the bandwidth consumed in transferring the data between each pair of the data centers involved in the stream processing application. The average tuple processing latency is the average time required to process a tuple measured from the time it is generated at the data stream until a result is produced. In SpanEdge, we distinguish between the local and the global results. Thus, we measure the average tuple processing latency for both the local and the global results.

1) Overall Bandwidth Consumption: In this experiment, we compare the overall bandwidth consumption for SpanEdge versus a central deployment. We fix the data generation rate for each streaming data source while varying their number. We increase the number of data sources from 4 to 16. The data sources are either Type A or Type B. By increasing the number of data sources, we increase the geographical distribution and the amount of data that needs to be processed.

The results, depicted in Fig. 8, shows that SpanEdge significantly reduces the overall bandwidth consumption between the data centers. This is because, in the case of a central deployment, the raw streaming data needs to be transferred to the central data center. While in the case of SpanEdge, the data before being transferred between the data centers is already processed near-the-edge (by the spoke-worker).

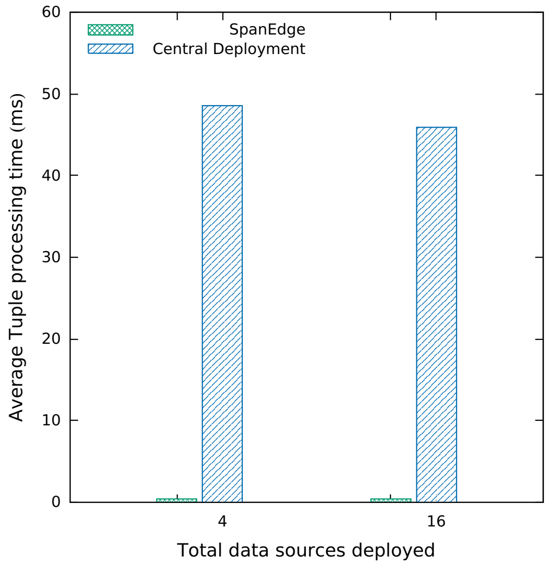

Fig. 9: The average tuple processing time (the local result).

2) Average Response Latency: In this experiment, we measure the average response latency for producing a result both for the case of local results and global results.

The average response latency for producing a local result is depicted in Fig. 9. In the case of SpanEdge, the latency is very low. This is because the stream processing computation is near the data stream and the consumer of the result. Thus, no WAN communication is needed for computing the local result. However, in the case of a central deployment, the high response latency is mainly due to the round trip needed to transfer the data stream to a central location and then sending the results back.

The average response latency for producing a global result is depicted in Fig. 10. The results show that both SpanEdge and the central deployment have the same average response latency. This is expected as in both cases the data is traveling through the same network path. However, note that SpanEdge achieves this latency while consuming less bandwidth as shown in the overall bandwidth consumption experiment. Thus, we expect SpanEdge to have a better response latency in scenarios where the bandwidth is the bottleneck.

VI. Related Work

We present the related work in four categories: 1) works that have been done in introducing novel stream processing systems, 2) researches on stream processing with respect to geo-distribution, 3) researches on improving the scheduler for stream processing systems and 4) works on distributed algorithms for stream processing.

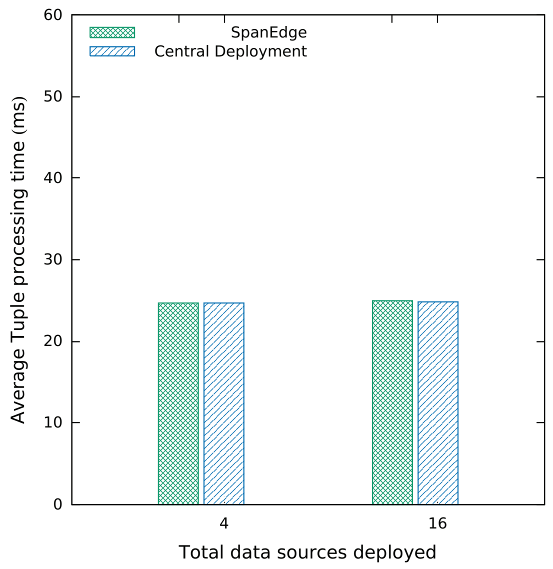

Fig. 10: The average tuple processing time (the global result).

A. *Stream Processing Systems*

Aurora [29] and its distributed descendant Borealis [30], TelegraphCQ [31], and STREAM [32] are the very first streaming databases. In these systems, they address the major problems in traditional data management systems with respect to management of unbounded streams of data. These systems provide extensions to SQL language to fit continuous queries and their system model is based on processing streams of data in real-time. Their successful solutions have paved the way for the next generation of large-scale stream processing systems in industry. Flink Streaming [3], Spark Streaming [2], Storm [1], Heron [15] and MillWheel [16] are some of the existing systems that has been developed to be used within data centers. Our work is mainly inspired by studying these existing systems. Our contribution to the existing stream processing systems is that we propose an approach to extend a stream processing system beyond a single data center. Our approach can extend the existing stream processing system to deploy stream processing applications across multiple data centers.

B. *Geo-distributed Streaming Data Analytics*

There have been some researches on streaming data analytics considering diverse geographical locations for streaming data sources. In [33] and [34], the authors try to propose solutions for placement of the operators over a set of computational resources scattered in a wide-area network. However, due to the recent advancements in the geo-distributed infrastructure, we expect a combination of two layers of central data centers and near-the-edge infrastructure. JetStream [35] is another stream processing system across a wide-area network. The system provides adaptive filtering and data aggregation that can adjust the transferred data over wide-area network according to the available bandwidth. However, JetStream's query model does not support programming arbitrary algorithms. It is entirely based on aggregation and approximation. In SpanEdge, we provide a more general approach for programming stream processing applications regardless of the underlying technology. In [36], the authors focus on optimization of the grouped aggregation in hub-and-spoke model across data centers. However, they do not provide any solution for development, placement, and scheduling of stream processing applications across data centers.

C. *Stream Processing Scheduler*

There are some researches on improving the scheduler for stream processing systems. In [25] and [37], the authors try to improve the default scheduler in Storm [1], which is a simple round-robin scheduler. Their goal is to provide a network-aware scheduler that can reschedule tasks based on the network traffic. In both of these works, the focus is on a improving the network performance in a single data center. However, in this paper, we try to improve scheduling across several data centers with respect to the location of data sources. We expect that the state-of-the-art solutions to improve scheduling inside a data center can be employed in SpanEdge for an optimal scheduling of the tasks assigned to each each data center.

D. *Distributed algorithms for Streaming Data*

There has been several studies done in the area of distributed data mining and algorithms to process distributed streaming data. Cormode et al. [38] and Rodrigues et al. [39] propose distributed algorithms to cluster streaming data without aggregating all data in a central location. There has been also distributed algorithms for outlier detection [40], detection of denial of service attacks [41], mining frequent items [42] and classification [43]. These algorithms try to decrease the network communication between the computation nodes, which are suitable for environments with distributed streaming datas. One of our contribution is to provide an approach that enables programmers to implement these algorithms in a geo-distributed infrastructure with a standard programming language and stream processing graph.

VII. Conclusions and Future Work

In this paper, we addressed the problems with the current approach for stream processing, which includes transferring raw data streams from the network edge to a central data center. We proposed SpanEdge, a novel approach to unify stream processing across the central and the near-the-edge data centers. We discussed that SpanEdge can utilize the near-the-edge data centers in order to reduce the network communication over the WAN links and consequently, to avoid the incurred network latency. We explained that in SpanEdge, programmers can develop a stream processing application, regardless of the number of data sources and their geographical distributions. We achieved this by introducing two new task groupings, which enables programmers to specify the parts of their application that should run close to the data sources. We presented the geo-aware scheduler that processes the stream processing graphs and distributes the application components optimally. We implemented a prototype of SpanEdge as a proof of concept by augmenting the Apache Storm stream processing system. We evaluated our work in an emulated environment using the CORE network emulator. We demonstrated that

SpanEdge can optimally deploy the stream processing applications in a geo-distributed infrastructure, which significantly reduces the bandwidth consumption and the response latency.

Encouraged by the results, we are interested in extending our solution in different dimensions including: i) enabling the geo-aware scheduler to dynamically adapt changes in the network conditions and the available resources on the edge, ii) taking into account the mobility of devices and their state migration, and iii) research on more sophisticated fault-tolerance mechanisms.

ACKNOWLEDGEMENT

The authors would like to thank Prof. Klara Nahrstedt and the anonymous reviewers for their valuable comments to improve quality of the paper. This research is supported by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research under the contract RIT10-0043 and the FFI - Big Automotive Data Analytics (BADA) project funded by Vinnova under the contract 2015-00677.

REFERENCES

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@Twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 147–156. [Online]. Available: http://dx.doi.org/10.1145/2588555.2595641

[2] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[3] (2016) Apache flink. [Online]. Available: https://flink.apache.org/

[4] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[5] (2016) Apache spark. [Online]. Available: http://spark.apache.org/streaming/

[6] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[7] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, vol. 1, pp. 1–11, 2011.

[8] K. Church, A. G. Greenberg, and J. R. Hamilton, "On delivering embarrassingly distributed cloud services." in *HotNets*. Citeseer, 2008, pp. 55–60.

[9] D. Fesehaye, Y. Gao, K. Nahrstedt, and G. Wang, "Impact of cloudlets on interactive mobile cloud applications," in *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*. IEEE, 2012, pp. 123–132.

[10] T. Taleb, "Toward carrier cloud: Potential, challenges, and solutions," *Wireless Communications, IEEE*, vol. 21, no. 3, pp. 80–91, 2014.

[11] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14–23, 2009.

[12] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[13] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[14] H. P. Sajjad, F. Rahimian, and V. Vlassov, "Smart partitioning of geo-distributed resources to improve cloud network performance," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*. IEEE, 2015, pp. 112–118.

[15] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://dx.doi.org/10.1145/2723372.2742788

[16] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[17] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[18] (2016) Rabbitmq. [Online]. Available: https://www.rabbitmq.com/

[19] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing." NetDB, 2011.

[20] (2016) Google cloud pub/sub. [Online]. Available: https://cloud.google.com/pubsub/docs

[21] (2016) Azure event hub. [Online]. Available: https://azure.microsoft.com/en-us/services/event-hubs/

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[25] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 207–218.

[26] K. Danniswara, H. P. Sajjad, A. Al-Shishtawy, and V. Vlassov, "Stream processing in community network clouds," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pp. 800–805.

[27] (2016) Common open research emulator (core). [Online]. Available: http://www.nrl.navy.mil/itd/ncs/products/core

[28] (2016) Yahoo's storm performance test. [Online]. Available: https://github.com/yahoo/storm-perf-test

[29] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[30] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine." in *CIDR*, vol. 5, 2005, pp. 277–289.

[31] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 668–668.

[32] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," *Book chapter*, 2004.

[33] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 2006, pp. 49–49.

[34] J.-H. Hwang, U. Çetintemel, and S. Zdonik, "Fast and reliable stream processing over wide area networks," in *Data Engineering Workshop,*

2007 IEEE 23rd International Conference on. IEEE, 2007, pp. 604–613.

[35] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, 2014, pp. 275–288.

[36] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing grouped aggregation in geo-distributed streaming analytics," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 133–144.

[37] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: traffic-aware online scheduling in storm," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 535–544.

[38] G. Cormode, S. Muthukrishnan, and W. Zhuang, "Conquering the divide: Continuous clustering of distributed data streams," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 1036–1045.

[39] P. P. Rodrigues, J. Gama, and J. P. Pedroso, "Hierarchical clustering of time-series data streams," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 20, no. 5, pp. 615–627, 2008.

[40] M. E. Otey, A. Ghoting, and S. Parthasarathy, "Fast distributed outlier detection in mixed-attribute data sets," *Data Mining and Knowledge Discovery*, vol. 12, no. 2-3, pp. 203–228, 2006.

[41] W. Lee, R. A. Nimbalkar, K. K. Yee, S. B. Patil, P. H. Desai, T. T. Tran, and S. J. Stolfo, "A data mining and cidf based approach for detecting novel and distributed intrusions," in *Recent Advances in Intrusion Detection*. Springer, 2000, pp. 49–65.

[42] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 346–357.

[43] R. Chen, K. Sivakumar, and H. Kargupta, "An approach to online bayesian learning from multiple data streams," in *Proceedings of Workshop on Mobile and Distributed Data Mining, PKDD*, vol. 1. Citeseer, 2001, pp. 31–45.