Shepherd: Seamless Stream Processing on the Edge

***Abstract*—Stream processing frameworks are a popular solution for handling computation in latency-sensitive applications. However, next generation applications such as augmented/virtual reality, autonomous driving, and Industry 4.0, have tighter latency constraints and produce much larger amounts of data. To address the real-time nature and high bandwidth usage of new applications, edge computing provides an extension to the cloud infrastructure through a hierarchy of datacenters located between the edge devices and the cloud.**

Using existing stream processing frameworks out-of-the-box in this hierarchy is not efficient, as they adopt a stop-the-world approach during reconfiguration. This approach can lead to stalls on the order of several minutes. While costly reconfiguration was not an issue in the cloud where these events are extremely rare, edge computing is a more dynamic environment, characterized by frequent reconfigurations.

In this paper, we propose Shepherd, a new stream processing framework for edge computing. Shepherd minimizes downtime during application reconfiguration, with almost no impact on data processing latency. Our experiments show that, compared to Apache Storm, Shepherd reduces application downtime from several minutes to a few tens of milliseconds.

***Index Terms*—stream processing, reconfiguration, late binding, hierarchical edge computing, seamless**

I. Introduction

Next generation applications such as autonomous driving, augmented/virtual reality, smart technologies, interactive games, and Industry 4.0 produce massive scales of data that must be analyzed in a timely fashion [1]. Stream processing frameworks are often used to address this need [2].

Stream processing applications are often structured as a dataflow graph. Vertices can be sources that generate streams of data tuples, or operators that execute a function over incoming data streams. Sinks, are a special type of vertices that consume the processed data but are terminal and represent the end of the flow. Traditionally, all application components are placed in the cloud to take advantage of powerful data centers. Unfortunately, this approach is not compatible with next generation applications, since sending data over wide-area links to the cloud results in high bandwidth usage and high application latency.

Edge computing expands cloud computing with a hierarchy of computational resources located along the path between the edge and the cloud [1], [3]. In this paper, we argue that efficient use of edge computing for stream processing requires support for seamless reconfiguration and deployment of application operators without disrupting application execution. In particular, throughput should be stable, and latency should not spike during the reconfiguration. Seamless reconfiguration is required to enable efficient resource sharing between applications running on edge datacenters. The smaller size of edge datacenter leads to higher costs for storage and computation relative to the cloud (e.g., AWS Wavelength [4] is 40% more expensive than EC2). It is therefore impractical (and may not be even possible due to limited resources) to run all applications continuously on the edge.

Figure 1 illustrates the benefits of dynamic reconfiguration for an application running on a hierarchy of edge data centers with three levels: cloud, region, and city. The application provides analytics about traffic on city roads by performing object detection on video frames produced by a network of motion-activated street cameras. The application consists of a sequence of three operators: [F], a frame filter operator that removes frames that do not significantly change compared to the previous frame; [O], an object detector operator; and finally, [A], an aggregation operator that computes statistics.

Initially, all operators are deployed on the cloud as the number of active cameras is small, and the network cost of transmitting the raw images to the cloud is low (Figure 1a). As more cameras become active in cities in Region 1, network traffic grows and it becomes profitable to filter frames closer to the source by deploying operator $[F_1]$ in Region 1 (Figure 1b). The original [F] operator continues to run in the cloud, where it processes traffic from Region 2. Similarly, operators [O] and [A] are not replicated and continue to run only on the cloud. Operator [O] is CPU intensive and benefits from the cheaper cloud cycles. Operator [A] aggregates data across regions and has to run on the cloud where it gets a global view. As traffic continues to grown in City 1, the application adapts by creating a new replica of the operator $[F_2]$ in this city (Figure 1c). This process is repeated with the creation of $[F_3]$ as more cameras become active in City 2. Finally, the $[F_1]$ operator in Region 1 is removed as it no longer necessary (Figure 1d).

Unfortunately, state-of-the-art frameworks, such as Apache Flink [5], Apache Spark [6], and Apache Storm [7] do not support seamless application reconfiguration. These frameworks were designed to run on the cloud, where application reconfiguration is exceedingly rare. It is therefore not surprising that reconfiguration in these frameworks is implemented as a high-latency, system-wide *stop-the-world* event that involves expensive coordination, often in the form of a global barrier. Global coordination is required because these frameworks use *early-binding* routing, where upstream and downstream operators have direct socket connections. Our experiments show that even for a small deployment of a few nodes, the stoppage time (the interval where the application stops processing data) is measured in tens of seconds and grows with the size of the deployment. Such an approach is incompatible with edge applications, which often have real-time requirements and where reconfiguration is a common occurrence.

This paper introduces Shepherd: a stream processing frame-

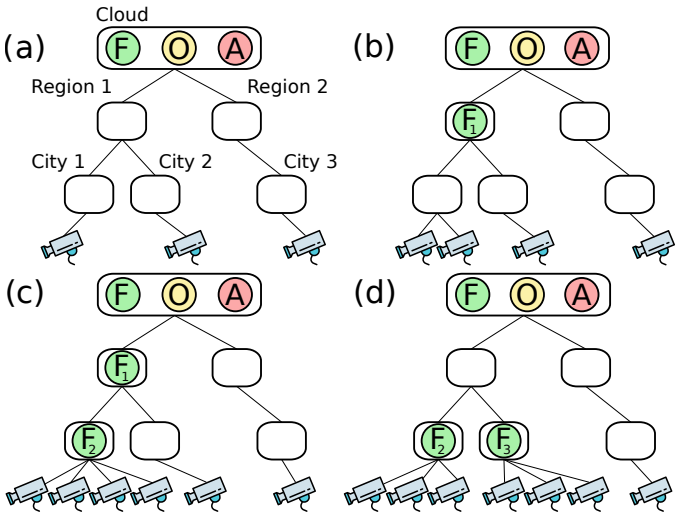

Fig. 1. A video analytics service is dynamically reconfigured to make efficient use of an edge network.

work for edge networks which enables seamless application reconfiguration with minimal stoppage time. Shepherd's architecture uses a network of software routers to transfer data tuples between operators. Shepherd implements a *late-binding* approach to tuple routing, where an operator does not know the location of the next operator that will consume the tuples it produces. Instead, tuples are *shepherded* to their destination by Shepherd's network of routers. This approach allows for flexible reconfiguration with minimal stoppage time and without requiring global coordination. As an optimization, Shepherd uses direct network connections (bypassing the router) to transfer messages between operators deployed in the same datacenter; however, this is done without compromising reconfigurability.

Shepherd currently only supports the reconfiguration of stateless operators. We argue that for a wide range of edge computing applications this is not a significant limitation as the operators that benefit the most from running close to the edge, such as the filtering operator in our example video analytics application, are naturally stateless. Relaxing this limitation, however, is the subject of our future work.

We evaluated Shepherd on a hierarchical edge network composed of several Amazon datacenters geographically distributed across North America, with emulated edge devices. Our experiments show that Shepherd reduces stoppage time by up to 97.5% compared to Apache Storm. In contrast to Apache Storm, Shepherd's stoppage time does not increase with the number of data centers in the network, network latency, or application size.

In summary, this paper makes the following contributions:

- A stream processing framework for the edge that leverages late binding routing to enable seamless application reconfiguration. We plan to open source the Shepherd implementation;
- A mechanism that reduces the effect of operator slow start;
- An evaluation on an emulated edge network using real-world wide area links.

The reminder of this paper is structured as follows: Section II provides background on existing stream processing systems focusing on their reconfiguration mechanisms; Sections III and IV describe the design and implementation of Shepherd; Section V presents our experimental evaluation; Sections VI and VII discuss related work and present our conclusions.

II. Motivation and Background

In this section, we first discuss how cloud-based stream processing frameworks, such as Apache Storm and Apache Flink, handle reconfiguration. We then discuss previous edge-based stream processing efforts and their limitations.

A. Cloud Frameworks

Stream processing frameworks use a dataflow programming model where an application is represented as a direct acyclic graph (DAG). The DAG consists of vertices that represent operators and edges that represent data streams between operators. Operators can be data sources, sinks, or user-defined functions, such as filtering, aggregation, convolution. Data tuples are produced by data sources and consumed by operators until the data reaches a data sink. In the edge computing context, the data sources are often sensors at the edges of the Internet, while the data sinks can be placed on cloud data centers as well as actuators at the network edge.

The stream processing application submitted by a user is called a *logical plan*. The stream processing framework transforms the logical plan into a *physical plan* that contains the number of physical instances of each logical operation, and their mapping to computing devices. Reconfiguration changes the physical plan by modifying the number of operator replicas or their mapping to computational resources. Reconfiguration is typically triggered as a result of changes in the application's workload and is intended to improve performance metrics such as monetary cost, bandwidth, or latency.

We evaluated the effect that reconfiguration has on the performance of an application that performs sentiment analysis and consists of 6 operators [P, F, S, T, C, E]. We run this application on an emulated 2-level edge network consisting of a cloud data center and 5 edge data centers. Attached to each edge is a separate data source that produces 400 messages per second when active, and 0 otherwise. Each emulated datacenter runs on a separate virtual machine (VM), all collocated in a single Amazon datacenter, with no additional latency added. We show in Section V that reconfiguration time is negatively affected by increasing latency, making this idealized setup a best-case scenario.

Figure 2a shows the physical plan at the start of the experiment, with operators [S, T, C, E] deployed on the cloud, and operators [P, F] deployed on edge 1 which has the only active data source. In a second stage, the data source attached to edge 2 becomes active and the application is reconfigured

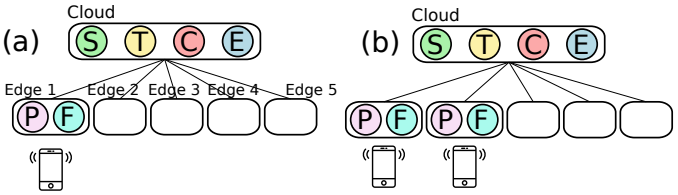

Fig. 2. Multiple physical plans used to evaluate Apache Storm's reconfigurations.

by deploying additional replicas of operators [P, F] on edge 2 (see Figure 2b). This process is repeated 3 more times until all 5 sources are active and replicas of the [P, F] operators are running on all edges.

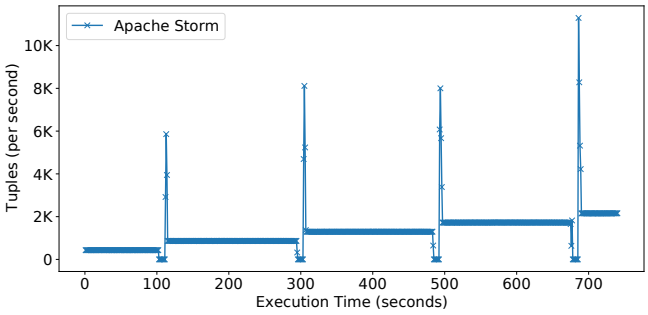

Fig. 3. Physical plan reconfiguration.

Figure 3 shows the effects on application throughput of running this experiment on Apache Storm. The figure shows that throughput increases as more data sources become active; however, each reconfiguration results in significant stoppage time when the application stops processing messages. Moreover, the time it takes the application to get back to steady state increases as the number of operator replicas grows.

Reconfiguration is a high-latency operation in cloud-based stream processing frameworks, such as Apache Storm and Apache Flink. These frameworks use a *stop-the-world* approach that uses a global barrier to empty all queues, wait for all operators to reach a stable state, make all changes, and then restart. The global coordination is required because these frameworks use *early-binding* routing, where upstream and downstream operators have direct socket connections; when a reconfiguration is performed, all socket connections are recreated. Long reconfiguration times and application stoppages are acceptable when reconfiguration is rare, such as a typical cloud environment. In contrast, reconfiguration is frequent in edge computing applications, which often have real-time requirements and cannot tolerate long stoppage times.

B. Static-Partitioned Edge Frameworks

Traditional stream processing frameworks are designed to run on a single datacenter, where latency between operators is small and uniform. In contrast, edge computing weaves together data centers spanned across different geographic regions communicating over high latency wide area links. Applying existing stream processing frameworks to edge computing is difficult because their architectures do not incorporate features to handle the high variation in latency and bandwidth. To get good throughput, users need to configure multiple heartbeat values and set the size of low-level network parameters, such as the TCP window size. Moreover, the high latency is not compatible with flow control approaches that rely on the next operator explicitly requesting the next item one at a time.

Previous efforts at enabling stream processing on edge networks have focused on adapting existing frameworks to operate over high latency links [8]–[10]. These approaches statically partition an application into sub-applications, where each sub-application comprises a set of operators and is placed in a different data center. The communication between sub-applications happens using a queueing system, such as Apache ActiveMQ, RabbitMQ, or Apache Kafka to transfer tuples from one data center to another.

This approach complicates the design and deployment of stream processing applications, as it requires application developers to manually build multiple different sub-applications. In addition, this approach makes reconfiguration slower by requiring coordination between the sub-applications, each of which has to execute its own stop-the-world event.

III. Shepherd

Shepherd provides a familiar DAG-based dataflow programming model to develop stream processing applications. The framework transparently deploys and reconfigures the application as needed for latency sensitive applications.

A. Overview

In our endeavor towards designing a new stream processing system that could handle the challenges of the edge, a single principle emerged and underpinned our design decisions: keep the transport layer decoupled from the processing layer. This allowed us to design the transport layer to be malleable so that quick changes could be made without impacting the processing of tuples. These quick changes to the transport layer at run-time allows Shepherd to meet tight Service Level Agreement (SLA) latency guarantees while operating under variable network conditions that are common at the edge.

Shepherd is deployed on a hierarchical edge network where nodes communicate through interconnected FIFO (First In First Out) queues. As shown in Figure 4, the computing hierarchy is organized as a tree, with a traditional wide-area cloud datacenter at its root and an arbitrary number of additional layers of datacenters. Shepherd provides at-most-one tuple delivery semantics (i.e., a tuple is delivered only once or not at all). The framework always deploys a full copy of the application on the root datacenter. Tuples are produced by sources on the edge, and are forwarded by the router network towards the cloud. A tuple is consumed by the first compatible operator along this path. If none are deployed along the path, a tuple will be consumed by the replica running on the cloud. For instance, in Figure 4a, a tuple that needs to be processed by operator F is sent to the Cloud datacenter, however, in

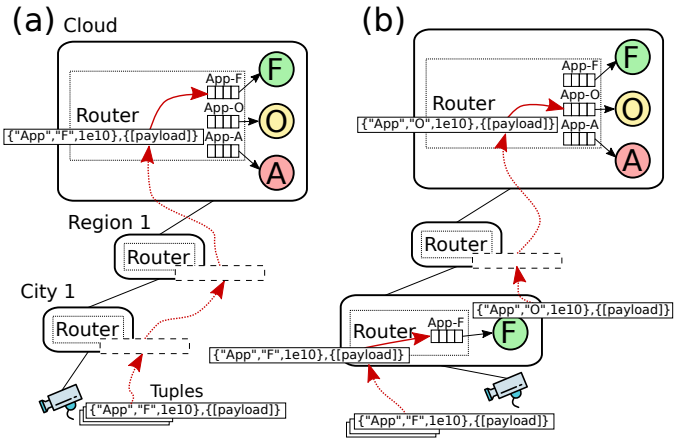

Fig. 4. Shepherd operators deployed as a tree along a geo-distributed network hierarchy

Figure 4b, this tuple is consumed earlier in the hierarchy as an operator replica F is placed to City 1.

Shepherd currently only supports the reconfiguration of stateless operators. This is not a significant limitation for a wide range of edge computing applications as the operators that benefit the most from running close to the edge (e.g., parsing, filtering, threshold) are typically stateless. Relaxing this limitation, however, is the subject of our future work.

B. Late Binding Routing System

The use of late-binding routing allows the connections between operators to be adaptive and dynamic, where operator replicas can be added or removed without stopping the application flow to (re)create the links between source and destination operators. To make a stream processing system adaptive, producers and consumers need to come and go as needed with the expectation that removing or adding operators will not break the stream processing job. Late binding makes it possible to independently deploy operators without having global knowledge of where producers and consumers are physically located. To make it possible to late bind a replica, routing information is needed from the incoming tuples. In Shepherd's dataflow programming model, a tuple carries a header and a body. The header is customizable and contains base attributes such as the type of tuple (type_id), the topology_id, and the timestamp of when the tuple was created in the data source. The body comprises the payload, which is the application data (e.g., a sensor reading or image frame as a byte array). With the topology_id and the type_id, the framework is able to find an online operator that can consume the data. The routing system performs this search for an operator task by reading the topology_id and type_id and determining if there is a local operator installed in its datacenter that can consumer the tuple, otherwise it pushes the tuple up to its parent. The decision to route local or up to the parent is the only decision made by the Router whenever any tuple arrives at a given datacenter.

Using the Object detection application as an illustrative example, we can see in Figure 4a, a pipeline application named App which contains operators [F, O, A] and is initially deployed in the cloud. Before the reconfiguration occurs, the [F] tuples are sent to the cloud. After the reconfiguration as depicted in Figure 4b, the [F] tuples get consumed at the city level and the resulting [O] tuples get routed up to the cloud where they are consumed.

C. The Data Model API

Shepherd provides the familiar API features of current state-of-the-art stream processing frameworks through a new API written in Java. For example, the user submits a logical plan as a DAG. The DAG is composed of a list of source and destination operators, where the user provides a pair of source and destination operators individually, as shown in Listing 1.

```
ArrayList<Pair> dag = new ArrayList<>();
dag.add(new Pair<>("OpF","OpO"));
dag.add(new Pair<>("OpO","OpA"));
dag.add(new Pair<>("OpA","NONE"));
submit(dag);
```

Listing 1. Submission of the logical plan in Shepherd.

The user also provides the user-defined functions written in Java for each one of the operators, as depicted in Listing 2. The operators extend the class *ShepherdOperator*, and the business logic that needs to be applied to incoming tuples is invoked when the *processTuple* function is called.

```
public class OpF extends ShepherdOperator{
    public OpF(){
        \\Constructor
    }
    public void processTuple(Tuple tuple){
        \\User-defined functions
    }
}
```

Listing 2. Operator code in Shepherd.

D. The Reconfiguration API

Shepherd provides an API that allow users to write custom reconfiguration triggers that define scheduling policies that are used to update the physical plan of any running job. For example, in Figure 4 a trigger was activated in the cloud to respond to traffic increasing at the edge a replica of type [F] was deployed. The trigger is fired based on the metrics data (e.g., operator replica metrics – latency, throughput, arrival rate, departure rate, and queues size – or computing/network metrics – memory, CPU, and bandwidth) that is transmitted back to the cloud from all the child datacenters. The reconfiguration instructions come strictly from the Shepherd framework instance running in the cloud. One or more triggers can be defined as a package to deploy multiple changes in parallel in the case where many datacenters need to be reconfigured at once. Triggers are defined by the user and are submitted to the framework along with the user code as a package. Each user defined trigger can contain any type of rule which is

called in a feedback loop to check whether the current state of the edge hierarchy requires a reconfiguration. Listing 3 shows an example of programming code that extends class *ReconfigurationTrigger* to construct a user-defined reconfiguration trigger (*ruleTriggered*) and a custom scheduling policy (*getExecutionPlans*). The *ReconfigurationStats* and *DataCenterManager* objects contain the collected performance metrics from all data centers.

Shepherd reconfiguration actions are minimally disruptive, Shepherd only contacts datacenters involved in the reconfiguration. In addition, Shepherd does the reconfiguration by computing the difference between the previous physical plan and the new physical plan to reduce the number of impacted datacenters.

```
public class UserTrigger extends
ReconfigurationTrigger{
 public UserTrigger(String triggerID,
 String topologyID) {
    //Constructor
 }
 @Override
 public boolean ruleTriggered(
 ReconfigurationStats reconfigurationStats){
    //User-defined trigger
 }

 @Override
 public ArrayList<ExecutionPlan>
 getExecutionPlans(DataCenterManager
 dataCenterManager){
    //Scheduling policy
 }
}
```

Listing 3. Example of a custom reconfiguration trigger and scheduling policy.

E. Architecture

Shepherd comprises two major subsystems: (1) the Shepherd Cloud Master as depicted in Figure 5 and (2) the Shepherd Datacenter Components shown in Figure 6. The Shepherd Cloud Master is responsible for accepting user jobs, translating logical plans into physical plans and publishing initial/re-configurations plans. The Task Manager component subscribes to the Shepherd Cloud Master and receives physical plans that contain instructions to instantiate operators and reconfigure the transport layer in its own datacenter.

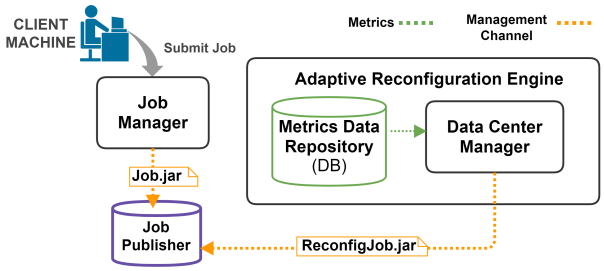

Fig. 5. The Shepherd Cloud Master

Each datacenter (including the cloud root) contains the full Shepherd Datacenter Component Set, but only the cloud root contains both the Shepherd Cloud Master and the Shepherd Datacenter Component Set. This is because it is always assumed that any jobs deployed on the Shepherd framework have the full application running in the cloud and can process tuples that are not handled by operators running on a lower layer of the edge hierarchy. Only one instance of the Shepherd Cloud Master is needed, and it always runs in the cloud root because it needs to receive metrics and publish instructions to all child nodes.

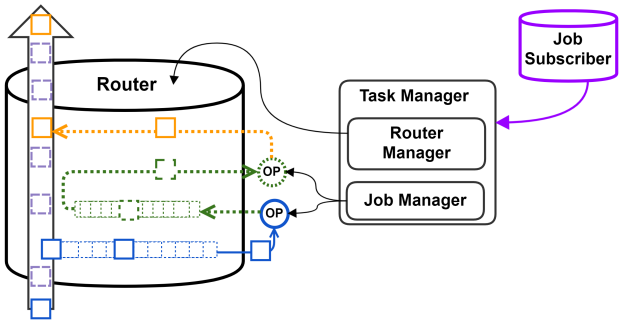

Fig. 6. Shepherd Datacenter Component Set

The Task Manager is responsible for processing instructions that have been published to its receiving queue by the Shepherd Cloud Master. When a new operator replica needs to be set up it is sent as a message to the target datacenter via the router network by the Datacenter Manager. The router network provides reliable transport over WANs and for this reason it is preferred over simple ssh socket connections which have no built-in resiliency. Once the Task Manager receives an instruction to update the physical plan, it unpacks the payload which contains the user code and allocates a slot on one of its machines in that datacenter. A slot is a unit of compute which is a CPU core and the Task Manager is aware of how many slots it can allocate to ensure the slots are not over-provisioned. The operator code is then started in its own JVM and makes a connection to the router that will persist for the life of the operator. In the case where the instruction from the Shepherd Cloud Master is to remove a replica, a signal is sent to terminate the JVM and clean up the queues if there is no longer any replicas of that type in the datacenter.

F. The Router: A Modifiable Transport Layer

- *Within the datacenter*: The key to the success of the late binding system is that Shepherd performs all the steps required to bring an operator online, and then a final action is triggered to redirect tuples to the new operator replica. To do this, we created a modifiable transport layer that could be manipulated independently of the data processing layer where the tuple processing takes place. To illustrate how the system works, Figure 6 shows that the Job Manager launches the operators and from this

point on, its only function is to decommission the operator if needed. The operators are autonomous, consuming data and publishing metrics directly to a separate data stream that sends metrics data to the cloud. The key point of this design is that the Router Manager can setup complex queue structures *around* the live stream and only introduce this new structure to the tuple flow when all operator code has been downloaded and the operators have acknowledgement back to the Task Manager that they are ready to receive tuples from the router. Current state-of-the-art frameworks deactivate the topology from the moment the reconfiguration is triggered and therefore they incur downtime that consists of downloading the user code, standing up the operator instance and finally activating the network layer to let tuples flow between operators.

- *Between datacenters*: Network latency can differ significantly between the cloud data center and the edges, depending on the distance in between. Stream processing frameworks that operate within datacenters do not have to deal with this problem because LAN latency is sub millisecond. However, WAN latency can be dozens of milliseconds and this has an impact on the ability to process tuples at high throughput and the network latency can impact the time it takes to stand up new nodes in the network because large amounts of code must be delivered to the target datacenter. To mitigate this problem, Shepherd can change the number of TCP connections at run-time when sending data over the Internet to other datacenters. This allows more packets to be in flight at once and can be customized per link depending on the amount of latency and available bandwidth on the link.

G. Reducing Operator Cold Start

A common issue with all VM-based programming languages, such as Java, is that they suffer from a cold start because the code is lazily optimized based on how often a particular branch of code is invoked. This presents a serious problem for stream processing systems that cannot accept sudden tuple latency because a new cold replica was allowed to process tuples. Shepherd uses a novel warm-up technique to overcome this problem that uses a sample of the live stream of tuples to warm up the operator code *before* the operator starts to participate in contributing to the processing of tuples for the job.

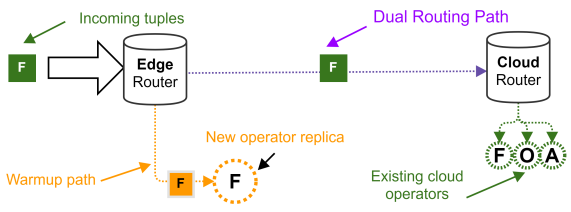

Fig. 7. Operator warm-up process.

Figure 7 illustrates the warming process for a new [F] operator replica that is added to the edge datacenter. The router is reconfigured to provide a sample of the data stream to the operator for a configurable period of time. Once this time elapses, it is assumed that the code inside the operator is now 'hot'. A reset tuple is injected into the data stream and the flow is released behind the reset tuple. When the reset tuple arrives inside the operator, the reset() method is called and the operator begins processing live tuples. The inject and switch operations have minimal impact on the tuple latency and finish in under 20 ms. This feature supports chained warm-ups as part of a multiple replica reconfiguration in the same datacenter. Each operator is designed to propagate the reset tuple to any local successor operators so that they too can be reset and begin processing live tuples.

IV. Implementation

We implemented a prototype of Shepherd as a Java application that makes use of Apache ActiveMQ Artemis [11] which is an off the shelf message broker. This broker provides Shepherd with the building blocks necessary to construct our modifiable transport layer that is optimized for WAN based communication. The processing layer is implemented as Java processes running inside Docker containers.

A. Transport Layer Implementation

The Shepherd Router is created using the features of Artemis to establish an interconnected network that spans the hierarchy from cloud to edge. Message brokers typically implement a variety of queuing structures that can be used to create complex routing paths and mechanisms to adjust the configuration at run-time. Shepherd uses the broker network to transport both user data and management messages. Separate communication paths are established for user data and management channels are restricted for use by the framework only. This ensures that tuples from different domains and data from multiple running jobs are not mixed. User data is transported inside Shepherd tuples, which are then wrapped in the broker messaging protocol to make our tuples compatible with the brokers own network stack. This wrapper contains its own header that is used to route messages in the broker network. The information in the ShepherdTuple header can be extracted and exposed to the broker network message header to route tuples based on (key, value) pairs defined by the user. This allows the Shepherd Router to consider tuple metadata appended by the operator to be used in routing decisions. Artemis is a good choice for creating an edge computing network layer because it has built in support for resilient communication (timeout and automatic retry mechanisms) when working with unstable and high latency links. Most importantly, it supports duplicate checking, which helps ensure correctness in Shepherd. It also supports very large messages (GB) which helps Shepherd to deploy large amounts of user code. As previously mentioned, the issue of high latency can impact the performance of applications and impact downtime. Artemis provides bridges to interconnect

brokers, which provides Shepherd with the ability to setup multiple connections to overcome WAN latency. Security in the transport layer is also provided out of the box (encryption, credentialed access, etc) and any incoming tuples that do not belong to a topology are routed to a dead-letter-address in the cloud for further review. Tuples are only routed to operator instances if the data source (IoT device) can authenticate to the broker and has the unique topology_id. Essentially, Artemis functions as proxy for data to reach Shepherd.

Intra Datacenter Communication Optimization: In our initial design of the framework, we used the broker for all communication between operators. This has two drawbacks. First, this creates a lot of load on the router. Second, the extra communication adds additional time to the overall end-to-end latency of the application. Since edge computing applications are already subject to fixed high network latency, we optimized our design to reduce the communication overhead when operators in the *same* datacenter need to communicate. ZeroMQ, which is a lightweight message passing library, is used to enable direct point-to-point communication between operator containers collocated on the same datacenter; however, this is done without compromising reconfigurability.

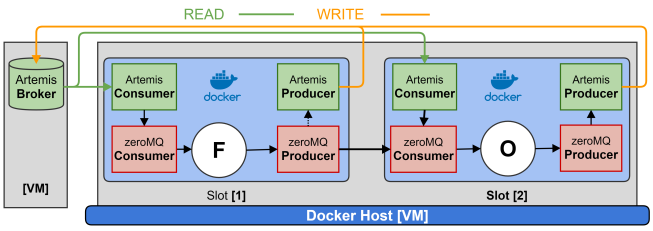

Fig. 8. Hybrid ZeroMQ and ActiveMQ Artemis Implementation

As shown in Figure 8 operators are simultaneously connected to Artemis and the other operators via ZeroMQ. This dual connectivity pattern allows Shepherd to retain its flexibility to late bind operators because adding the operator to the Artemis broker does not interrupt the stream processing job or the flow of tuples moving through the broker. Since operators can be added or removed at anytime, an important point to note is that when we late bind an operator to an existing deployment, an extra step is needed if the *predecessor* operator is currently writing its output to the broker. Looking again at Figure 8 the [F] operator would have been sending its output to the broker if the [O] operator was not installed. When the [O] operator is added at a later date, it goes through 2 steps to become an active participant in the topology. First, the new replica gets its warm up tuples *directly* from the [F] operator via special sockets that provide a copy of [O] tuples that it is producing. Second, after the warm-up phase, the [O] operator automatically detaches from the warm-up socket, attaches itself to the live output socket of [F], sends a message to the [F] operator via the management channel to stop sending tuples to the broker and instead send its output to the new [O] replica. The new [O] replica then becomes responsible for writing out to the broker. This switch takes about 10ms to occur.

B. Processing Layer Implementation

Operators are implemented as Java applications running inside Docker containers. The containers are created by the Task Manager, which keeps an accounting record of what is running in each datacenter. The operators can exchange management messages with the Task Manager as well as the other operators in the same datacenter via a special queue in Artemis. When an operator is launched by the Task Manager, it waits to receive a confirmation of 'ready' state from the operators. This allows the framework to wait until the most optimal point to divert tuples to that container.

C. Fault Tolerance

The current shepherd prototype does not guarantee tuple processing in case of failure. If an operator or router fails it is restarted, but any tuples in flight are dropped. Shepherd, however, leverages tuple queues in Artemis to provide resiliency to intermittent network failures. Adding fault tolerance support to Shepherd is the subject of our future work.

V. Performance Evaluation

In this section, we conduct an experimental comparison of Shepherd with Apache Storm on emulated edge environments. Our evaluation quantifies the effects of reconfiguration on application throughput and latency.

A. Experimental Setup:

We conducted our evaluation using an emulated distributed edge network consisting of virtual machines running on 4 Amazon datacenters (N. California, Oregon, Ohio, and Virginia), which we organized as a 2-level hierarchy with N. California as the root and the other 3 datacenters configured as edge nodes as depicted in Figure 9. The figure also shows the measured average round trip times (RTTs) between datacenters.

All used VMs are of the type m5.2xlarge that has 32 GB of RAM and 8 vCPU/Threads from a 3.1 GHz Intel Xeon® Platinum 8175M processor. All clocks are synchronized using Amazons NTP time service, which is the best practice for time keeping on the AWS infrastructure.

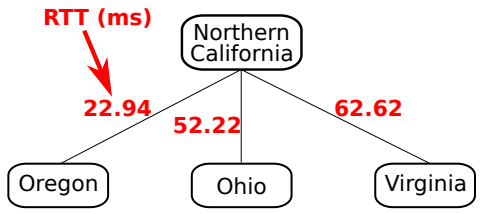

Fig. 9. The 2-tier network infrastructure with Oregon, Virginia, Ohio as edges, and the cloud is N. California. The network latencies are the average round trip times (RTTs).

B. Benchmark Applications:

To the best of our knowledge, there is no current standard benchmark for stream processing in edge networks. Instead, we use in our evaluation three applications that we believe are representative of functionality that is likely to leverage edge streaming: data filtering, natural language processing, and object recognition. The applications have different workload profiles, code sizes and latency requirements. The code sizes for each application are listed in Table I.

TABLE I
THE SIZE OF THE APPLICATION'S EXECUTABLE

Application	Size in MB
ETL	54
Twitter Sentiment Analysis	54
Object Detection - Lite	305
Object Detection - Full	1002

Object Detection [12], [13]: This application uses video cameras and OpenCV [14] to track vehicles for smart traffic monitoring [15], [16]. The data set used in these experiments is a camera feed of city road traffic obtained from Urban Tracker [17]. The Object Detection application pipeline consists of a chain of 3 operators: frame filter [F] removes frames that are not significantly different from the previous frame; object detection [O] uses a pre-trained YoloV3 [18] model to detect objects in the image frame; car counter [A] aggregates statistics. The [O] operator is CPU intensive and can process just a few frames per second. The Obj Detection app has two versions. The Lite version does not embed the OpenCV library into the application executable which was done to reduce the payload size of the user executable. Instead, the OpenCV library is pre-loaded directly into Storm's classpath in all datacenters. The Obj Detection Full version embeds the OpenCV library *with* the executable. The purpose of creating 2 versions of this application was to measure the impact of the executable size on the overall reconfiguration time.

Twitter Sentiment Analysis [19]: This application applies NLP to text-based tweets to analyze the polarity of tweets by counting positive and negative words and computing the difference. The Sentiment Analysis application represents a class of applications that produce small tuples at a very high frequency. The tweets are JSON dictionaries, each tweet corresponding to a tuple. The application pipeline consists of a chain of 6 operators: parsing [P], English filtering [F], lowercase and symbol removal [S], remove stop words [T], word counter [C], and positive or negative word score [E]. The data set that we used contains real tweets collected from the global public Twitter API. The regional nature of news broadcasting makes Twitter an interesting application because the geographic context of a Tweet adds to its meaning. For example, recent work has shown that deploying Twitter on the edge can enable location-based top-k popular topic queries and return the result with lower latency as compared to offloading the processing to the cloud [20].

Extraction, Transform and Load (ETL) [21]: This pre-processing application consumes data from smart building environmental sensors to be used in analytics applications that are used to optimize building maintenance and energy usage. The dataset for this experiment is from Sense Your City [22]. The sources are sensors that emit JSON tuples. Processing this information earlier at the edge can reduce the time to find anomalous readings in the data stream. The following pipeline processes the tuples: two filters, range [R] and Bloom [B] filter outliers, interpolation [I] predicts the values that fall within a range and lastly, annotation [A] adds additional meta-data into the observed fields of the tuple. This application has smaller payloads as compared to the Twitter application (integer data vs text data) but has similar data production patterns.

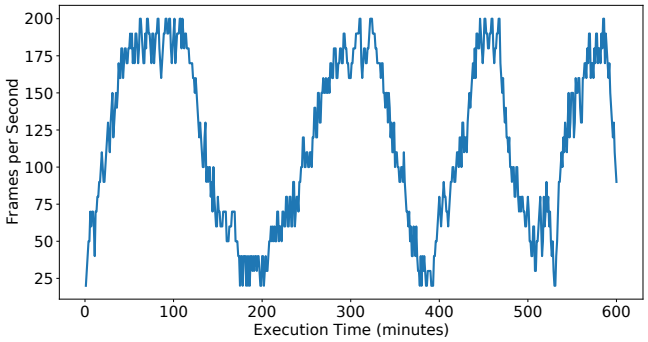

Fig. 10. The workload pattern of 5 different cameras deployed in a large North American city shows large variation across the day.

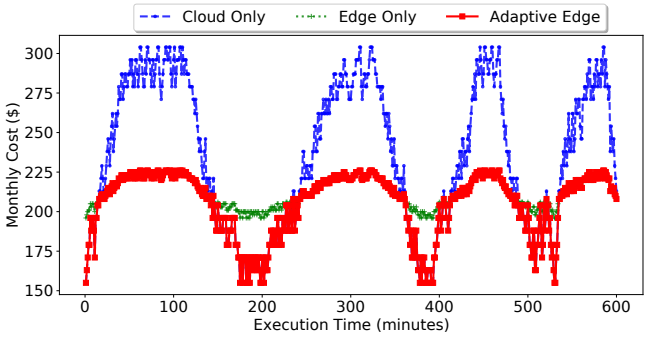

Fig. 11. The adaptive edge strategy leverages both the cloud and edge which results in the lowest monthly average cost as compared to cloud or edge only.

C. Benefits of Dynamic Reconfiguration

We illustrate the benefits of reconfiguration using the object detection application. In this scenario, the application consumes video frames (i.e., standard definition – dimensions 884x498 pixels and average size of 130 KB) from five motion-activated cameras spread across multiple streets in a large North American city. Each camera emits video frames at different rate as some streets are constantly busy, some are idle, and some are busy on specific hours. Figure 10 demonstrates

the skewness of the combined production rate, where the number of active cameras varies over time.

The service provider pays for data transfers and the utilization of computing resources. The communication cost is based on AWS PrivateLink [23] that charges users by bandwidth usage, and the computation cost is priced according to AWS Wavelength [4] for edge data centers and AWS EC2 for cloud data centers. Deploying VMs on AWS Wavelength increases the cost by approximately 40%.

We consider three deployment options: Cloud Only, Edge Only, and Adaptive Edge, which dynamically adapts the placement of operators based on the workload. The application has a significant data reduction after the object detection operator [O], which decreases transferred data by over 80%.

Figure 11 shows that deploying the application on Cloud Only, the traditional method, results in the highest cost given the high communication cost of sending data over AWS PrivateLinks when there is high demand. In contrast, Edge Only reduces cost overall, but is more expensive than Cloud Only when the frame rate is low as the AWS Wavelength cost dominate. Adaptive Edge achieves the lowest cost by leveraging the edge on high demand times and the cloud on low demand periods. This adaptive approach reduces the costs by over 13% and 3% when compared to Cloud Only and Edge Only, respectively. The savings are more significant when there is a high fluctuation in the production rate, numerous cameras involved, or dataframes with higher resolution (e.g., HD, 4K, and 8K).

D. Low Latency Case

We evaluated the effect that reconfiguration has on the performance of the Twitter Sentiment Analysis application when running on top of Shepherd and Storm. We run this application on an emulated 2-level edge network consisting of a cloud data center and 5 edge data centers. Attached to each edge is a separate data source that produces 600 messages per second when active, and 0 otherwise. Each emulated datacenter runs on a separate virtual machine (VM). Since reconfiguration time is affected by high network latency, we consider an idealized best-case scenario where all VMs are collocated in a single Amazon datacenter (N. California).

The reconfigurations were triggered in Storm using the re-balance command. We created a custom Storm scheduler by extending the $ISchedulder$ class that can react to the throughput increases and create new physical plans. Likewise, in Shepherd, triggers were created to react to the increase in traffic load and to deploy a new physical plan.

The experiment follows the timeline shown in Figure 2. The physical plan at the start of the experiment has operators [S, T, C, E] deployed on the cloud, and operators [P, F] deployed on edge 1 which has the only active data source. After some time, the data source attached to edge 2 becomes active and the application is reconfigured by deploying additional replicas of operators [P, F] on edge 2 (see Figure 2b). This process is repeated 3 more times until all 5 sources are active and replicas of the [P, F] operators are running on all edges.

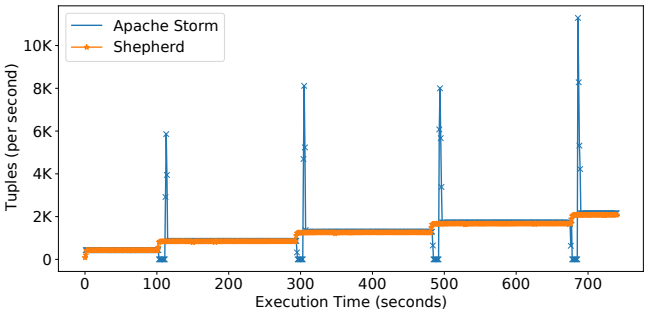

Fig. 12. Shepherd incurs no downtime when reconfiguring to add new replicas [P, F] to each edge in response to increased Twitter traffic. Storm incurs downtime of several seconds with throughput spikes after each reconfiguration.

Results discussion: Looking at Figure 12 we can see that at each reconfiguration event, Storm stops processing tuples for several seconds and the throughput goes to zero. Throughput is the number of tuples that have arrived at the sink (measured per second). In contrast to this downtime, Shepherd continues to process the tuples and the reconfiguration event has little to no impact on the throughput.

Table II shows detailed results for Storm. *Downtime* is defined as the amount of time that has elapsed since the framework has stopped processing tuples; *SLA violations* measure the number of tuples during the experiment that arrived at the sink with an end-to-end latency of *greater* than 100 ms; and *recovery time* reflects the time period in which the framework is working through the backlog of the tuples. Shepherd does not incur enough disruption (>100 ms), and therefore all values for these metrics are zero and not shown in the table for brevity. In contrast, every reconfiguration call in Storm causes a backlog that grows while the framework is in the process of building a new physical plan, uploading operator code to the target worker machines and standing up the operators. This is what causes the throughput spikes and the late tuples measured as SLA violations in Storm. The more edges there are, the larger the number SLA violations occur and they will continue to grow because the Downtime will likely grow as more worker nodes take longer to synchronize when they are added to the Storm cluster.

Figure 13 shows that Shepherds task setup time in N.California takes 107 seconds when installing the initial operator into the datacenter and additional reconfigurations take a total of 35 seconds to complete. The warm-up time portion is set by the user and its default value is 30 seconds. The framework only incurs a minor disruption of less than 50ms to update the physical plan once the task setup and warm-up steps have been completed. This transition time is less than 50ms because Shepherd performs reconfigurations in the *background*.

E. The Effect of WAN Latency

This experiment assesses the consequences of the WAN latency on the application reconfiguration for both Apache

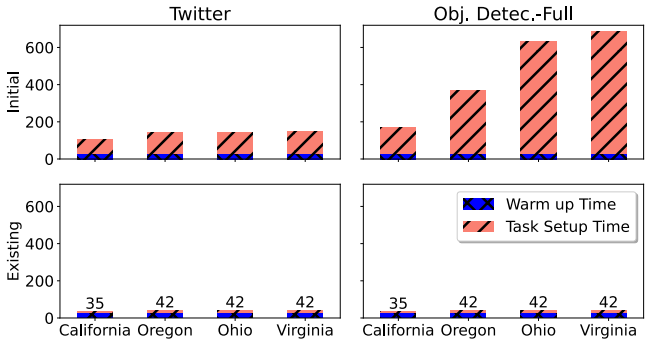

Fig. 13. Shepherd reconfigures in the background because the warm-up time and task setup time do not cause downtime when reconfiguring. The 30 second warm-up time shown here is configurable. The results for ETL (not shown) are similar to Twitter.

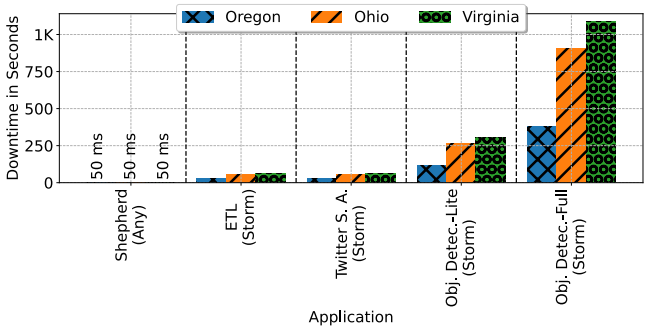

Fig. 14. Shepherd downtime is always <50ms when reconfiguring over the WAN, while for Storm the impact of user executable code size and network latency causes downtime of up to many minutes.

TABLE II
STORM: DOWNTIME, RECOVERY TIME, AND NUMBER OF SLA VIOLATIONS (LATENCY > 100 MS) FOR RECONFIGURATIONS INSIDE A LAN

	2nd	3rd	4th	5th
Downtime (seconds)	9.1	8.6	8.2	8.3
Recovery Time (seconds)	3	3	3	3
Number of SLA Violations	11404	15929	18074	22778

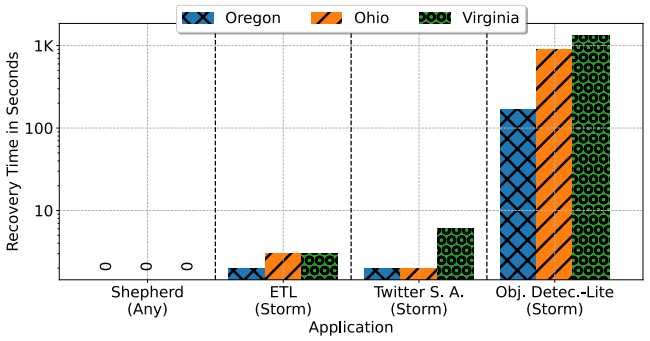

Fig. 15. Shepherd needs no time to recover after a reconfiguration over the WAN because it incurs <50ms of downtime. Storm spends several minutes in recovery, trying to clear the backlog of tuples.

Storm and Shepherd. We carry out our evaluation on the emulated distributed edge network shown in Figure 9 using the 4 benchmark applications described in Section V-B. Apache Storm's Zookeeper and Nimbus run in an isolated VM in N. California, as well as Shepherd's Cloud Master. In each edge datacenter, a data source produces 400 (Twitter and ETL) or 2 (Object Detection) messages per second when active, and 0 otherwise. When a data source becomes active, a reconfiguration is called to add [P, F] for the Twitter Sentiment Analysis application, [F, O] for the Object Detection, and [P, R] for the ETL to the same datacenter as the data source.

Each experiment in each application benchmark begins with an active data source in the Oregon datacenter, but all operator replicas are deployed in the N. California. The first reconfiguration adds operator replicas to the Oregon datacenter. The second reconfiguration happens because a data source becomes active in the Ohio datacenter. The last reconfiguration is because a data source starts in the Virginia datacenter.

WAN Downtime Results Discussion: Figure 13 that as could be expected network latency and the size of the application affect the total reconfiguration time in Shepherd. However, Figure 14 shows that neither the network latency nor the application sizes affect Shepherd's downtime (less than 50 ms and a 99% improvement over Apache Storm). This is because the bulk of the reconfiguration process in Shepherd happens in the background. In contract, for Apache Storm network latency and the application size have a large impact on downtime. The more interesting result is seen with the OBJ-Lite and OBJ-Full applications, where the differences in downtime are significant. The Twitter application and ETL application have the same size of executable and therefore incur the same downtime.

WAN Recovery Time and SLA Results Discussion: Figure 15 shows that Shepherd does not incur a recovery time penalty, while Figure 16 demonstrates that all tuples are delivered under the SLA threshold of less than 1 second. In contrast, the recovery time for Apache Storm is significant in a geographically distributed environment. The recovery time

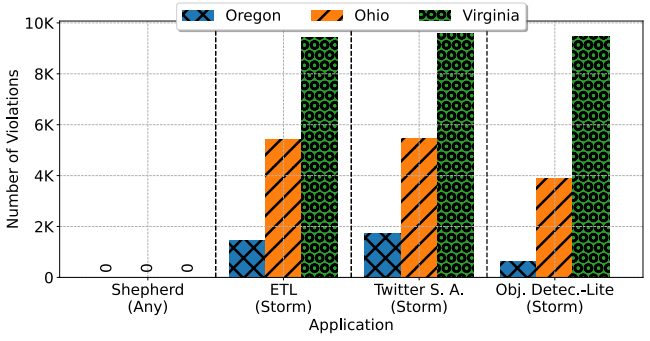

Fig. 16. Shepherd has no SLA violations (defined as latency > 1 second) after reconfiguring on the WAN, while Storm experiences large numbers of SLA violations (late tuples) because of its long recovery time.

is a consequence of the downtime, as many tuples pile up during the reconfiguration. Depending on the application, this recovery time is longer if the processing time is large and the operators cannot clear the backlog fast enough to make progress on the tuples that arrived after the reconfiguration. For example, the Object Detection application has a machine learning operator that requires at least 250 ms of compute time to detect objects. The 2, 13, and 16 minute recovery time for the OBJ-lite application likely is not an acceptable amount of downtime under any practical scenario when reconfiguring to any of the three edge datacenters. The amount of late tuples is excessive and grows at similar rate to the LAN experiment, and is further amplified by the WAN latency. These violations can be a deal breaker as many applications cannot afford to deliver late tuples.

F. Moving One Operator Closer to the Edge

This experiment evaluates the impact of a reconfiguration for deploying an operator replica of the ETL application in an active datacenter. We use the Oregon datacenter and the N. California datacenter from our emulated distributed edge network. Apache Storm's Zookeeper and Nimbus, and Shepherd's Cloud Master also run in an isolated VM in N. California. The experiment starts with an active data source and an operator replica of the Parser in the Oregon datacenter. The rest of the operators are placed in the N. California datacenter. Then, at mark 110 seconds, a reconfiguration pushes down the Range Filter to the Oregon datacenter. This evaluation validates the best scenario in Apache Storm, where all application jars are already loaded in the destination datacenter. As depicted in Table III, Shepherd outperforms Apache Storm by reducing the downtime by 97.50%. This also reduces the recovery time and the number of SLA violations, showing that Shepherd can meet tight SLA latency requirements that Storm cannot.

TABLE III
WHEN RECONFIGURING AN EXISTING DATACENTER, SHEPHERD DELIVERS ALL OF ITS TUPLES ON TIME, WHEREAS STORM STILL HAS LATE TUPLES EVEN WHEN THE RECONFIGURATION DOWNTIME IS VERY SHORT

	Shepherd	**Apache Storm**
Recovery Time (seconds)	0	2.6
Downtime (seconds)	.05	2
Number of SLA Violations	0	106

G. Operator Warm-up

This set of experiments demonstrates the value of having the warm-up feature in Shepherd. The experiment scenario begins with the Twitter Sentiment Analysis application entirely deployed on the N. California cloud datacenter, and at the 120s mark, the [F] operator is migrated to the Ohio data center. Figure 17 shows that when the warm-up feature is turned off there is a large drop in the throughput even at the lower tuple rate of 1200 per second. We chose the rates of 1200 and 2400 because the CPU of the machine reached 80% usage at the 2400 rate. The warm-up feature becomes even more valuable

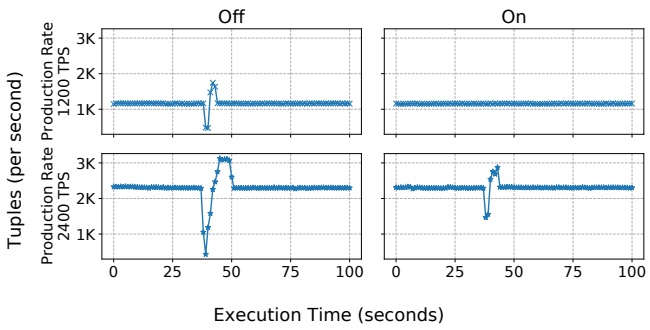

Fig. 17. Without Shepherds warm up feature turned on, the throughput drops are significant. The warm-up feature helps Shepherd meet tight SLA guarantees by reducing disruptions that increase end-to-end tuple latency.

if we double the rate to 2400 because when the feature is turned off there is a throughput drop of 73% and it takes a longer time to return to normal processing levels. The warm-up feature allows new operator replicas to warm-up before they can accept live tuples, and therefore avoiding the cold-start problem of JVMs.

H. Improving Communication within the Data Center: ActiveMQ vs ZeroMQ

We evaluate the performance gains of using ZeroMQ as compared to ActiveMQ for intra datacenter communication. The ETL application is used in this experiment with three different ingestion rates (1200,1600 and 2000 tuples per second). Figure 18 and Figure 19 show the benefits in terms of lower latency and higher throughput when replacing ActiveMQ with ZeroMQ for communication between operators in the same datacenter. The benefits increase at higher throughput levels as ActiveMQ begins to destabilize when the throughput hits 2000 tuples per second. Whereas, the two methods achieve similar latency and throughput when the system is processing a small number of tuples (1200). Even at the mid-range, the results show that the latency of ActiveMQ is over 93% higher than ZeroMQ when the production rate is 1600 tuples per second. As previously discussed in the Implementation section and shown in Figure 8, our use of ZeroMQ does not affect the reconfigurability of the operators as they remain connected to the Shepherd Router and can benefit from the features of our modifiable transport layer.

VI. RELATED WORK

This section discusses previous research on reconfiguration.

Stop the world: Apache Flink [5], Apache Spark [6], and Apache Storm [7] were designed to run on a single monolithic data center where reconfigurations happen quite rarely. For this reason, classical stream processing frameworks conduct reconfigurations by using the stop-of-the-world approach [24] where the application is paused, the new physical plan is deployed, and then the application is resumed. Unfortunately, this approach incurs high downtime since it demands synchronization barriers to maintain correctness. The stoppage times become longer when such frameworks are deployed on an

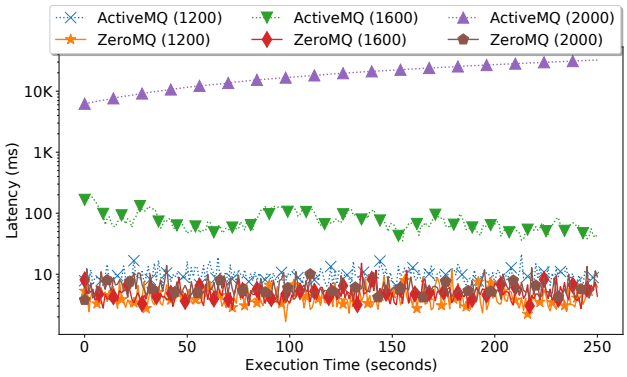

Fig. 18. Comparison between ZeroMQ and ActiveMQ.

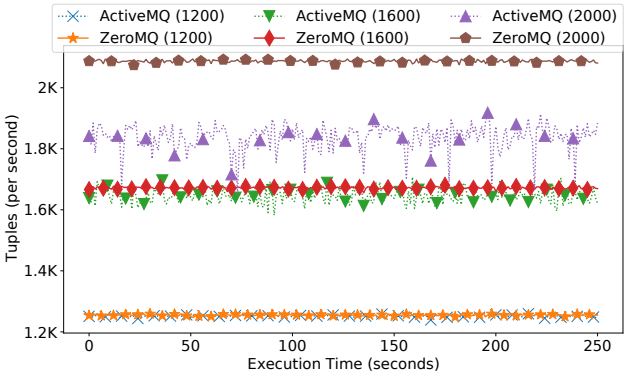

Fig. 19. Comparison between ZeroMQ and ActiveMQ.

edge computing infrastructure as they require several rounds of communication to synchronize each step (e.g., all cluster nodes enter in reconfiguration, the operator replicas are re-assigned, and the operators' connections are recreated). This multiple-round communication process leads to stalls of minutes of magnitude. Prior works, such as R-Storm [25], SpanEdge [26], EdgeWise [27], Trisk [28], E2DF [29], and DART [2], are optimized to run on geographically distributed infrastructure. Some of these solutions also introduce methods that reduce the amount of transferred data to make the reconfiguration faster. However, reconfiguration in these approaches remains a multi-second global operation that requires barriers to synchronize the state of all cluster nodes.

Parallel track: Parallel track [30] arises as an attractive solution to execute seamless reconfiguration. This method creates a new operator instance that runs concurrently until it synchronizes the old and the new operator replicas. However, the parallel track has not been deployed on the WAN due to the hard communication management and synchronization of multiple physical locations. For example, ChronoStream [31] and Gloss [32] use the parallel-track approach on a single data center. The former addresses the reconfiguration without accounting for the slow-start initialization of an operator replica, and it requires that downstream operators control duplicated messages by filtering them out. The latter looks at the reconfiguration as a recompilation process by rebuilding the whole DAG, running it concurrently for some time, and controlling the discarding of duplicate messages – making the reconfiguration more costly as it includes new operations. In contrast, Shepherd's design leverages a parallel track-inspired solution for an edge computing resource-limited shared infrastructure that overcomes the slow-start initialization of a new operator replica without creating the cost of duplicating the whole application DAG.

Finally, this paper builds on our workshop paper [33], which first identified the challenges associated with stop-the-world reconfiguration and proposed the use of late-binding routing as an alternative. This earlier work did not provide a full design or implementation, and as a result did not address real-world issues, such as high latency links, operator warm-up, and router overhead. In addition, the workshop paper only presents results from a single experiment based on a simple 2-operator micro-benchmark that moves tuples around but did not do any processing.

VII. Conclusion

In this paper, we introduced Shepherd, a novel stream processing framework for edge computing. Shepherd offers efficient dynamic placement of operators along a hierarchy of datacenters located between the edge devices and the cloud. Through its novel late binding, hierarchical routing, and warm-up techniques, Shepherd decreases downtime due to reconfiguration from a few minutes to milliseconds. Shepherd enables a wide-range of edge computing applications that rely on stateless stream processing. In future work, we plan to extend Shepherd with policies that automatically adjust the parallelism degree of operators, as well as offering support for stateful stream processing and fault tolerance.

References

[1] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[2] P. Liu, D. D. Silva, and L. Hu, "DART: A scalable and adaptive edge stream processing engine," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 239–252. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/liu

[3] B. Varghese, E. De Lara, A. Y. Ding, C.-H. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele *et al.*, "Revisiting the arguments for edge computing research," *IEEE Internet Computing*, vol. 25, no. 5, pp. 36–42, 2021.

[4] "Wavelength," 2022. [Online]. Available: https://aws.amazon.com/wavelength/

[5] "Apache Flink," 2022. [Online]. Available: http://flink.apache.org/

[6] "Apache Spark," 2022. [Online]. Available: https://spark.apache.org/

[7] "Apache Storm," 2022. [Online]. Available: https://storm.apache.org/

[8] Y. Fu and C. Soman, *Real-Time Data Infrastructure at Uber*. New York, NY, USA: Association for Computing Machinery, 2021, p. 2503–2516. [Online]. Available: https://doi.org/10.1145/3448016.3457552

[9] D. Battulga, D. Miorandi, and C. Tedeschi, "Fogguru: a fog computing platform based on apache flink," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 156–158.

[10] E. G. Renart, D. Balouek-Thomert, and M. Parashar, "An edge-based framework for enabling data-driven pipelines for iot systems," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 885–894.

[11] B. Snyder, D. Bosanac, and R. Davies, "Introduction to apache activemq," *Active MQ in action*, pp. 6–16, 2017.

[12] X. Zeng, B. Fang, H. Shen, and M. Zhang, *Distream: Scaling Live Video Analytics with Workload-Adaptive Distributed Edge Intelligence*. New York, NY, USA: Association for Computing Machinery, 2020, p. 409–421. [Online]. Available: https://doi.org/10.1145/3384419.3430721

[13] A. Anjum, T. Abdullah, M. F. Tariq, Y. Baltaci, and N. Antonopoulos, "Video stream analysis in clouds: An object detection and classification framework for high performance video analytics," *IEEE Transactions on Cloud Computing*, vol. 7, no. 4, pp. 1152–1167, 2019.

[14] "OpenCV," 2022. [Online]. Available: http://opencv.org/

[15] Q. Zhang, H. Sun, X. Wu, and H. Zhong, "Edge video analytics for public safety: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1675–1696, 2019.

[16] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[17] J.-P. Jodoin, G.-A. Bilodeau, and N. Saunier, "Urban tracker: Multiple object tracking in urban mixed traffic," in *IEEE Winter Conference on Applications of Computer Vision*, 2014, pp. 885–892.

[18] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018, cite arxiv:1804.02767Comment: Tech Report. [Online]. Available: http://arxiv.org/abs/1804.02767

[19] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes, "Dspbench: A suite of benchmark applications for distributed data stream processing systems," *IEEE Access*, vol. 8, pp. 222 900–222 917, 2020.

[20] A. Jonathan, A. Chandra, and J. Weissman, "Locality-aware load sharing in mobile cloud computing," in *Proceedings of The10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 141–150. [Online]. Available: https://doi.org/10.1145/3147213.3147228

[21] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017, e4257 cpe.4257. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4257

[22] "Urban environmental monitoring project," 2022. [Online]. Available: http://map.datacanvas.org

[23] "AWS PrivateLink," 2022. [Online]. Available: https://aws.amazon.com/privatelink

[24] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4334, 2018, e4334 cpe.4334. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4334

[25] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 149–161. [Online]. Available: https://doi.org/10.1145/2814576.2814808

[26] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 168–178.

[27] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A better stream processing engine for the edge," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 929–945.

[28] Y. Mao, Y. Huang, R. Tian, X. Wang, and R. T. B. Ma, *Trisk: Task-Centric Data Stream Reconfiguration*. New York, NY, USA: Association for Computing Machinery, 2021, p. 214–228. [Online]. Available: https://doi.org/10.1145/3472883.3487010

[29] M. Nardelli, G. Russo Russo, V. Cardellini, and F. Lo Presti, "A multi-level elasticity framework for distributed data stream processing," in *Euro-Par 2018: Parallel Processing Workshops*, G. Mencagli, D. B. Heras, V. Cardellini, E. Casalicchio, E. Jeannot, F. Wolf, A. Salis, C. Schifanella, R. R. Manumachu, L. Ricci, M. Beccuti, L. Antonelli, J. D. Garcia Sanchez, and S. L. Scott, Eds. Cham: Springer International Publishing, 2019, pp. 53–64.

[30] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-based data stream processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 238–245. [Online]. Available: https://doi.org/10.1145/2611286.2611309

[31] Y. Wu and K.-L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 723–734.

[32] S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe, "Gloss: Seamless live reconfiguration and reoptimization of stream programs," *SIGPLAN Not.*, vol. 53, no. 2, p. 98–112, mar 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173170

[33] Anonymous Authors, "Citation anonymized for review."