DART: A Scalable and Adaptive Edge Stream Processing Engine

Pinchao Liu, *Florida International University;*
Dilma Da Silva, *Texas A&M University;* Liting Hu, *Virginia Tech*

https://www.usenix.org/conference/atc21/presentation/liu

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

DART: A Scalable and Adaptive Edge Stream Processing Engine

Pinchao Liu
Florida International University

Dilma Da Silva
Texas A&M University

Liting Hu*
Virginia Tech

Abstract

Many Internet of Things (IoT) applications are time-critical and dynamically changing. However, traditional data processing systems (e.g., stream processing systems, cloud-based IoT data processing systems, wide-area data analytics systems) are not well-suited for these IoT applications. These systems often do not scale well with a large number of concurrently running IoT applications, do not support low-latency processing under limited computing resources, and do not adapt to the level of heterogeneity and dynamicity commonly present at edge environments. This suggests a need for a new edge stream processing system that advances the stream processing paradigm to achieve efficiency and flexibility under the constraints presented by edge computing architectures.

We present DART, a scalable and adaptive edge stream processing engine that enables fast processing of a large number of concurrent running IoT applications' queries in dynamic edge environments. The novelty of our work is the introduction of a dynamic dataflow abstraction by leveraging distributed hash table (DHT) based peer-to-peer (P2P) overlay networks, which can automatically place, chain, and scale stream operators to reduce query latency, adapt to edge dynamics, and recover from failures.

We show analytically and empirically that DART outperforms Storm and EdgeWise on query latency and significantly improves scalability and adaptability when processing a large number of real-world IoT stream applications' queries. DART significantly reduces application deployment setup times, becoming the first streaming engine to support DevOps for IoT applications on edge platforms.

1 Introduction

Internet-of-Things (IoT) applications such as self-driving cars, interactive gaming, and event monitoring have a tremendous potential to improve our lives. These applications generate a large influx of sensor data at massive scales (millions of sensors, hundreds of thousands of events per second [20,26]). Under many time-critical scenarios, these massive data streams must be processed in *a very short time* to derive actionable intelligence. However, many IoT applications [22,23] adopt the server-client architecture, where the front-end sensors send time-series observations of the physical or human system to the back-end cloud for analysis. Such a long-distance of processing makes it ***not*** appropriate or time-critical IoT application because: (1) the high latency may cause the results to be obsolete; and (2) the network infrastructure cannot afford the massive data streams.

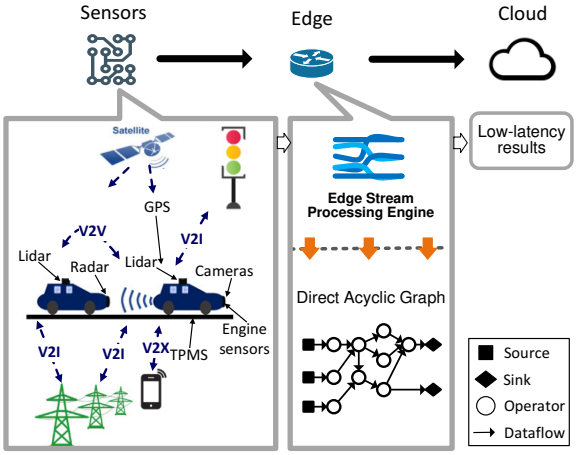


Figure 1: Edge stream processing use case.

A new trend to address this issue is edge stream processing. To put it simply, edge stream processing applies the stream processing paradigm to the edge computing architecture [37, 50]. Instead of relying on the cloud to process sensor data, the edge stream processing system relies on distributed edge compute nodes (Gateways, edge routers, and powerful sensors) which are near the data sources to process data and trigger actuators. The execution pipeline is as follows. Sensors (e.g., self-driving car sensors, smart wearables) generate data streams continuously. They are then consumed by the edge stream processing engine, which creates a logical topology of stream processing operators connected into a Directed Acyclic Graph (DAG), processes the tuples of streams as they flow through the DAG from sources to sinks, and outputs the results in a very short time. Each *source* node is an IoT sensor. Each *inner* node runs an operator or operators that can perform user-defined computation on data, ranging from simple computation such as `map`, `reduce`, `join`, `filter` to complex computation such as ML-based classification algorithms. Each *sink* node is an IoT actuator or a message queue to the cloud.

Figure 1 illustrates a use case scenario [10] that benefits from an edge stream processing engine. In future Intelligent Transportation Systems such as the efforts currently funded

*Liting is affiliated with Virginia Tech, but was at Florida International University during this work.

by the US Department of Transportation [27], cars are interconnected and equipped with wide-area network access. Even at low levels of autonomy, each car will generate at least 3 Gbit/s of sensor data [25]. On the back-end, many IoT stream applications will run concurrently, consuming these live data streams to quickly derive insights and make decisions. Examples of such applications include peer-to-peer services for traffic control and car-sharing safety and surveillance systems. Note that many of these services cannot be completed on onboard computers within a single car, requiring the cooperation of many computers, edge routers, and gateways with sensors and actuators as sources and sinks. They will involve a large number of cars and components from the road infrastructure.

However, as IoT systems grow in number and complexity, we face significant challenges in building edge stream processing engines that can meet their needs.

The first challenge is: *how to scale to numerous concurrently running IoT stream applications?* Due to the exponential growth of new IoT users, the number of concurrently running IoT stream applications will be significantly large and change dynamically. However, modern stream processing engines such as Storm [7], Flink [32], and Heron [44] and wide-area data analytic systems [39–41, 43, 53, 55, 62, 65, 66] mostly inherit a centralized architecture, in which the monolithic master is responsible for all scheduling activities. They use a first-come, first-serve method, making deployment times accumulate and leading to long-tail latencies. As such, this centralized architecture easily becomes scalability and performance bottlenecks.

The second challenge is: *how to adapt to the edge dynamics and recover from failures to ensure system reliability?* IoT stream applications run in a highly dynamic environment with load spikes and unpredictable occurrences of events. Existing studies on the adaptability in stream processing systems [34, 36, 38, 42, 64] mainly focus on the cloud environment, where the primary sources of dynamics come from workload variability, failures, and stragglers. In this case, a solution typically allocates additional computational resources or re-distributes the workload of the bottleneck execution across multiple nodes within a data center. However, the edge environment imposes additional difficulties: (1) edge nodes leave or fail unexpectedly (e.g., due to signal attenuation, interference, and wireless channel contention); and (2) accordingly, stream operators fail more frequently. Unfortunately, unlike the cloud servers, edge nodes have limited computing resources: few-core processors, little memory, and little permanent storage [37, 59] and they have no backpressure. As such, the previous adaptability techniques by re-allocating resources or buffering data at data sources cannot be applied in edge stream processing systems.

We present DART, a scalable and adaptive edge stream processing engine to address the challenges listed above. The key innovation is that DART re-architects the stream processing system runtime design. In sharp contrast to existing stream processing systems, there is no monolithic master. Instead, DART involves all peer nodes to participate in operator placement, dataflow path planning, and operator scaling, thereby revolutionarily improving scalability and adaptivity.

We make the following contributions in this paper.

First, we study the software architecture of existing stream processing systems and discuss their limitations in the edge setting. To our best knowledge, we are the first to observe the lack of scalability and adaptivity in stream processing systems for handling a large number of IoT applications (Sec. 2).

Second, we design a novel dynamic dataflow abstraction to automatically place, chain and parallelize stream operators using the distributed hash table (DHT) based peer-to-peer (P2P) overlay networks. The main advantage of a DHT is that it avoids the original monolithic master. All peer nodes jointly make operator-mapping decisions. Nodes can be added or removed with minimal work around re-distributing keys. This design allows our system to scale to extremely large numbers of nodes. To our best knowledge, we are the first to explore DHTs to pursue extreme scalability in edge stream processing (Sec. 3).

Third, using DHTs, we decompose the stream processing system architecture from `1:n` to `m:n`, which removes the centralized master and ensures that each edge zone can have an independent master for handling applications and operating autonomously without any centralized state (Sec. 4). As a result of our distributed management, DART improves overall query latencies for concurrently executing applications and significantly reduces application deployment times. To the best of our knowledge, we offer the first stream processing engine to make it feasible to operate IoT applications in a DevOps fashion.

Finally, We demonstrate DART's scalability and latency gains over Apache Storm [7] and EdgeWise [1] on IoT stream benchmarks (Sec. 5).

2 Background

2.1 Stream Processing Programming Model

Data engineers define an IoT stream application as a directed acyclic graph (DAG) that consists of operators (see Figure 1). Operators run user-defined functions such as `map`, `reduce`, `join`, `filter`, and ML algorithms. Data tuples flow through operators along the DAG (topology). In our case, DART supports both stateful batch processing by using *windows* as well as continuously event-based stateless processing. The application's query latency is defined as the elapsed time since the source operator receives the timestamp signaling the completion of the current window to when the sink operator externalizes the window results.

We consider typical edge environments. The edge compute nodes consist of sensors, routers, and sometimes gateways. They are connected by different connections such as WiFi,

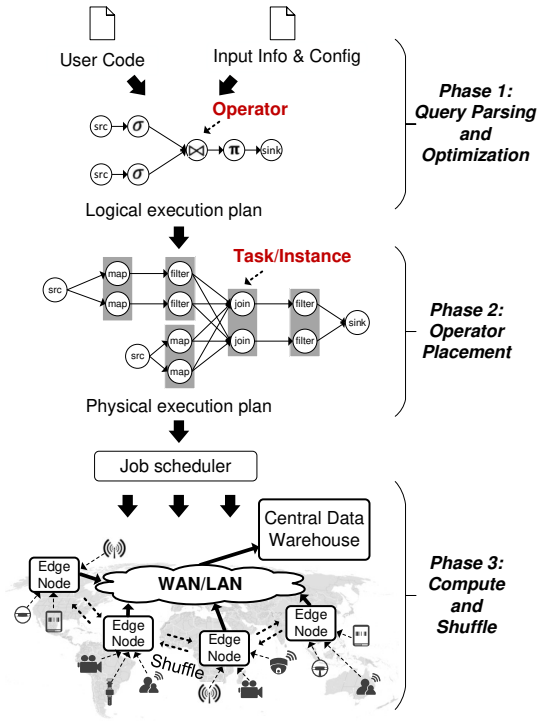


Figure 2: IoT stream applications execution pipeline.

Zigbee, BlueTooth, or LAN with diverse inbound and outbound bandwidths and latency. They have fewer resources compared to the cloud servers, but more resources than embedded sensor networks, and thus can afford reasonably complex operations (e.g., SenML parsers, Kalman filters, linear regressions). As shown in Figure 2, the execution pipeline for processing an IoT stream application has a few key phases:

- **Phase 1: Query parsing and optimization.** When an IoT stream application is submitted by a user, its user code containing transformations and actions is first parsed into a logical execution plan represented using a DAG, where the vertices correspond to stream operators and the edges refer to data flows between operators.
- **Phase 2: Operator placement.** Afterward, the DAG is converted into a physical execution plan, which consists of several execution stages. Each stage can be further broken down into multiple execution instances (tasks) that run in parallel, as determined by the stage's level of parallelism. This requires the system to place all operators' instances on distributed edge nodes that can minimize the query latency and maximize the throughput.
- **Phase 3: Compute and shuffle.** Operator instances independently compute their local shard of data and shuffle the intermediate results from one stage to the next stage. This requires the system to adapt to the workload variations, bandwidth variations, node joins and leaves, and failures and stragglers.

2.2 Stream Processing System Architecture

As shown in Figure 3, existing studies [37, 39–41, 43, 50, 53, 55, 62, 65, 66] mostly rely on a master-slave architecture, in which a "single" monolithic master is administering many applications (if any). The responsibilities include accepting new applications, parsing each application's DAG into stages, determining the number of parallel execution instances (tasks) under each stage, mapping these instances onto edge nodes, and tracking their progress.

This centralized architecture may run well for handling a small number of applications in the cloud. However, when it comes to IoT systems in the edge environment, new IoT users join and exit more frequently and launch a large number of IoT applications running at the same time, which makes the architecture easily become a scalability bottleneck and jeopardize the application's performance. This is because of (1) **high deployment latency**. These systems use a first-come, first-served approach to deploy applications, which causes applications to wait in a long queue and thus leads to long query latencies; and (2) **lack of flexibility for dataflow path planning**. They limit themselves to a fixed execution model and lack the flexibility to design different dataflow paths for different applications to adapt to the edge dynamics.

The limitation of the centralized architecture has been identified before in data processing frameworks such as YARN [63], Sparrow [51], Apollo [30]. They use two masters for task scheduling (one is the main master and one is the backup master). However, they remain fundamentally centralized [30, 51, 63] and restrict themselves to handle a small number of applications only.

3 Design

This section introduces DART's dynamic dataflow abstraction and shows how to scale up and down operators and perform failure recovery on top of this abstraction.

3.1 Overview

The DART system aims to achieve the following goals:

- **Low latency.** It achieves low latency for IoT queries.
- **Scalability.** It can process a large number of concurrently running applications at the same time.
- **Adaptivity.** It can adapt to the edge dynamics and recover from failures.

As shown in Figure 4, DART consists of three layers: *the DHT-based consistent ring overlay*, *the dynamic dataflow abstraction*, and *the scaling and failure recovery mechanisms*.

Layer 1: DHT-based consistent ring overlay. All distributed edge "nodes" (e.g., routers, gateways, or powerful sensors) are self-organized into a DHT-based overlay, which has been commonly used Bitcoin [48] and BitTorrent [35].

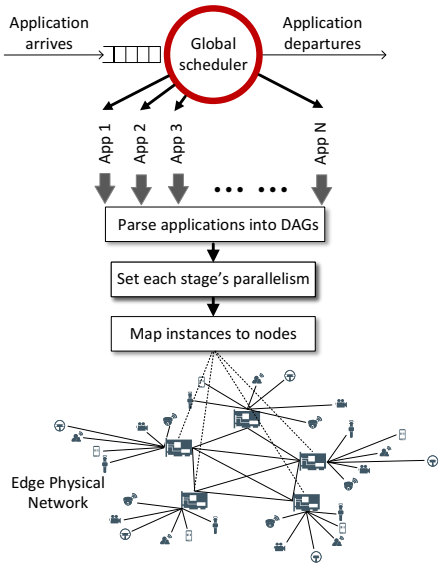


Figure 3: The global scheduler.

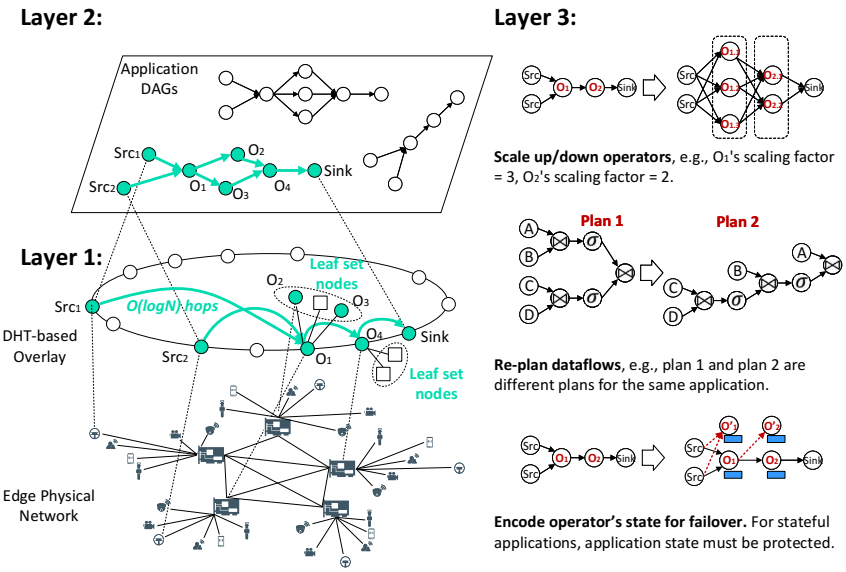


Figure 4: Dynamic dataflow graph abstraction for operator placement.

Each node is randomly assigned a unique "NodeId" in a large circular NodeId space. NodeIds are used to identify the nodes and route stream data. *No matter where the data is generated, it is guaranteed that the data can be routed to any destination node within $O(logN)$ hops.* To do that, each node needs to maintain two data structures: a routing table and a leaf set. The routing table is used for building dynamic dataflows. The leaf set is used for scaling and failure recovery.

Layer 2: Dynamic dataflow abstraction. Built upon the overlay, we introduce a novel dynamic dataflow abstraction. The key innovation is to leverage DHT-based routing protocols to approximate the optimal routes between source nodes and sink nodes, which can automatically place and chain operators to form a dataflow graph for each application.

Layer 3: Scaling and failure recovery mechanisms. Every node has a leaf set that contains physically "closest" nodes to this node. The leaf set provides the elasticity for (1) scaling up and down operators to adapt to the workload variations; (2) re-planning dataflows to adapt to the network variations. As stream data moves along the dataflow graph, the system makes dynamic decisions about the downstream node to send streams to, which increases network path diversity and becomes more resilient to changes in network conditions; and (3) replicating operators to handle failures and stragglers. If any node fails or becomes a straggler, the system can automatically switch over to a replica.

3.2 Dynamic Dataflow Abstraction

In the P2P model (e.g., Pastry [57], Chord [61]), each node is equal to the other nodes, and they have the same rights and duties. The primary purpose of the P2P model is to enable all nodes to work collaboratively to deliver a specific service. For example, in BitTorrent [35], if someone downloads some file, the file is downloaded to her computer in bits and parts that come from many other computers in the system that already have that file. At the same time, the file is also sent (uploaded) from her computer to others who ask for it. Similar to BitTorrent in which many machines work collaboratively to undertake the duties of downloading and uploading files, we enable all distributed edge nodes to work collaboratively to undertake the duties of the original monolithic master's.

Figure 5 shows the process of building the dynamic dataflow graph for an IoT stream application. *First*, we organize distributed edge nodes into a P2P overlay network, which is similar to the BitTorrent nodes that use the Kademila DHT [46] for "trackerless" torrents. Each node is randomly assigned a unique identifier known as the "NodeId" in a large circular node ID space (e.g., $0 \sim 2^{128}$). *Second*, given a stream application, we map the source operators to the sensors that generate the data streams. We map the sink operators to IoT actuators or message queues to the Cloud service. *Third*, every source node sends a `JOIN` message towards a key, where the key is the hash of the sink node's NodeId. Because all source nodes belonging to the same application have the same key, their messages will be routed to a rendezvous point—the sink node(s). Then we keep a record of the nodes that these messages pass through during routings and link them together to form the dataflow graph for this application.

To achieve low latency, the overlay guarantees that the stream data can be routed from source nodes to sink nodes within $O(logN)$ hops, thus ensuring the query latency upper bound. To achieve locality, the dynamic dataflow graph covers a set of nodes from sources to sinks. The first hop is always

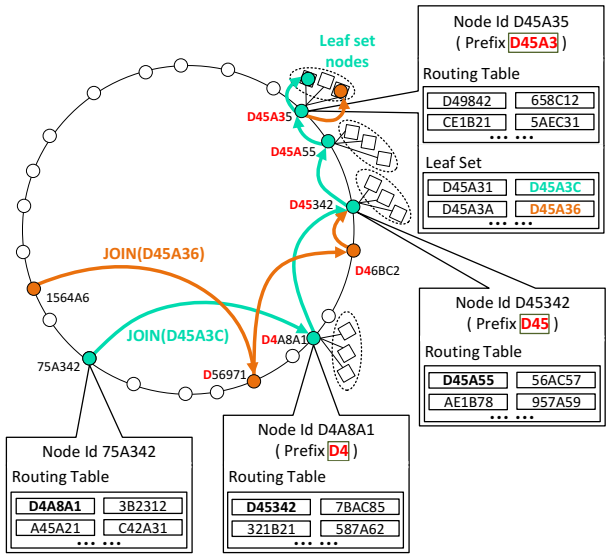


Figure 5: The process of building dynamic dataflow graph.

the node closer to the data source (data locality). Each node in the path has many leaf set nodes, which provides enough heterogeneous candidate nodes with different capacities and increases network path diversity. For example, if there are more operators than nodes, extra operators can map onto leaf set nodes. For that purpose, each node maintains two data structures: a routing table and a leaf set.

- *Routing table*: it consists of node characteristics organized in rows by the length of the common prefix. The routing works based on prefix-based matching. Every node knows m other nodes in the ring and the distance of the nodes it knows increases exponentially. It jumps closer and closer to the destination, like a greedy algorithm, within $\lceil log_{2^b} N \rceil$ hops. We add extra entries in the routing table to incorporate proximity metrics (e.g., hop count, RTT, cross-site link congestion level) in the routing process so as to handle the bandwidth variations.
- *Leaf set*: it contains a fixed number of nodes whose NodeIds are "physically" closest to that node, which assists in rebuilding routing tables and reconstructing the operator's state when any node fails.

As shown in Figure 5, node `75A342` and node `1564A6` are two source nodes and node `D45A3C` is the sink node. The source nodes route `JOIN` messages towards the sink node, and their messages are routed to a rendezvous point (s) — the sink node(s). We choose the forwarder nodes along routing paths based on RTT and node capacity. Afterward, we keep a record of the nodes that their messages pass through during routings (e.g., node `D4A8A1`, node `D45342`, node `D45A55`, node `D45A35`, node `D45A3C`), and reversely link them together to build the dataflow graph.

The key to efficiency comes from several factors. First, the application's instances can be instantly placed without the intervention of any centralized master, which benefits the time-critical deadline-based IoT application's queries. Second, because keys are different, the paths and the rendezvous nodes of all application's dataflow graphs will also be different, distributing operators evenly over the overlay, which significantly improves the scalability. Third, the DHT-based leaf set increases elasticity for handling failures and adapting to the bandwidth and workload variations.

3.3 Elastic Scaling Mechanism

After an application's operators are mapped onto the nodes along this application's dataflow graph, *how to auto-scale them to adapt to the edge dynamics?* We need to consider various factors. Scaling up/down is to increase/decrease the parallelism (#instances) of the operator within a node. Scaling out is to instantiate new instances on another node by redistributing the data streams across extra network links. In general, scaling up/down incurs smaller overhead. However, scaling out can solve the bandwidth bottleneck by increasing network path diversity, while scaling up/down may not.

We design a heuristic approach that adapts execution based on various factors. If there are computational bottlenecks, we scale up the problematic operators. The intuition is that when data queuing increases, automatically adding more instances to the system will avoid the bottleneck. We leverage the Secant root-finding method [29] to automatically calculate the optimal instance number based on the current system's health value. The policy is pluggable. Let $f(x)$ represent the health score based on the input rate and the queue size ($0 < f(x) < 1$, with 1 being the highest score). Let x_n and x_{n-1} be the number of instances during phases p_n and p_{n-1}. Then the number of instances required for the next phase p_{n+1} such that $f \cong 1$ can be given by:

$$x_{n+1} = x_n + (1 - f(x_n)) \times \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \qquad (1)$$

For bandwidth bottlenecks, we further consider whether the operator is stateless or stateful. In the case of stateless operators, we simply scale out operators across nodes. For stateful operators, we migrate the operator with its state to a new node in the leaf set that increases the network path diversity. Intuitively, when the original path only achieves low throughput, an operator may achieve higher throughput by sending the data over another network path.

3.4 Failure Recovery Mechanism

Since the overlay is self-organizing and self-repairing, the dataflow graph for each IoT application can be automatically recovered by restarting the failed operator on another node. Here, the challenge is, *how to resume the processing without losing intermediate data (i.e., operator state)*? Examples of operator states include keeping some aggregation or summary

of the received tuples in memory or keeping a state machine for detecting patterns for fraudulent financial transactions in memory. A general approach is checkpointing [7, 8, 54, 64], which periodically checkpoints all operators' states to a persistent storage system (e.g., HDFS) and the failover node retrieves the checkpointed state upon failures. This approach, however, is slow because it must transfer state over the edge networks that typically have very limited bandwidth.

We design a parallel recovery approach by leveraging the robustness of the P2P overlay and our previous experience in stateful stream processing [45]. Periodically, the larger-than-memory state is divided, replicated, and checkpointed to each node's leaf set nodes by using erasure codes [56]. Once any failure happens, the backup node takes over and retrieves state fragments from a subset of leaf set nodes to recompute state and resume processing. By doing that, we do not need a central master. The failure recovery process is fast because many nodes can leverage the dataflow graph to recompute the lost state in parallel upon failures. The replica number, the checkpointing frequency, the number of encoded blocks and the number of raw blocks are tunable parameters. They are determined based on state size, running environment and the application's service-level agreements (SLAs.)

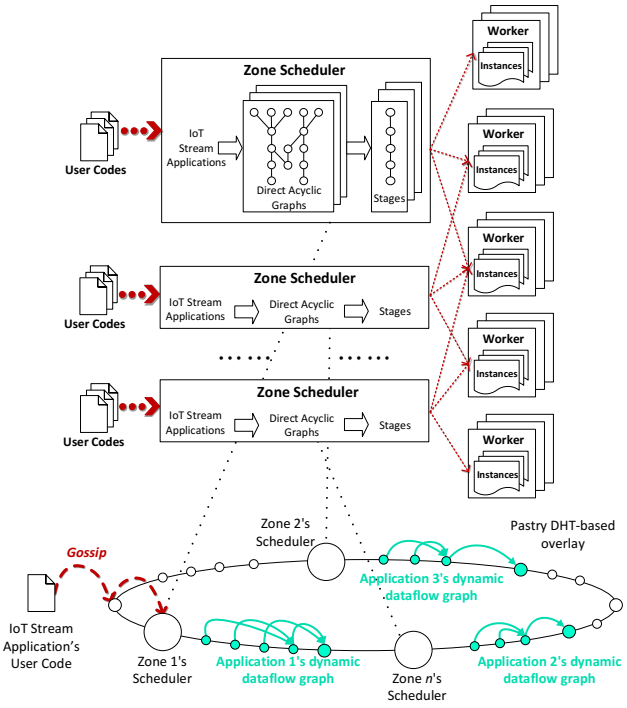


Figure 6: The DART system architecture.

4 Implementation

Instead of implementing another distributed system core, we implement DART on top of Apache Flume [3] (v.1.9.0) and Pastry [16] (v.2.1) software stacks. Flume is a distributed service for collecting and aggregating large amounts of streaming event data, which is widely used with Kafka [4] and the Spark ecosystem. Pastry is an overlay network and routing network for the implementation of a distributed hash table (DHT) similar to Chord [61], which is widely used in applications such as Bitcoin [48], BitTorrent [35], and FAROO [15]. We leverage Flume's excellent runtime system (e.g., basic API, code interpreter, transportation layer) and Pastry's routing substrate and event transport layer to implement the DART system.

We made three major modifications to Flume and Pastry: (1) we implemented the dynamic dataflow abstraction for operator placement and path planning algorithm, which includes a list of operations to track the DHT routing paths for chaining operators and a list of operations to capture the performance metrics of nodes for placing operators; (2) we implemented the scaling mechanism and the failure recovery mechanism by introducing queuing-related metrics (queue length, input rate, and output rate), buffering operator's in-memory state, encoding and replicating state to leaf set nodes; and (3) we implemented the distributed schedulers by using Scribe [33] topic-based trees on top of Pastry.

Figure 6 shows the high-level architecture of the DART system. The system has two components: a set of distributed schedulers that span geographical zones and a set of workers. Unlike traditional stream processing systems that manually assign nodes as "master" or "workers", DART dynamically assigns nodes as "schedulers" or "workers". For the first step, when any new IoT stream application is launched, it looks for a nearby scheduler by using the gossip protocol [31], which is a procedure of P2P communication that is based on the way that epidemics spread. If it successfully finds a scheduler within $log(N)$ hops, the application registers itself to this scheduler. Otherwise, it votes any random nearby node to be the scheduler and registers itself to that scheduler. For the second step, the scheduler processes this application's queries by parsing the application's user code into a DAG and dividing this DAG into stages. Then the scheduler automatically parallelizes, chains operators, and places the instances on edge nodes using *the proposed dynamic dataflow abstraction*. These nodes are then set as this application's workers. The system automatically scales up and out operators, re-plans, and replicates operators to adapt to the edge dynamics and recover from failures by using *the proposed scaling mechanism and failure recovery mechanism*.

The key to efficiency comes from several factors. First, all nodes in the system are equal peers with the same rights and duties. Each node may act as one application's worker, another application's worker, a zone's scheduler, or any combination of the above, resulting in all load being evenly distributed. Second, the scheduler is no longer any central bottleneck. Third, the system automatically creates more schedulers for application intensive zones and fewer ones for sparse zones, thus scaling to extremely large numbers of nodes and applications.

5 Evaluation

We evaluate DART on a real hardware testbed (using Raspberry Pis) and emulation testbed in a distributed network environment. We explore its performance for real-world IoT stream applications. Our evaluation answers these questions:

- Does DART improve latency when processing a large number of IoT stream applications?
- Does DART scale with the number of concurrently running IoT stream applications?
- Does DART improve adaptivity in the presence of workload changes, transient failures and mobility?
- What is the runtime overhead of DART?

5.1 Setup

Real hardware. Real hardware experiments use an intermediate class computing device representative of IoT edge devices. Specifically, we use 10 Raspberry Pi 4 Model B devices for hosting source operators, each of which has a 1.5GHz 64-bit quad-core ARMv8 CPU with 4GB of RAM and runs Linux raspberrypi 4.19.57. Raspberry Pis are equipped with Gigabit Ethernet Dual-band Wi-Fi. We use 100 Linux virtual machines (VMs) to represent the gateways and routers for hosting internal and sink operators, each of which has a quad-core processor and 1GB of RAM (equivalent to Cisco's IoT gateway [11]). These VMs are connected through a local-area network. In order to make our experiments closer to real edge network scenarios, we used the TC tool [17] to control link bandwidth differences.

Emulation deployment. Emulation experiments are conducted on a testbed of 100 VMs running Linux 3.10.0, all connected via Gigabit Ethernet. Each VM has 4 cores and 8GB of RAM, and 60GB disk. Specifically, to evaluate DART's scalability, we use one JVM to emulate one logical edge node and can emulate up to 10,000 edge nodes in our testbed.

Baseline. We used Storm and EdgeWise [37] as the edge stream processing engine baseline. Apache Storm version is 2.0.0 [7] and EdgeWise [37] is downloaded from GitHub [14]. Both of them are configured with 10 TaskManagers, each with 4 slots (maximum parallelism per operator = 36). We run Nimbus and ZooKeeper [9] on the VMs and run supervisors on the Raspberry Pis. We use Pastry 2.1 [57] configured with leaf set size of 24, max open sockets of 5000 and transport buffer size of 6 MB.

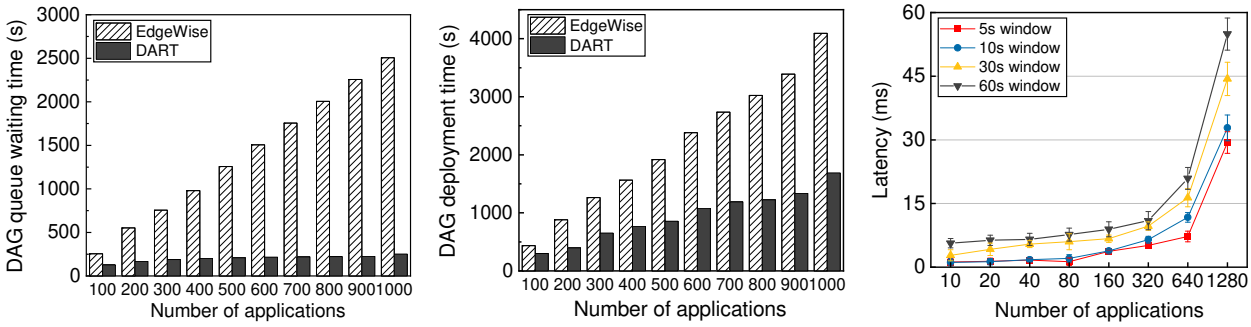
Benchmark and applications. We deploy a large number of applications (topologies) simultaneously to demonstrate the scalability of our system. The applications in the mixed set are chosen from a full-stack standard IoT stream processing benchmark [60]. We also implement four IoT stream processing applications that use real-world datasets [12, 13, 24, 47]. They employ various techniques such as predictive analysis, model training, data preprocessing, and statistical summarization. Their operators run functions such as `transform`, `filter`, `flatmap`, `aggregate`, `duplicate`, and `hash`. For example, we implement the DEBS 2015 application [13] to process spatio-temporal data streams and calculate real-time indicators of the most frequent routes and most profitable areas in New York City. The sensor data consists of taxi trip reports that include start and drop-off points, corresponding timestamps, and payment information. Data are reported at the end of the trip. Although the prediction tasks available in this application do not require real-time responses, it captures the data dissemination and query patterns of more complex upcoming transportation engines. An application that integrates additional data sources – bus, subway, car-for-hire (e.g., Uber), ride-sharing, traffic, and weather conditions – would exhibit the same structural topology and query rates that we use in our experiments while offering decision-making support in the scale of seconds. We implement the Urban sensing application [12] to aggregate pollution, dust, light, sound, temperature, and humidity data across seven cities to understand urban environmental changes in real-time. Since a practical deployment of environmental sensing can easily extend to thousands of such sensors per city, a temporal scaling of $1000\times$ the native input rate can be used to simulate a larger deployment of 90,000 sensors.

Metrics. We focus on the performance metrics of query latency. Query latency is measured by sampling 5% of the tuples, assigning each tuple a unique ID and comparing timestamps at source and the same sink. To evaluate the scalability of DART, we measure how operators are distributed over nodes and how distributed schedulers are distributed over zones. To evaluate the adaptivity of DART, we cause bottlenecks by intentionally adding resource contention and we intentionally disable nodes through human intervention.

5.2 Query Latency

We measure the query latencies for running real-world IoT stream applications on the Raspberry Pis and VMs across a wide range of input rates.

Figure 7a and Figure 7b show the latency comparison of DART vs EdgeWise for (a) DAG queue waiting time and (b) DAG deployment time for an increasing number of concurrently running applications. We choose applications from a pool that contains dataflow topologies (DAGs) including `ExclamationTopology`, `JoinBoltExample`, `LambdaTopology`, `Prefix`, `SingleJoinExample`, `SlidingTupleTsTopology`, `SlidingWindowTopology` and `WordCountTopology`. EdgeWise is built on top of Storm. Both of them rely on a centralized master (Nimbus) to deploy the application's DAGs, and then process them one by one on a first-come, first-served basis. Therefore, we can see that EdgeWise's DAG queue waiting time and deployment time increase linearly as the number of applications increases. As such, the centralized master will easily become a scalability bottleneck. In contrast,

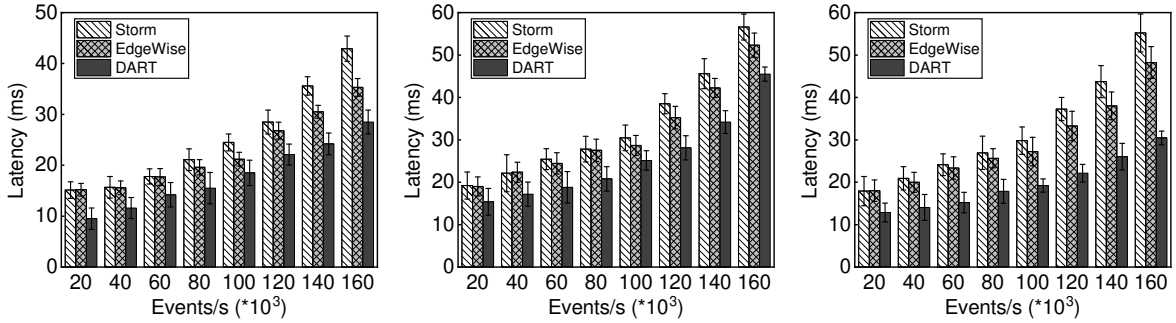


(a) DAG queue waiting time comparison of DART vs EdgeWise.

(b) DAG deployment time comparison of DART vs EdgeWise.

(c) Query processing time.

Figure 7: The latency comparison of DART vs EDGEWISE for (a) DAG queue waiting time, (b) DAG deployment time, and (c) query processing time by increasing the number of concurrently running applications.



(a) Calculating the frequent route in the taxi application.

(b) Calculating the most profit area in the taxi application.

(c) Visualizing environmental changes in the urban sensing application.

Figure 8: The latency comparison of DART vs Storm vs EdgeWise for (a) frequent route application, (b) profitable area application, and (c) urban sensing application.

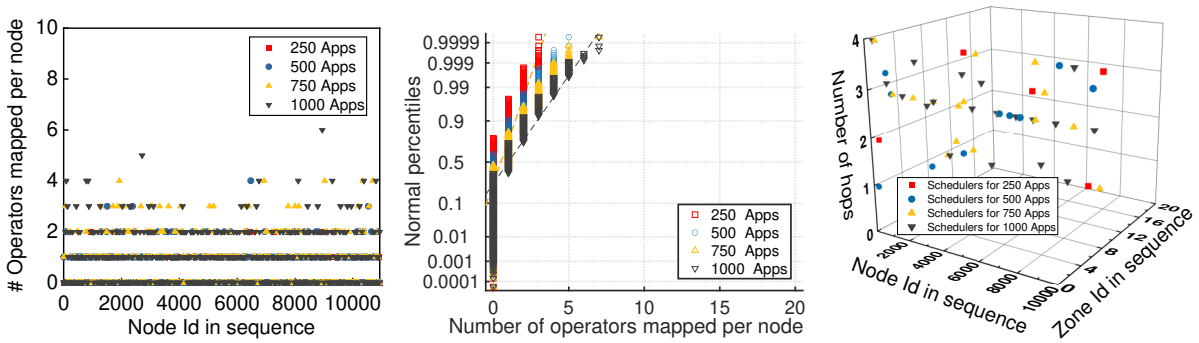
DART avoids this scalability bottleneck because DART's decentralized architecture does not rely on any centralized master to analyze the DAGs and deploy DAGs.

Figure 7c shows the query latency of DART for an increasing number of concurrently running applications. Results show that DART scales well with a large number of concurrently running applications. First, DART's distributed schedulers can process these applications' queries independently, thus precluding them from queuing on a single central scheduler which results in large queuing delay. This is similar to the idea that supermarkets add cashiers to reduce the waiting queues when there are many people in supermarkets. Second, DART's P2P model ensures that every available node in the system can participate in the process of operator mapping, auto-scaling, and failure recovery, which could avoid the central bottleneck, balance the workload, and speed up the process.

The performance comparison results for running the frequent route application, the profitable areas application, and the urban sensing application are shown in Figure 8. In general, DART, Storm and EdgeWise [37] have similar performance when the system is under-utilized (with low input). When the system is averagely utilized (with relatively high input), DART achieves around 16.7% $\sim$ 52.7% less query latency compared to Storm. DART achieves 9.8 % $\sim$ 45.6% less query latency compared to EdgeWise. This is because DART limits the number of hops between the source operators to sink operators within $log(N)$ hops by using the DHT-based consistent ring overlay, and DART can dynamically scale operators when input rate changes. DART has better performance in the urban sensing application because this application needs to split data into different channels and aggregate data from these channels, which results in a lot of I/Os and data transfers that can benefit from DART's dynamic dataflow abstraction. We expect further latency improvement under a limited bandwidth environment since DART selects the path with less traffic for the data flow by using the path planning algorithm.

5.3 Scalability Analysis

We now show scalability: DART decomposes the traditional centralized architecture of stream processing engines into a new decentralized architecture for operator mapping and query scheduling, which dramatically improves the scalability



(a) The distribution of DART's operators over different edge nodes.

(b) Normal probability plot of the number of operators mapped per node.

(c) The distribution of DART's schedulers over different zones.

Figure 9: Scalability study of DART for the distribution of operators and schedulers over edge nodes.

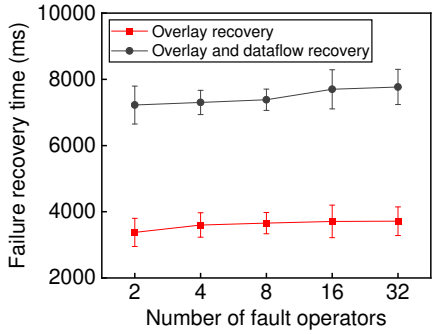


Figure 10: Overlay and dataflow recovery time.

for the system to scale with a large number of concurrently running applications, application's operators, and zones.

Figure 9a shows the mappings of DART's operators on edge nodes for 250, 500, 750, and 1,000 concurrently running applications, respectively. These applications run a mix of topologies with different numbers of operators (an average value of 10). Figure 9b shows the normal probability plot of the number of operators per node. Results show that when deploying 250 and 500 applications, around 96.52% nodes host less than 3 operators; and when deploying 750 and 1000 applications, around 99.84% nodes host less than 4 operators. From Figure 9a and Figure 9b, we can see that these applications' operators are evenly distributed on all edge nodes. This is because DART essentially leverages the DHT routing to map operators on edge nodes. Since the application's dataflow topologies are different, their routing paths and the rendezvous points will also be different, resulting in operators well balanced across all edge nodes.

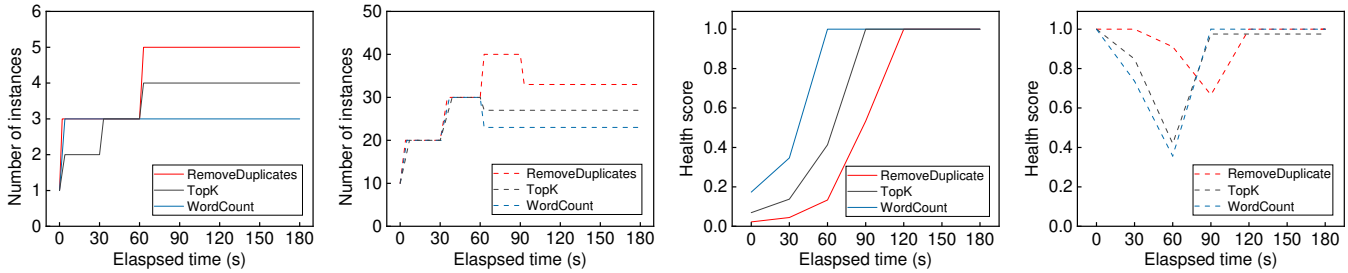
Figure 9c shows the mappings of DART's distributed schedulers on edge nodes and zones for 250, 500, 750 and 1,000 concurrently running applications, and the average number of hops for these applications to look for a scheduler. For DART, it adds a scheduler for every new 50 applications. According to the P2P's gossip protocol, each application looks for a scheduler in the zone within $\lceil log_{2^b} N \rceil$ hops, where $b = 4$. If there is no scheduler in the zone or the number of applications in the zone exceeds a certain threshold, a peer node (usually with powerful computing resources) will be elected as a new scheduler. Results show that as the number of concurrently running applications increases, the number of schedulers over zones increases accordingly. All schedulers are evenly distributed over different zones. Most of the schedulers can be searched within 4 hops.

The above results demonstrate DART's load balance and scalability properties: (1) by using DHT-based consist ring overlay, the IoT stream application's workloads are well distributed over all edge nodes; and (2) DART can scale well with the number of zones and concurrently running applications.

5.4 Failure Recovery Analysis

We next show fault tolerance: in the case of stateless IoT applications, DART simply resumes the whole execution pipeline since there is no need for recovering state. In the case of stateful IoT applications, distributed states in operators are continuously checkpointed to the leaf set nodes in parallel and are reconstructed upon failures. We show that even when many nodes fail or leave the system, DART can achieve a relatively stable time to recover the overlay and dataflow topology.

Figure 10 shows the overlay recovery time and the dataflow topology recovery time for an increasing number of simultaneous operator failures. To cause simultaneous failures, we deliberately remove some working nodes from the overlay and evaluate the time for DART to recover. The time cost includes recomputing the routing table entries, re-planning the dataflow path, synchronizing operators, and resuming the computation. Results show that DART achieves a stable recovery time for an increasing number of simultaneous failures. This is because, in DART, each failed node can be quickly detected and recovered by its neighbors through heartbeat messages without having to talk to a central coordinator, so many simultaneous failures can be repaired in parallel.



(a) Process of scaling up.

(b) Process of scaling up and scaling out.

(c) Health score changes corresponding to Figure 11a.

(d) Health score changes corresponding to Figure 11b.

Figure 11: Adaptivity study of DART for the scaling up and the scaling out.

5.5 Elastic Scaling Analysis

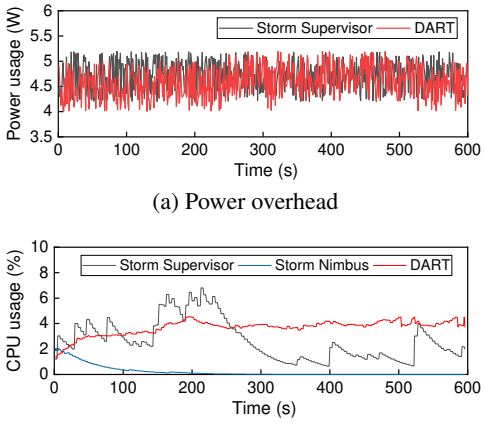
Although scaling is a subject that has been studied for a long time, our innovation is that we use the DHT leaf set to select the best candidate nodes for scaling up or scaling out. Therefore, our approach does not need a central master to control, which is fully distributed. If many operators have bottlenecks at the same time, the system can adjust them all together. The periodical maintenance and update of the leaf set ensure that the leaf set nodes are good candidates, which are close to the bottleneck operator with abundant bandwidth, so there is no need for us to search for the appropriate nodes globally.

The auto-scaling process takes the system snapshot collected every 30 seconds for statistical analysis. We deploy three 4-stage topologies (`RemoveDuplicates`, `TopK`, `WordCount`). Figure 11a shows the process of scaling up only. The process starts from the moment of detecting the bottleneck to the moment that the system is stabilized. Figure 11b shows the process of scaling up and then scaling out. For this experiment, we put pressure on the system by gradually increasing the number of instances (tasks) (10 every 30 s) until a bandwidth bottleneck occurs (at 60 s for the blue line and the black line, and at 90 s for the red line). This bottleneck can only be resolved by scaling up. Results show that the system is stabilized by migrating the instance to another node.

Figure 11c shows how the health score changes corresponding to Figure 11a. Figure 11d shows how the health score changes corresponding to Figure 11b. Note that if the goal of pursuing a higher health score conflicts with the goal of improving throughput, we need to strike a balance between health score and system throughput by adjusting the health score function, i.e., aiming at a lower score.

5.6 Overhead Analysis

We evaluate the DART's runtime overhead in terms of the power usage and the CPU overhead. We run the same DEBS 2015 application [13] in Sec. 5.2 to calculate real-time indicators of most frequent routes in New York City with source rate at 100K $events/s$.



(a) Power overhead

(b) CPU overhead

Figure 12: Overhead comparison of DART vs Storm.

Power usage. Most IoT devices rely on batteries or energy harvesters. Given that their energy budget is limited, we want to ensure that the performance gains achieved come with an acceptable cost in terms of power consumption. To evaluate DART's power usage, we use the MakerHawk USB Power Meter Tester [18] to measure the power usage of the Raspberry Pi 4. When plugged into a wall socket, the idle power usage is 3.35 Watt. Figure 12a shows the comparison of the averaged single device per-node power usage of DART node with Storm's supervisor when running the DEBS 2015 application. Results show that DART has less power usage with an average value of 5.24 Watt compared to Storm with an average value of 5.41 Watt, demonstrating that DART efficiently uses energy resources.

CPU overhead. Figure 12b shows the comparison of the CPU overhead of DART with Storm. Results show that DART uses more CPU than Storm Nimbus and Storm supervisor. DART continuously monitors the health status of all operators to make auto-scaling decisions to adapt to workload variations and bandwidth variations in the edge environment, while Storm ignores it. This CPU overhead is an acceptable trade-off for maintaining performance and could be further reduced with a larger auto-scaling interval.

6 Related Work

Existing studies can be divided into four categories: *cluster-based stream processing systems*, *cloud-based IoT data processing systems*, *edge-based data processing systems*, and *wide-area data analytics systems*.

Category 1: Cluster-based stream processing systems. Over the last decade, a bloom of industry stream processing systems has been developed including Flink [2], Samza [5], Spark [6], Storm [7], Millwheel [28], Heron [44], S4 [49]. These systems, however, are designed for low-latency intra-datacenter settings that have powerful computing resources and stable high-bandwidth connectivity, making them unsuitable for edge stream processing. Moreover, they mostly inherit MapReduce's "single master/many workers" architecture that relies on a monolithic scheduler for scheduling all tasks and handling failures and stragglers, suffering significant shortcomings due to the centralized bottleneck. SBONs [52] leverages distributed hash table (DHTs) for service placement. However, it does not support DAG parsing, task scheduling, data shuffling and elastic scaling, which are required for modern stream processing engines.

Category 2: Cloud-based IoT data processing systems. In such a model, most of the data is sent to the cloud for analysis. Today many computationally-intensive IoT applications [22, 23] leverage this model because cloud environments can offer unlimited computational resources. Such solutions, however, cannot be applied to time-critical IoT stream applications because: (1) they cause long delays and strain the backhaul network bandwidth; and (2) offloading sensitive data to third-party cloud providers may cause privacy issues.

Category 3: Edge-based data processing systems. In such a model, data processing is performed at the edge without connectivity to a cloud backend [19, 21, 58]. This requires installing a hub device at the edge to collect data from other IoT devices and perform data processing. These solutions, however, are limited by the computational capabilities of the hub service and cannot support distributed data-parallel processing across many devices and thus have limited throughput. It may also introduce a single point of failure once the hub device fails.

Category 4: Wide-area data analytics systems. Many Apache Spark-based systems (e.g., Flutter [39], Iridium [53], JetStream [55], SAGE [62], and many others [40, 41, 43, 65, 66]) are proposed for enabling geo-distributed stream processing in wide-area networks. They optimize the execution by intelligently assigning individual tasks to the best datacenters (e.g., more data locality) or moving data sets to the best datacenters (e.g., more bandwidth). However, they make certain assumptions based on some theoretical models which do not always hold in practice. For example, Flutter [39], Tetrium [40], Iridium [53], Clarinet [65], and Geode [66] formulate the task scheduling problem as a ILP problem. Pixida [43] formulates the task scheduling problem as a Min k-Cut problem. They assume that the workload, the inter-DC transfer time, and the WAN bandwidth are known beforehand and do not change, which is rarely the case in practice. Moreover, these systems also suffer significant shortcomings due to the centralized bottleneck.

To our best knowledge, Edgent [1], EdgeWise [37], and Frontier [50] are the only other stream processing engines tailored for the edge. They all point out the criticality of edge stream processing, but no effective solutions were proposed towards scalable and adaptive edge stream processing. Edgent [1] is designed for data processing at individual IoT devices rather than full-fledged distributed stream processing. EdgeWise [37] develops a congestion-aware scheduler to reduce backpressure, but it can not scale well due to the centralized bottleneck. Frontier [50] develops replicated dataflow graphs for fault-tolerance, but it ignores the edge dynamics and heterogeneity.

7 Conclusion

Existing stream processing engines were designed for the cloud environments and may behave poorly in the edge context. In this paper, we present DART, a scalable and adaptive edge stream processing engine that enables fast stream processing for a large number of concurrent running IoT applications in the dynamic edge environment. DART leverages DHT-based P2P overlay networks to create a decentralized architecture and design a dynamic dataflow abstraction to automatically place, chain, scale, and recover stream operators, which significantly improves performance, scalability, and adaptivity for handling large IoT stream applications.

An interesting question for future work is how to optimize data shuffling services for edge stream processing engines like DART. Common operators such as `union` and `join` may require intermediate data to be transmitted over edge networks since their inputs are generated at different locations. Each shard of the shuffle data has to go through a long path of data serialization, disk I/O, edge networks, and data deserialization. Shuffle, if planned poorly, may delay the query processing. We plan to explore a customizable shuffle library that can customize the data shuffling path (e.g., ring shuffle, hierarchical tree shuffle, butterfly wrap shuffle) at runtime to optimize shuffling. We will release DART as open source, together with the data used to produce the results in this paper[1].

8 Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Dr. Amy Lynn Murphy, for their insightful suggestions and comments that improved this paper. This work is supported by the National Science Foundation (NSF-CAREER-1943071, NSF-SPX-1919126, NSF-SPX-1919181).

[1] https://github.com/fiu-elves/DART

References

[1] Apache Edgent - A Community for Accelerating Analytics at the Edge. https://edgent.apache.org/.

[2] Apache Flink. https://flink.apache.org/.

[3] Apache Flume. http://flume.apache.org/.

[4] Apache Kafka. https://kafka.apache.org/.

[5] Apache Samza. http://samza.apache.org/.

[6] Apache Spark. https://spark.apache.org/.

[7] Apache Storm. http://storm.apache.org/.

[8] Apache Trident. http://storm.apache.org/releases/current/Trident-tutorial.html.

[9] Apache ZooKeeper. https://zookeeper.apache.org/.

[10] AVA: Automated Vehicles for All. https://www.transportation.gov/policy-initiatives/automated-vehicles/10-texas-am-engineering-experiment-station.

[11] Cisco Kinetic Edge & Fog Processing Module (EFM). https://www.cisco.com/c/dam/en/us/solutions/collateral/internet-of-things/kinetic-datasheet-efm.pdf.

[12] Data Canvas: Sense Your City. https://grayarea.org/initiative/data-canvas-sense-your-city/.

[13] DEBS 2015 Grand Challenge: Taxi trips. https://debs.org/grand-challenges/2015/.

[14] EdgeWise source code. https://github.com/XinweiFu/EdgeWise-ATC-19.

[15] FAROO - Peer-to-peer Web Search: History. http://faroo.com/.

[16] FreePastry. https://www.freepastry.org/.

[17] Linux Traffic Control. https://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html.

[18] MakerHawk USB Power Meter Tester. https://www.makerhawk.com/products/.

[19] Microsoft Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge.

[20] A new reality for oil & gas. https://www.cisco.com/c/dam/en_us/solutions/industries/energy/docs/OilGasDigitalTransformationWhitePaper.pdf, 2017.

[21] Amazon AWS Greengrass. https://aws.amazon.com/greengrass, 2017.

[22] Google Nest Cam. https://nest.com/cameras, 2017.

[23] Netatmo. https://www.netatmo.com, 2017.

[24] Soil Moisture Profiles and Temperature Data from SoilSCAPE Sites. https://daac.ornl.gov/LAND_VAL/guides/SoilSCAPE.html, 2017.

[25] Autonomous cars will generate more than 300 tb of data per year. https://www.tuxera.com/blog/autonomous-cars-300-tb-of-data-per-year/, 2019.

[26] HORTONWORKS: iot and predictive big data analytics for oil and gas. https://hortonworks.com/solutions/oil-gas/, 2019.

[27] The ITS JPO's New Strategic Plan 2020-2025. https://www.its.dot.gov/stratplan2020/index.htm, 2020.

[28] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[29] Mordecai Avriel. *Nonlinear programming: analysis and methods*. Courier Corporation, 2003.

[30] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 285–300, Berkeley, CA, USA, 2014. USENIX Association.

[31] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE/ACM Trans. Netw.*, 14(SI):2508–2530, June 2006.

[32] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[33] M. Castro, P. Druschel, A. . Kermarrec, and A. I. T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, Oct 2002.

[34] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[35] Federico Concone, Alessandra De Paola, Giuseppe Lo Re, and Marco Morana. Twitter analysis for real-time malware discovery. In *AEIT International Annual Conference, 2017*, pages 1–6. IEEE, 2017.

[36] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow.*, 10(12):1825–1836, aug 2017.

[37] Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee. Edgewise: A better stream processing engine for the edge. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019. USENIX Association.

[38] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, dec 2012.

[39] Z Hu, B Li, and J Luo. Flutter: Scheduling tasks closer to data across geo-distributed datacenters. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, apr 2016.

[40] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 12:1—-12:16, New York, NY, USA, 2018. ACM.

[41] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. Scheduling Jobs Across Geo-distributed Datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 111–124, New York, NY, USA, 2015. ACM.

[42] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, Carlsbad, CA, 2018. USENIX Association.

[43] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. Pixida: Optimizing data parallel jobs in wide-area data analytics. *Proc. VLDB Endow.*, 9(2):72–83, oct 2015.

[44] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[45] Pinchao Liu, Hailu Xu, Dilma Da Silva, Qingyang Wang, Sarker Tanzir Ahmed, and Liting Hu. Fp4s: Fragment-based parallel state recovery for stateful stream applications. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020.

[46] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[47] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. Predicting taxi passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, Sep. 2013.

[48] Satoshi Nakamoto. Bitcoin : A Peer-to-Peer Electronic Cash System. Technical report, 2008.

[49] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed Stream Computing Platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[50] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch. Frontier: resilient edge processing for the internet of things. *Proceedings of the VLDB Endowment*, 11:1178–1191, 2018.

[51] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[52] Peter Pietzuch, Jeffrey Shneidman, Jonathan Ledlie, Matt Welsh, Margo Seltzer, and Mema Roussopoulos. Evaluating dht-based service placement for stream-based overlays. In *International Workshop on Peer-to-Peer Systems*, pages 275–286. Springer, 2005.

[53] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 421–434, New York, NY, USA, 2015. ACM.

[54] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14, New York, NY, USA, 2013. ACM.

[55] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 275–288, Berkeley, CA, USA, 2014. USENIX Association.

[56] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[57] Antony I T Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware'01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[58] Zhitao Shen, Vikram Kumaran, Michael J Franklin, Sailesh Krishnamurthy, and Amit Bhat. CSA : Streaming Engine for Internet of Things. *Bulletin of the Technical Committee on Data Engineering*, 38(4):39–50, 2015.

[59] W Shi, J Cao, Q Zhang, Y Li, and L Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, oct 2016.

[60] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017.

[61] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[62] R Tudoran, G Antoniu, and L Bouge. SAGE: Geo-Distributed Streaming Data Analysis in Clouds. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 2278–2281, may 2013.

[63] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1———5:16, New York, NY, USA, 2013. ACM.

[64] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.

[65] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. CLARINET: Wan-aware optimization for analytics queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 435–450, Savannah, GA, 2016. USENIX Association.

[66] Ashish Vulimiri, Carlo Curino, P Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 323–336, Berkeley, CA, USA, 2015. USENIX Association.