SDN-enabled Resource Provisioning Framework for Geo-Distributed Streaming Analytics

HABIB MOSTAFAEI, Eindhoven University of Technology, The Netherlands
SHAFI AFRIDI, Technische Universität Berlin, Germany

Geographically distributed (geo-distributed) datacenters for stream data processing typically comprise multiple edges and core datacenters connected through **Wide-Area Network (WAN)** with a master node responsible for allocating tasks to worker nodes. Since WAN links significantly impact the performance of distributed task execution, the existing task assignment approach is unsuitable for distributed stream data processing with low latency and high throughput demand. In this paper, we propose SAFA, a resource provisioning framework using the **Software-Defined Networking (SDN)** concept with an SDN controller responsible for monitoring the WAN, selecting an appropriate subset of worker nodes, and assigning tasks to the designated worker nodes. We implemented the data plane of the framework in P4 and the control plane components in Python. We tested the performance of the proposed system on Apache Spark, Apache Storm, and Apache Flink using the Yahoo! streaming benchmark on a set of custom topologies. The results of the experiments validate that the proposed approach is viable for distributed stream processing and confirm that it can improve at least 1.64× the processing time of incoming events of the current stream processing systems.

CCS Concepts: • **Networks** → **Programmable networks**; **Cloud computing**; **Network monitoring;**

Additional Key Words and Phrases: Cluster manager, geo-distributed stream analytics, stream processing, Software-Defined Networking (SDN)

ACM Reference format:
Habib Mostafaei and Shafi Afridi. 2023. SDN-enabled Resource Provisioning Framework for Geo-Distributed Streaming Analytics. *ACM Trans. Internet Technol.* 23, 1, Article 18 (February 2023), 21 pages.
https://doi.org/10.1145/3571158

1 INTRODUCTION

Information technology advancements have led to the proliferation of datacenters that provide Internet-based services and large-scale stream data processing capacity. Emerging applications such as social networks and the Internet of Things generate a large number of datasets from various geographically distributed (geo-distributed) locations. These internet-scale applications have necessitated the use of geo-distributed datacenters **(distributed processing systems (DPS))** for analyzing the distributed data to extract insights [48]. As a result, many cloud service providers

Part of this work has been conducted while Habib Mostafaei was with Technische Universität Berlin.
This work was partially funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).
Authors' addresses: H. Mostafaei, Eindhoven University of Technology, De Zaale, 05 MetaForum P.O.Box 513 5600 MB Eindhoven, Eindhoven, The Netherlands; email: h.mostafaei@tue.nl; S. Afridi, Technische Universität Berlin, FG INET, EN 18, Einsteinufer 17, 10587 Berlin, Germany.

1533-5399/2023/02-ART18
https://doi.org/10.1145/3571158

such as Google, Amazon, and Microsoft offer geo-distributed datacenters to cater to local users. While analytics over data distributed across different datacenters is increasingly becoming common, achieving low latency analytics has become a critical challenge. Several solutions have been proposed in the literature [36, 43, 44] to address this challenge. These solutions are mostly focused on processing data in a batch fashion. Specifically, they focus on the task or data placements on currently available computing powers in the datacenters. Recent attempts in [17, 25, 26, 47] process stream of data in geographically dispersed datacenters. These works focus on different aspects of geo-distributed analytics, such as minimizing the task execution latency or efficient WAN bandwidth usage.

The common thread among the DPSs is that they all rely on built-in or third-party cluster managers like Apache Mesos or Hadoop-Yarn to manage the physical resources within the cluster in the datacenter. Also, they have trailed solutions mostly based on a modified version of standard DPSs and thus are inappropriate for general use. Other approaches like [2] consider utilizing SDN to optimize the available bandwidth among the running stream processing tasks in a datacenter. Nevertheless, this work cannot be extended to the geo-distributed scenarios. These approaches use the standard **Wide-Area Network (WAN)** to interconnect geographically dispersed DPSs. For example, the data should cross the error-prone links with limited WAN links bandwidth [1] that reduces the amount of data transmission. Therefore, the classic WAN suffers several drawbacks, making it unfavorable for geo-distributed stream data processing [20]. Moreover, they are unaware of the underlying network infrastructure [36]; thus, the performance of the DPSs can suffer from issues associated with WAN.

Moreover, prior work on batch and stream processing systems modifies the source of a specific version of each DPS to improve performance [27]. The modified versions are application-specific and do not get integrated into the DPSs framework. Furthermore, the community of each DPS releases a new version typically in an interval of less than a year, possibly with a new set of features. These rapid developments hinder the integration of the customized version with the community version.

The DSPs have to process many data flows in geo-distributed streaming scenarios. These data flows can carry critical information such as business transaction data that cross WAN links with diverse properties such as latency and bandwidth. Knowing link property information can lead to an efficient task execution in geo-distributed settings. However, obtaining such information dynamically is not straightforward for the whole network since the network operators need to develop several scripts for monitoring the network. In addition, these scripts are prone to fail due to changes in the network, like link removals. We can overcome these problems by leveraging the **Software-Defined Network (SDN)** advantages and offloading the collection of such information to the controller. Then, we can develop **Application Programming Interfaces (APIs)** to interact with the DSPs to handle their network.

In this paper, we propose a framework based on the SDN to relieve the drawbacks that inhibit using the classic WAN for stream data processing. The proposed framework, unlike the existing solutions which modify the DPSs to a customize worker node placement, does not alter the underlying DSP, and thus, it is not bounded to a specific version of a DPS. Moreover, SAFA contains components that monitor the network and several selection algorithms used to place the worker nodes in an appropriate location based on the underlying network topology of the cluster and handle a geo-distributed cluster. We summarize our contributions as follows:

- We propose an SDN-based architecture for geo-distributed big data streaming analytics systems that is integrable with popular DSPs;
- We propose two worker node selection algorithms to minimize the query execution latency while considering WAN links related parameters such as the delay;

- We evaluate the performance of the selection algorithms on all networks taken from TopologyZoo and three custom topologies. We report that the proposed selection algorithms improve 2.4× the inter-node link latency compared with the current default algorithm; and
- We evaluate the impact of WAN link delays using the **Yahoo! streaming benchmark (YSB)** [13] on Apache Spark, Apache Storm, and Apache Flink. Additionally, our delay-based ranking algorithm improves 2.6× the performance of Flink on a custom topology within WAN link delays in the range of 26 to 75ms for each link similar to [34].

The remaining sections of the paper are as follows. Section 2 briefly reports the current works in the area of geo-distributed analytics systems. Section 3 presents the background and motivations of our work in distributed stream processing systems. The placement problem is formulated in Section 4. The architecture of SAFA with the placement algorithms detail are presented in Section 5. Section 6 states the proof-of-concept implementation of our system. Section 7 reports the evaluations of our system on custom and real-networks. Section 8 concludes this work.

2 RELATED WORK

Running big data analytics over the geo-distributed datacenters has become a common solution to process the huge amount of data generated by applications [11, 21, 27, 33, 36, 44, 47]. Resource provisioning in such systems is one of the key problems to ensure that the deployed service can meet the SLAs and lead to a minimized cost [28]. The nature of WAN links connecting these systems adds further complexity to managing them. For example, transferring data in such a network can incur a high cost [35]. The survey in [28] summarizes the critical considerations for designing a resource provisioning framework for the DSPs.

Streaming tasks running on DSPs are more sensitive to the network-related parameters such as delay than batch tasks. The main reason for such a sensitivity comes from the task response time, computed as the time an event enters a DSP and gets processed [22]. Therefore, the link latency among the different compute resources in geo-distributed settings can highly impact the overall event execution time. The existing resource provisioning attempts [2, 12, 18, 29, 32, 40, 42] partially address either one of the key properties of the WAN links or one of the popular DSPs. For example, the work of [40] tried to provision the available resources of Apache Storm dynamically. An SDN-based resource provisioning system for the Apache Storm cluster in the multi-cloud setting is presented in [42] to monitor the status of the worker nodes in the cluster and assign the resources according to the workload. The SDN controller of the system monitors the worker node's physical resource consumption, such as CPU and memory, and schedules the tasks for better performance. However, the work [42] cannot be applied to Apache Spark and Apache Flink clusters. The work of [18] ensures the QoS of streaming tasks on Apache Storm without considering the WAN scenarios. The authors of [2] proposed an SDN-based architecture to meet the bandwidth requirements of streaming applications in a single datacenter. Therefore, the existing solutions are not general-purpose ones.

Current DPSs mostly support Apache Hadoop-Yarn [5], Apache Mesos [7], and Kubernetes [23] to manage the available resources. These systems work fine within a cluster in a local datacenter. They provide monitoring information such as physical resource usage (CPU, memory, disk, network), and the cluster manager in such systems is responsible for job scheduling among the workers. Nevertheless, these systems are unaware of WAN link status in a geo-distributed setting.

3 BACKGROUND AND MOTIVATIONS

In this section, we briefly explain the architecture of geo-distributed data analytic systems and the motivations of our work.

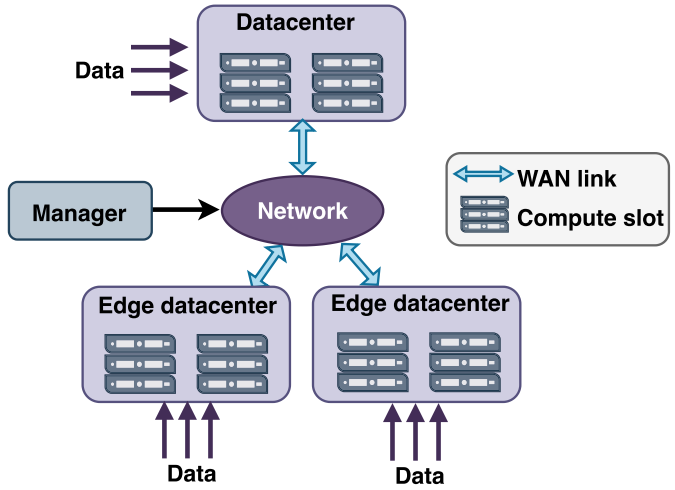


Fig. 1. An example of a geo-distributed stream analytics model with a datacenter and two Edge datacenters that are connected through WAN links. The manager deploys the streaming tasks on each datacenter to process the data streams.

3.1 Architecture of Geo-distributed Analytics Systems

The current DPSs follow a master-slave architecture in which a master runs the analytical tasks on a set of slaves or workers. These systems are designed to run diverse analytic tasks ranging from batch processing to machine learning and stream processing tasks on a local cluster in a rack on a datacenter. The master node schedules the user tasks to execute on the available workers of the analytic cluster. However, the applications of DPSs in the geo-distributed scenarios are also considered [44]. In a geographically distributed system, multiple datacenters, including edges and a central datacenter, are used to process the data. We aim to process data close to their origin for several reasons, like data privacy due to the privacy regulations such as the **General Data Protection Regulation (GDPR)**.

The datacenters in a geo-distributed setting are connected via WAN links with different properties like delay and bandwidth. One of the main bottlenecks for running the big data analytics tasks is the WAN links limitations, such as delay and the available bandwidth. Furthermore, the WAN links have different up-link and down-link bandwidths due to the heterogeneity of applications that share the links [36]. Therefore, the analytic tasks should be planned carefully to meet the user applications requirements and save the cost. Figure 1 shows an example of a geo-distributed analytics system with a datacenter and two edge datacenters. Each one has its computing powers, and the WAN links interconnect them.

The global manager in the current architecture for the geo-distributed systems converts each user's task into a **Directed Acyclic Graph (DAG)** of operators with several vertices and edges. Each DAG can be executed in a parallel and pipelined manner to provide low-latency execution and high-throughput. Each DPS provides a set of Application Programming Interfaces (APIs) to simplify the application developments for the developers. Due to the heterogeneity of resources in the different datacenters, minimizing the execution or response time for each query in the short window time is crucial. However, the current DSPs mostly rely on built-in or third-party components to select the workers. Otherwise, a customized version of that DPS is needed to take care of placement. This can be seen as data, tasks, or worker nodes in geo-distributed systems.

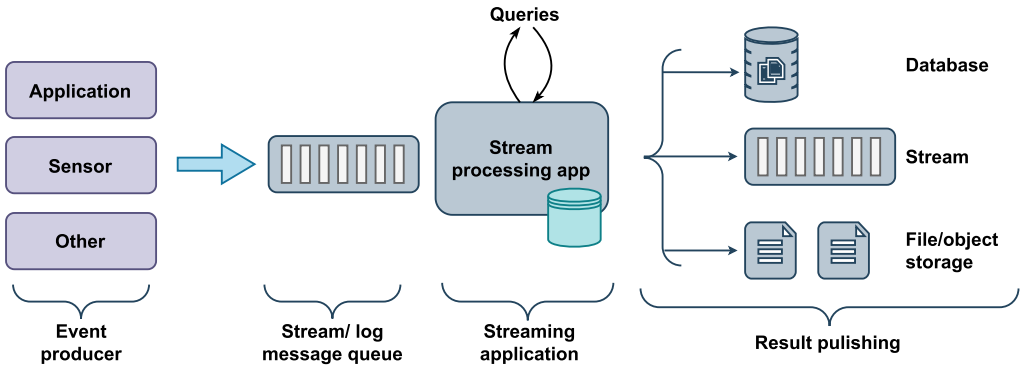


Fig. 2. The stream processing model in a DSP that takes the input streams from diverse sources, puts them into a queue, processes them, and writes the results into proper storage [41].

3.2 Stream Processing Model

In a stream processing model, the DSPs deal with processing continuous and unbounded data streams. Each DSP can run a set of streaming tasks on data streams, and each data item should be processed by a task when it becomes available. The goal is to extract insights from such data streams [19]. DSPs can run multiple parallel streaming tasks to improve the throughput of processing data items and reduce their processing latency. Since each streaming task can process a limited number of data items, we can adjust the available resources to improve the overall performance of the system.

Figure 2 shows the processing model of a streaming application on a DSP. The event producers, such as applications or sensors, generate the input data streams and feed them to a stream message queue system. The streaming application running on a DSP processes the incoming streams, and the users can submit queries to the ongoing data streams to get insights. The results of the processed events can be stored in several formats, such as a database or files [41].

3.3 Motivations

The tasks in the streaming scenarios have an indefinite lifespan and typically follow a producer/-consumer model. The results of running a streaming task in the geo-distributed network of worker nodes cross WAN links with transcontinental delays [33]. In such scenarios, the processing model demands higher inter-communications among the worker nodes [28]. Furthermore, the streaming tasks involve real-time decision-making due to the nature and dynamics of the input data stream, which adds extra complexity and intensity to them. Additionally, when the data is processed in several locations, and the results should be aggregated in a single location, we need to have proper worker node selection algorithms aware of underlying link properties. Therefore, leveraging improper WAN links among the worker nodes can impose a huge business impact on the enterprises. For instance, the report in [46] confirms dropping the customers of online businesses due to the delay in processing their requests. Thus, we should carefully consider the network-related parameters depending on the type of streaming applications in the geo-distributed settings.

The performance of DPSs can benefit from the recent advances in Software-Defined Networking (SDN) and programmable networks [45]. These technologies can be leveraged to efficiently handle the inter-communications among the worker nodes in a geo-distributed cluster and speed up packet processing in the core networks. They can also offer the administrators a wide range of flexibility to better benefit from the available resources. For instance, various datacenters in the

geo-distributed processing systems are connected through WAN links. The generated traffic by each of these datacenters possibly crosses several networks or **Internet eXchange Points (IXPs)** to reach the destination. Several WAN-related parameters need to be considered in this processing model; (1) The available bandwidth for inter-cluster communications can vary due to the amount of traffic and the availability of the WAN links. (2) The WAN delay affects the time the traffic reaches the destination. (3) The steering network traffic via transcontinental links can be costly. We have better control of the network by adopting the recent advances of SDN by employing the controller to handle the inter-datacenter traffic in the geo-distributed settings. Therefore, we propose a new architecture and system with different selection algorithms to improve the performance of the systems.

4 PROBLEM STATEMENT

We aim to minimize the execution latency of a streaming task in DSPs by considering the network-related parameters like the link delay using the power of the controller in the programmable networks. We assume the network topology graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ shows the set of nodes and $\mathcal{E}$ indicates the set of edges (links). Each node indicates a typical worker node in the DPS systems. Each link in $\mathcal{G}$ is associated with a delay. However, other network-related parameters, such as the available bandwidth and cost, can also be considered. Our objective is to select a set of worker nodes in the network to minimize the inter-node delays while using the maximum available bandwidth with minimum cost.

4.1 Delay

In classical stream processing scenarios, the streaming query is executed in a cluster of worker nodes placed in a rack on a data center. In such clusters, the internode delay is in order of a few milliseconds. However, the WAN links come with a longer delay that has a determinant impact on the geo-distributed systems running the delay-sensitive applications. Consider a scenario in which every worker node needs to process a small number of streaming data in situ, and the results should be aggregated in a single location [24]. In such scenarios, the inter-node properties of the links dictate the overall execution time of the task for each event. However, the system aims to process the incoming data in order of a few milliseconds and undertake a suitable action. Therefore, we aim at selecting a set of worker nodes from graph $\mathcal{G}$ by minimizing the maximum internode delays. Mathematically,

$$D_{\mathcal{G}'} = \sum_{\forall (u,v) \in \mathcal{E}'} D_u^v, \tag{1}$$

where $D_u^v > 0$ is the maximum link delay from node u to v, and $D_{\mathcal{G}'}$ is the sum of link delays in graph $\mathcal{G}'$– which is a sub-graph of $\mathcal{G}$. Our goal is to minimize the total delay among the nodes in graph $\mathcal{G}'$.

4.2 Cost

We assume that C_u^v is the data transmission cost from node u to v. We can compute the total transmission cost C_c as follows.

$$C_c = \sum_{(u,v) \in \mathcal{E}'} C_u^v \tag{2}$$

We just consider the cost of steering the traffic on the links. However, extra cost like the cost of executing the task on the node can be also considered here. The link cost to steer the traffic stays unchanged in a network deployed in a continent [3, 3, 15]. Therefore, the link cost is the same for most networks.

4.3 Bandwidth

We assume that B_{uv} is the capacity of the link from node u and v. If B^f_{uv} is the amount of network traffic from u to v using the execution assignment strategy f then we can compute the total generated traffic on link $(u, v) \in \mathcal{E}$ as follows.

$$B^v_u = \sum_{\forall f} B^f_{uv}, \tag{3}$$

Assume that we have three applications steering traffic from link $(u, v) \in \mathcal{E}$, namely, $f1$, $f2$, and $f3$ that share the same link. The total traffic crossing this link will be the sum of the traffic crossing from these three paths, called T_uv. Furthermore, the available bandwidth A_{BW} can be computed as follows.

$$A_{BW} = B_{uv} - T_{uv} \tag{4}$$

4.4 Overall Formulation

We model the problem as the placement problem using the idea of SDN. The main aim is to jointly minimize the query execution latency and the cost while maximizing bandwidth utilization. This problem is formulated as follows.

$$\min_D \min_T \max_B \left[\alpha \cdot \sum_{\forall (u,v) \in \mathcal{E}} D^v_u + \beta \cdot \sum_{\forall (u,v) \in \mathcal{E}} T^v_u + \gamma \cdot \sum_{\forall (u,v) \in \mathcal{E}} B^v_u \right], \tag{5}$$

where $\alpha, \beta, \gamma \geq 0$ and $\alpha+\beta+\gamma = 1$. The α determines the impact of delay, β specifies the importance of cost, and γ states the importance of the available bandwidth.

5 THE SYSTEM

This section describes our architecture for distributed streaming processing systems in a geo-distributed scenario. First, we explain the structure of our architecture. Then, we state the detail of our algorithms for worker node selection using the WAN links delay and other network-related parameters like the available bandwidth and cost.

5.1 Architecture

Our proposed architecture is a general-purpose one that can be applied to all DPSs. The architecture has four different components, namely, *user*, *SDN controller*, *master node*, *datacenters*. Figure 3 depicts the components and their interactions.

We explain how different components interact with each other in our architecture. The SDN controller component receives the query execution request from the application developer or user, asking for the desired number of worker nodes or task slots for the execution. Therefore, we assume that the required number of worker nodes is given as input. The SDN controller monitors the network to find the best set of worker nodes satisfying the goals in Equation (5). To do this, it collects the network topology information by sending a set of probe packets. The network topology consists of the datacenters' network connected through WAN links. We assume each node in the network graph plays the role of a datacenter. Each datacenter has at least one available worker node to execute a task. Each worker node in the architecture sends a proper response to the received probe packets. After collecting all the information, the controller has a global view of the network. It knows how many worker nodes are available in the cluster and the different characteristics of each link among the nodes in the cluster.

The SDN controller runs the selection algorithms to choose a subset of nodes according to the user input. We propose two algorithms for the worker node selection depending on the

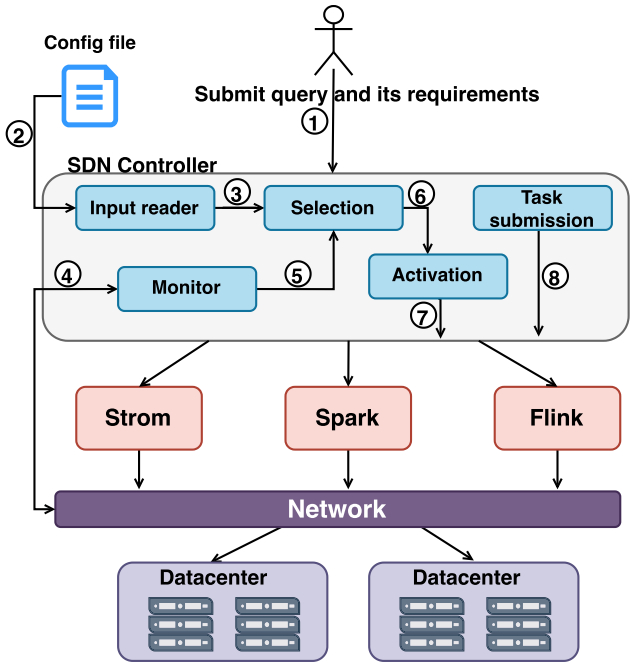


Fig. 3. The proposed SDN-based architecture for the geo-distributed analytics systems integrated with Storm, Spark, and Flink. The DSP cluster is deployed over the Edge datacenter, and the SDN controller is responsible for network monitoring, selecting the worker nodes, and managing the available resources.

WAN-relevant parameters. More detail on the algorithms comes in Sections 5.2.1 and 5.2.2. Then, the controller starts the daemons of the worker nodes of the DPS for query execution. Each DPS has a daemon to start a worker node that can be leveraged later by the master node of the DPSs for task execution. However, due to some necessary interactions among the master and worker nodes in DPSs, such as Apache Flink, we start the master daemon before starting the worker ones. At this moment, the worker nodes are ready to receive the tasks from the master node. This procedure repeats when the user submits a new query. Now, the controller submits the job using the APIs provided by each DPS to the master node.

The internal architecture of the controller. The SDN controller can periodically monitor the network for the available resources and adapts the cluster using the obtained information. For example, it can query the worker nodes for further details like the CPU, RAM, and other available resources. The SDN controller has the following components: `InputReader`, *Monitor*, *Selection*, *Activation*, and *TaskSubmission*. We now explain how these components interact with each other in SAFA.

The controller receives the number of desired worker nodes and the selection algorithm information, i.e., one of the algorithms in Sections 5.2.1 and 5.2.2, from the application developer using the `InputReader` component. This is done using the configuration file of our framework.

Network monitoring. The controller needs to monitor the cluster network for the network-relevant parameters and obtain the required information to apply the selection algorithm. This step is carried out using the *Monitor* component. We rely on the available tools such as *ping* to gather the delay information of the WAN links. Using ping is a design choice; otherwise, we can generate probe packets to obtain the delay of the links. There are tools to monitor the links for the

```
{"config": {
 "Spark": [
  {"SPARK_MASTER_HOST": "<IP of master>"},
  {"Spark": "path/to/Spark"},
 ],
 "Storm": [
  {"nimbus.host": "<IP of master>"},
  {"storm.zookeeper.servers": "<IP of ZooKeeper>"},
  {"Storm": "path/to/Storm"},
 ],
 "Flink": [
  {"jobmanager.rpc.address": "<IP of master>"},
  {"Flink": "path/to/Flink"},
 ],
  "Selection_algorithm": ["DCM","ECM","Default"],
  "Parameter_weight": [α,β,γ]
 }
}
```

Fig. 4. A configuration example file of our architecture for Apache Spark, Apache Storm, and Apache Flink.

available bandwidth among each pair of worker nodes in the network, such as *Iperf*. We can also utilize other available bandwidth measurement tools/techniques to estimate the remaining capacity of the links to send the traffic. Furthermore, we can use the telemetry information provided by **In-band Network Telemetry (INT)** [39] of P4 to get such information.

Note that the combination of *ping* and *Iperf* cannot provide the advantages of using the SDN controller in SAFA for the following reasons. (1) By combining *ping* and *Iperf*, the network operator of geo-distributed streaming analytics can get some insights from the underlying network conditions from a part of the network without being able to get information on all links. (2) The master node cannot also dynamically change the forwarding rules among the devices in the network to adapt the network to the new changes.

Selection of worker nodes. The controller leverages the *Selection* component to choose the worker nodes based on the user's needs. This happens by applying the selection algorithm to the information obtained by the *Monitor* component. The output of this component is a set of worker nodes with their corresponding IP addresses. The IP addresses of the worker nodes are used in the next step to activate the daemons of each worker node.

Starting the cluster. After selecting the worker nodes, the controller needs to run the corresponding daemon of each DPS to activate the worker node of the system. This is done using the *Activation* component. This component needs the directory of the daemon in the worker node system in which the daemon resides. For example, Apache Flink has a daemon called *taskManager* to start a worker node in a cluster. The corresponding daemon and its directory come from the same configuration file provided by the developer. The controller also monitors the status of the worker nodes and starts a new one according to the output of the selection algorithms in Section 5.2 if one fails.

Task submission. At this point, the cluster is ready for task submission. This is performed using the *TaskSubmission* component. The controller leverages the provided API by each DPS to submit the task using the master node. Figure 4 shows an example configuration file. This example configuration file is for the three widely used stream processing systems, namely, Apache Spark [8], Apache Storm [9], and Apache Flink [4]. To start the master or worker nodes in each DPS, the SDN controller requires the directory of DPS in the target VM to start the corresponding daemon of master or worker nodes. The controller reads this information from the configuration file by checking the value of the corresponding key in the JSON file. The controller also needs the IP address of

the ZooKeeper VM to start Apache Storm because it relies on ZooKeeper for state management. However, the controller writes the IP addresses of master nodes in each DPS after applying the selection algorithms. This avoids the master node selection for the upcoming tasks.

5.2 Selection Algorithms

This section describes our selection algorithms to select a subset of worker nodes. As the internode delay plays a major role in the delay-sensitive streaming applications, we first propose two delay-based selection algorithms, namely, *DCM* and *ECM*. Then, we assess the impact of the bandwidth and cost to find the tradeoff among the network-relevant parameters in Section 7.

5.2.1 DCM. The delay-based ranking algorithm *DCM* has two phases, namely, *ranking* and *node selection*. In the former phase, the nodes in the network topology are ranked based on the number of neighbor nodes and their corresponding delay. The latter phase selects the required number of nodes t for the cluster. Algorithm 1 shows the pseudo-code of this algorithm.

ALGORITHM 1: DCM algorithm

input : Network graph $\mathcal{G}$, links delay, number of worker nodes
output : a set of nodes $\mathcal{W}$

```
1  W = ∅;                                          /* The set of worker nodes */
2  H = ∅;                                          /* The set of unvisited nodes */
3  u = findFirstNodeDCM(G);
4  W = W ⋃ u;
5  while |W| < t do
6      for v ∈ u.Neighbors do
7          H = H ⋃ v;
8      end
9      tmp = null;
10     t_d = ∞;
11     for v ∈ H \ u do
12         if t_d < d_u^v then
               ; /* d_u^v is the delay from u to v                          */
13             t_d = d_u^v;
14             tmp = v;
15         end
16         H = H \ u;
17     end
18     u = tmp;
19     W = W ⋃ u;
20 end
```

Ranking. In the ranking phase, the algorithm computes the ranking score of all nodes. To do this, it picks a node from the network graph $\mathcal{G}$ and checks all of its neighbors for their corresponding delay. We use Equation (6) to compute the rank of each node.

$$R_u^{DCM} = \frac{\sum_{v=1}^{\mathcal{N}} D_u^v}{|\mathcal{N}|}, \quad \forall v \in \mathcal{N}_u, \tag{6}$$

where $\mathcal{N}$ is the list neighbors of node u and D_u^v is the link delay from node u to v. After computing the score of all nodes, the algorithm returns the node with the minimum score as the location of the first node to place the master node in the network. We co-locate the first worker node with the master in the cluster. However, this co-location choice is not restricted, and it can be extended to use another location for the first node. Algorithm 2 shows the pseudo-code of this algorithm.

ALGORITHM 2: Rank nodes in DCM

```
input  : Network graph 𝒢, links delay
output : A node 𝒩_best
1  Function findFirstNodeDCM(𝒢):
2      𝒟_min = ∞;
3      𝒩_best = null;
4      for each u ∈ 𝒢 do
5          𝒩 = 𝒢.getNeighbors(u);
6          sum = 0;
7          for v ∈ 𝒩 do
8              sum + = v.getLatency();
9          end
10         w = avg(sum);
11         if w < 𝒟_min then
12             𝒟_min = w;
13             𝒩_best = u;
14         end
15     end
16     return 𝒩_best;
17 End Function
```

Node selection. In the node selection phase, the algorithm adds further nodes to the list of chosen nodes. We use $\mathcal{W}$ as the list of selected nodes and $\mathcal{H}$ as the list of candidate nodes. In the beginning, the algorithm calls the `findFirstNodeDCM` function to select the first node and adds it to $\mathcal{W}$. Then, it adds the neighbors of node u to the $\mathcal{H}$ list. The algorithm continuously updates $\mathcal{H}$ by adding a new node to $\mathcal{W}$. To select the second worker node, the algorithm looks for the shortest delay from node u to one of the nodes in $\mathcal{H}$ called node v. Then, it adds node v to $\mathcal{W}$. At this point, we have the neighbors of nodes u and v in $\mathcal{H}$. The node selection procedure continues until the suitable number of worker nodes t is chosen (see lines 5–20) of Algorithm 1.

5.2.2 ECM. ECM prefers the worker nodes based on the number of egress ports. The motivation to propose ECM is that the worker nodes with more neighbors have better connectivity in the network graph. This connectivity feature can lead to a few WAN links among the chosen nodes. It also leads to the selection goal in Equation (5). The ECM algorithm has also two phases, namely, `ranking` and `node selection`. The nodes will be ranked for the first node selection in the ranking phase, and adding nodes to the cluster is carried out in the node selection phase. ECM runs this phase just once for the first node selection. We explain each phase in more detail.

Ranking. We rank all the nodes of the graph by giving the highest weight to the nodes with the highest number of neighbors and their average delays. Equation (7) computes the rank of each

node as follows.

$$R_u^{ECM} = |\mathcal{N}_u| + \frac{|\mathcal{N}_u|}{\frac{\sum_{\forall v \in \mathcal{N}_u} D_u^v}{|\mathcal{N}_u|}}, \tag{7}$$

where $|\mathcal{N}_u|$ is the number of neighbors of node u. Algorithm 3 shows the pseudo code of first node selection in ECM.

ALGORITHM 3: Rank nodes in ECM

input : Network graph $\mathcal{G}$, links delay
output: A node $\mathcal{N}_{best}$

```
1  Function findFirstNodeECM(G):
2      R_best = 0 ;
3      N_best = null ;
4      for each u ∈ G do
5          N = G.getNeighbors(u);
6          sum = 0;
7          for v ∈ N do
8              sum += v.getLatency();
9          end
10         R_u = |N| + |N| / sum;
11         if R_u > R_best then
12             R_best = R_u;
13             N_best = u;
14         end
15     end
16     return N_best;
17 End Function
```

Node selection. This phase of the algorithm is similar to DCM. However, the remaining part of the algorithm is similar to Algorithm 1. We call this version of algorithm ECM in Section 7.

5.3 Running Example

We now provide a running example for the selection algorithms of our architecture. Figure 5 presents the Abilene network with its internode **Round Trip Time (RTT)** delays in milliseconds. The cost of steering traffic among the nodes in this network is 0.02$ per GB of data transfer [3, 15]. This graph has 11 nodes, and the goal is to place four worker nodes for the cluster in this network. We first explain the DCM algorithm. Then, we focus on node selection in the ECM algorithm.

5.3.1 DCM Example. **First node selection.** To place the first node in the DCM algorithm, we need to compute the rank of all nodes in the graph. This method ranks the nodes based on the WAN link delays among the neighbors of each node. We use Equation (6) for ranking, and Table 1 shows the rank of each node after applying this equation. According to DCM, the location of the first node is *Was*.

Node selection. After selecting *Was* as the first node, DCM adds this node to $\mathcal{W}$ as the list of chosen worker nodes. DCM also adds the neighbors of this node to $\mathcal{H}$ as the list of candidate nodes, i.e., *NY* and *Atl*, for the selection in the next step.

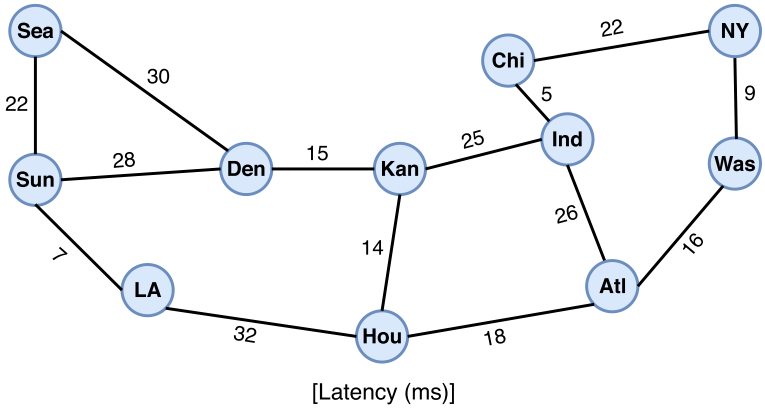


Fig. 5. The Abilene network topology with the inter-node delays in milliseconds taken from [34] to place four worker nodes using the selection algorithms.

Table 1. The Average Latency Values for Each Node for the Selection First Node

Node	$\lvert\mathcal{N}\rvert$	R_u^{DCM}	R_u^{ECM}
NY	2	15.5	2.129
Chi	2	13.5	2.148
Was	2	**12.5**	2.160
Sea	2	26.0	2.077
Sun	3	19.0	3.157
LA	2	19.5	2.102
Den	3	24.3	3.123
Kan	3	18.0	**3.166**
Hou	3	21.3	3.140
Atl	3	20.0	3.150
Ind	3	18.6	3.160

Table 2. The Values of $\mathcal{W}$ and $\mathcal{H}$ while Applying the DCM and ECM Algorithms

DCM			**ECM**		
Worker node	$\mathcal{W}$	$\mathcal{H}$	**Worker node**	$\mathcal{W}$	$\mathcal{H}$
Was	{Was}	{NY, Atl}	Kan	{Kan}	{Den, Hou, Ind}
NY	{Was, NY}	{Atl, Chi}	Hou	{Kan, Hou}	{Den, Ind, LA, Atl}
Atl	{Was, NY, Atl}	{Chi, Ind, Hou}	Den	{Kan, Hou, Den}	{Ind, LA, Atl, SF, Sea}
Hou	{Was, NY, Atl, Hou}	{Chi, Ind, Kan, LA}	Atl	{Kan, Hou, Den, Atl}	{Ind, LA, SF, Sea, Was}

DCM selects *NY* as the place of the second node and puts it into $\mathcal{W}$ and removes it from $\mathcal{H}$. Now, the algorithm adds the neighbors of *NY* to $\mathcal{H}$. This procedure continues until four nodes are chosen from this graph as the desired number of worker nodes. Table 2 shows the corresponding values of $\mathcal{W}$ and $\mathcal{H}$ lists. Each row of this table shows how DCM updates the two lists. The places of the final four nodes are *Was*, *NY*, *Atl*, and *Hou*.

5.3.2 ECM Example. **First node selection.** In ECM, the number of neighbors has the most significant impact on the first node selection. We rank all nodes in the graph using Equation (7),

and the fourth column of Table 1 shows the computed rank of each node. ECM selects the node in Kan as the first chosen worker node since it has the highest rank.

Node selection. After selecting Kan, ECM adds this node to $\mathcal{W}$ and the neighbors of this node to $\mathcal{H}$. This procedure continues until four nodes in the graph have been selected. Table 2 shows the values of $\mathcal{W}$ and $\mathcal{H}$ in each step of node selection of ECM. The final four nodes are *Kan*, *Hou*, *Den*, and *Atl*.

6 PROOF-OF-CONCEPT

We implement the node selection algorithms of SAFA[1] in Python and the networking part in P4 [10], which is a data plane programming language offering wide flexibility in programming the forwarding behavior of the network devices. However, this choice is not restrictive to P4-based programmable networks, and the architecture can work on any OpenFlow-based networks. In more detail, we report the proof-of-concept architecture implementation in P4 and briefly explain the OpenFlow-based implementation using the Ryu controller.

Rule generation. We need to steer the network traffic among the selected worker nodes to be able to execute the submitted tasks. This requires forwarding rules for all the involved switches or routers among the workers in the network. Therefore, we generate IPv4 forwarding rules while selecting the nodes. The source and destination IP addresses of the node, along with the corresponding egress port, are used to steer the network traffic. We claim that having the egress port number is mandatory in generating the forwarding rules because there might be multiple paths among the pair of source and destination IP addresses. In such a scenario, we aim to steer the traffic of workers through the links that are selected by our algorithms. Otherwise, the traffic can cross using different links with higher internode delay.

We also generate the forwarding rules to handle the connectivity between the workers and the switches connecting them to other workers. Examples of such forwarding rules are **Address Resolution Protocols (ARP)**. Furthermore, the master and the controller nodes in each DPS need full access to the workers to execute the tasks. Therefore, we rely on the recommendations proposed by the DPSs and use the **Secure Shell (SSH)** protocol for this purpose.

Rule installation. We use the P4 runtime to install the forwarding rules. It reads the required configuration information from corresponding files and installs rules on top of P4 switches. Additionally, we use P4 runtime also to monitor the network status periodically. For example, consider a scenario where there is a link failure in the network, and the traffic of this link is forwarded through a delay link. This will add an extra delay to the task execution. To alleviate this, the P4 runtime periodically sends probe packets to the network and updates the forwarding rules accordingly. However, this is not the only advantage of using P4 because we can mark the streaming packets in the switch with high priority to be forwarded without being queued on the intermediate P4 switches along the path while there is congestion in the network.

Updating rules. Our controller monitors the network by issuing the probe packets to keep the chosen links among different switches updated. For example, if the controller finds a connection link with shorter latency, it can update the forwarding rules to steer the inter-node traffic via the new links.

Memory cost. We need 32 bits to store each source and destination IP addresses. Each egress port number requires 9 bits. Therefore, we need 73 bits to install each forwarding rule. The number of forwarding rules for each cluster depends on the number of nodes. Consider a cluster with t number of workers; we need to generate $(t \times (t-1))/2$ rules to steer the traffic of the cluster among the workers.

[1] https://github.com/mostafaei/SAFA.

Table 3. Custom Graphs Properties

Graph	Nodes	Links	Link delay range [ms]
Topo#1	20	35	[1, 25]
Topo#2	30	65	[26, 75]
Topo#3	50	140	[76, 125]

Implementation on OpenFlow-based networks. We also implemented SAFA on an OpenFlow-based network using the Ryu controller. The controller reads the criteria to place the worker nodes from the configuration file and activate them via the APIs provided by each DSP to run the streaming tasks. Our controller periodically monitors the links and updates the forwarding rules of the switches to detour the traffic if a short latency link is found. Similar operations happen when the criteria for the placement change to the available bandwidth or cost. The controller sends the flow modification messages to the corresponding switches affected by the selection to update the forwarding rules.

7 EVALUATIONS

In this section, we first evaluate the effectiveness of our selection algorithms on a set of custom and real network topologies. Then, we run real-world experiments using the extended version of YSB [13] in [14] on a set of chosen worker nodes from our custom topologies to assess the performance of the algorithms and improvements gained using our architecture.

We generate three random connected graphs, namely *Topo*#1, *Topo*#2, and *Topo*#3 using the attributes shown in Table 3. The reason to choose such delay values for each link is to model the real networks' delay [34] by checking the real-world datacenter locations [16, 30]. Additionally, we check the performance of our node selection algorithms on all network topologies obtained from TopologyZoo in terms of average internode delays. We obtain the link delay information of each network in TopologyZoo using the coordination information and report the results for those having such properties.

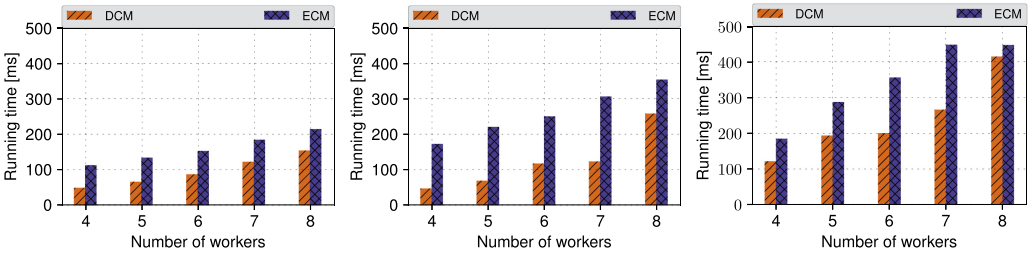
Systems under tests. We run experiments on three widely used DPSs, namely, Apache Flink [4], Apache Storm [9], and Apache Spark [9]. We use the extended [14] version of YSB [13] to run a streaming query on these systems. This benchmark has an advertisement query that generates data and puts them into Kafka consumers. The worker nodes consume the generated data and write the output into the Redis database. The benchmark measures the processing time of events received in a specific time window. We report the 99-percentile latency of the processed events generated by the benchmark in our experiments. More detail can be found in [13].

7.1 Running Time

We now measure the running time of DCM and ECM algorithms in selecting the required number of worker nodes. Note that the running time of the Default approach comes with the minimum latency since it chooses the worker nodes from the available list. Therefore, we exclude its running time in this section. Figure 6(a), 6(b), and 6(c) show that the running time of both algorithms increases by looking for more worker nodes in the network graph. However, ECM has the lowest running time compared with DCM since it checks less number of nodes in the network graph.

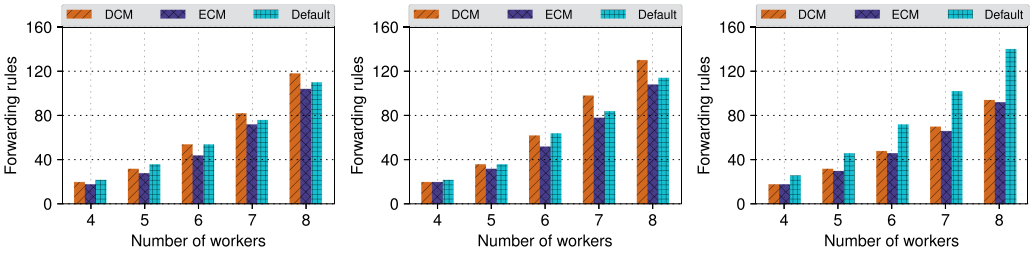
7.2 Forwarding Rules

We report the number of forwarding rules needed to steer the network traffic among the chosen worker nodes in each topology. These rules are control plane rules installed on the chosen switches in the network. Figure 7(a), 7(b), and 7(c) show that ECM installs fewer forwarding rules because it



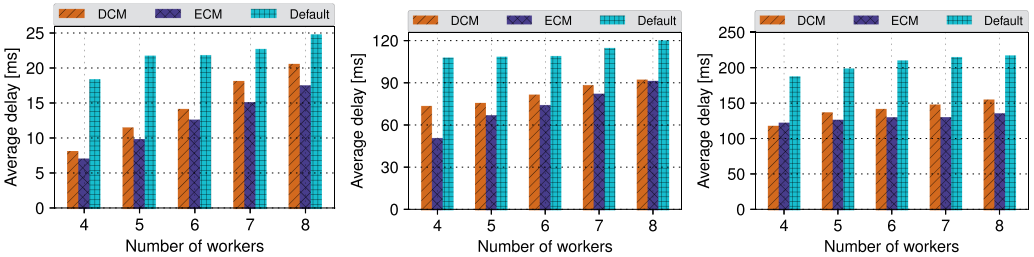
(a) $Topo$#1 (b) $Topo$#2 (c) $Topo$#3

Fig. 6. The running time of DCM and ECM on the custom topologies.



(a) $Topo$#1 (b) $Topo$#2 (c) $Topo$#3

Fig. 7. The total number of forwarding rules needed to steer the traffic among the chosen worker nodes in each topology.



(a) $Topo$#1 (b) $Topo$#2 (c) $Topo$#3

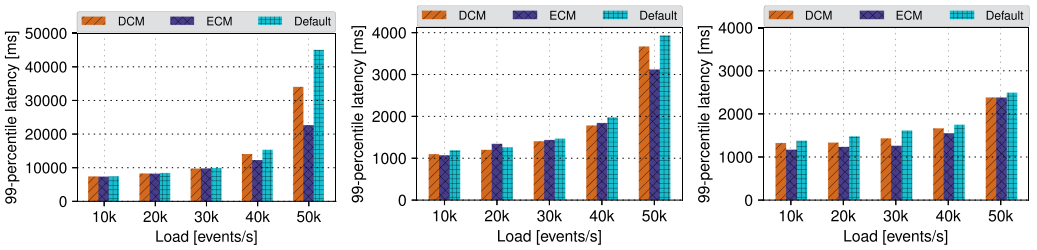
Fig. 8. The average delay among the chosen worker nodes on $Topo$#1, $Topo$#2, and $Topo$#3.

selects the switch with the highest number of egress ports as the first node. This decision reduces the number of forwarding rules and saves the switches' memory for further usage.

7.3 Topology-aware Results

We first report the average delay among the worker nodes on the three topologies. The current systems, without leveraging third-party systems like Apache Mesos or Hadoop-Yarn, select the worker nodes based on the available physical resources on the worker nodes. Therefore, they pick the worker nodes in an ordered fashion if they have the same features. We call this mechanism in our evaluations the "Default" algorithm.

Average delay among worker nodes. Figure 8 depicts the average delay among the selected nodes by the DCM, ECM, and Default algorithms. We vary the number of worker nodes in the range of 4 to 8 nodes. Figure 8(a), 8(b), and 8(c) illustrate that the average delay among the worker



(a) Spark (b) Storm (c) Flink

Fig. 9. The 99-percentile execution latency of Spark, Storm, and Flink on custom *Topo*#1 when varying the input data rate load.

nodes slightly increases by increasing the selection of more worker nodes. ECM can select worker nodes with 88%, 58%, and 58% less delay than the Default approach in three topologies.

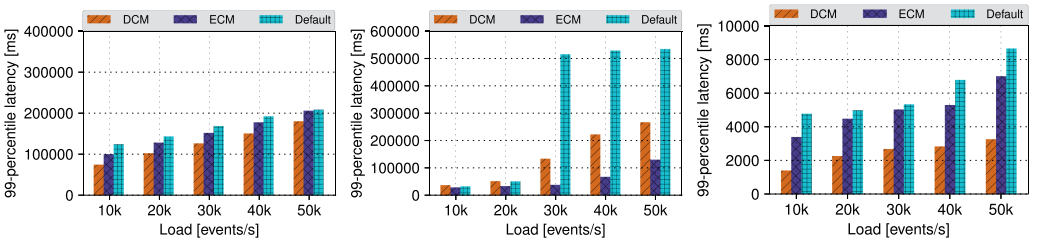
7.4 Evaluation Testbed

To evaluate the performance of our architecture on the popular DSPs in a programmable network, we need to have enough P4 switches to create the network. Since we lack many P4 switches to build the cluster, we use the Mininet [31] emulator to build the topologies. Then, we apply the selection algorithms to the network to obtain the link properties of the chosen subset of worker nodes. At this step, we have a list of nodes and their corresponding link information. Now, we use our dedicated cluster to run the YSB task on the chosen nodes.

Our cluster has 11 VMs, each with 16 CPU cores with 8 GB of RAM running Debian 10. We dedicate one VM for the master node and 8 VMs for the worker nodes. We also run Kafka and Redis on separated VMs. The Kafka [6] and Redis [38] VMs are used for data generation, and we connect these two VMs to the worker nodes without applying any internode delays among them. We can use our testbed to simulate real-world application scenarios. We measure the impact of the placement algorithms in our testbed using YSB [13] for different topologies. We use the *tc* tool to add artificial delays to the links among the nodes in the graph. We use similar configurations for all three systems in this benchmark. The obtained results are for 10 minutes of running the streaming task on our cluster.

7.4.1 Topo#1 Results. We assess the impact of the worker node placement on the execution latency of the streaming query on the small custom topology for Spark, Storm, and Flink by placing 8 worker nodes. To do so, we vary the input data rate from 10k to 50k events/second and measure the execution latency of the three systems. We used 10 seconds of batch time for Spark experiments while setting Storm to no acknowledgment for the events. Figures 9(a), 9(b), and 9(c) show the 99-percentile execution latency of three systems. We observe that increasing the system load increases the execution latency of the DPSs. Flink has a similar performance regardless of placement algorithms, while Spark has very high execution latency. The ECM has a better performance by increasing the load in Storm.

7.4.2 Topo#2 Results. We also did the same measurements on *Topo*#2 and found that Spark and Storm have very high execution latency while Flink can still process the incoming data reasonably. The main difference between the medium and small topologies is the inter-node delays among the nodes. Figures 10(a), 10(b), and 10(c) show the DCM algorithm improves 1.64×, 4.00×, and 2.69× the execution latency Spark, Storm, and Flink compared with the Default method, respectively. Furthermore, ECM improves up to 1.34×, 13.47×, and 1.27× the percentile execution latency of all systems compared to the Default mechanism.



(a) Spark (b) Storm (c) Flink

Fig. 10. The 99-percentile execution latency of Spark, Storm, and Flink on custom *Topo*#2 when varying the input data rate load.

Table 4. The Trade-off Among the Different Network-related Metrics on Each Custom Topology

	Topo#1			*Topo*#2			*Topo*#3		
Prioritized metric	min(**BW**)	**Latency**	**∑Cost**	min(**BW**)	**Latency**	**∑Cost**	min(**BW**)	**Latency**	**∑Cost**
Bandwidth	7.7Mbps	25.71 ms	16.64	8.3Mbps	92.25 ms	17.42	7.7Mbps	145.26 ms	7.29
Latency	7.4Mbps	17.59 ms	11.36	7.5Mbps	86.63 ms	21.57	7.2Mbps	135.85 ms	7.21
Cost	7.4Mbps	20.93 ms	9.11	7.4Mbps	115.26 ms	9.75	7.6Mbps	206.07 ms	2.89

7.5 Understanding Tradeoff

Tradeoff among the parameters. This section reports the tradeoff of assigning the higher priority to bandwidth, link latency, and cost by changing the weight of each one on the custom topologies. Herein, we set the highest weight to one parameter and keep the other parameters fixed in both algorithms, i.e., DCM and ECM. The users can prioritize the selection of worker nodes for each submitted task in SAFA. We set the cost in the custom topologies using the data in [3, 15], which randomly varies between 0.02 to 0.25 $ per GB of transferring data. We also randomly set the available bandwidth according to the Akamai report in [1] in the range of 7 to 14 Mbps. In this approach, we select the highest rank among the neighbors of a node in selecting a worker node in each step using the α, β, and γ coefficients in Equation (5) until the required number of worker nodes is chosen.

Table 4 summarize the results of this experiment for *Topo*#1, *Topo*#2, and *Topo*#3 topologies. The values of Cost are USD per GB. In this table, we also report the minimum available bandwidth ($\min_{BW}$), the average of links latency ($\overline{Latency}$), and sum the Cost ($\sum$Cost) among the chosen worker nodes. The values in each row of Table 4 show that our algorithm can properly select the worker nodes according to the user needs. We highlight the preferred values for each prioritized metric in red. For example, if the bandwidth has the highest priority for the user in the *Topo*#1, the framework selects worker nodes with the minimum available bandwidth of 7.7 Mbps. The available bandwidth is 7.4 Mbps when the latency or Cost has the highest priority.

Tradeoff among the Default vs DCM and ECM. We now show the tradeoff among different selection algorithms on the performance metrics of the system. The Default approach is fast in selecting the worker nodes since it selects them from the available list. Also, the Default approach needs more forwarding rules to steer the traffic among the chosen worker nodes and has a higher average delay than ECM and DCM algorithms. ECM algorithm performs better in selecting the worker nodes with lower inter-node latency, but its running time is higher than DCM and the Default methods. The ECM algorithm also installs a slightly less number of forwarding rules compared with the other approaches. Finally, DCM achieves a better execution latency on Apache Spark and Apache Flink in most scenarios, while ECM performs better on Apache Storm.

7.6 Delay Fluctuations

The WAN links delay can over time, and the recent study in [37] showed that the delay variations could be in the order of a few milliseconds. We did experiments to understand the impact of the delay variation on the performance of all systems and reported no specific change execution latency of Spark, Storm, and Flink [33].

8 CONCLUSION

This paper presents a WAN topology-aware framework for geo-distributed stream processing systems using programmable networks. The system monitors the network topology for the WAN delays among the nodes and selects the subset of worker nodes with minimum delays. Our proposed framework can be executed on any distributed processing system that offers the possibility of starting the cluster nodes separately. The framework does not need any modifications to the current DPSs to manage their cluster. The proposed placement algorithms of SAFA show that they can place the workers with fewer internode delays among the worker nodes of all the current DPSs. Our results show that the framework can improve at least 1.64× the execution latency of the current Apache-based DPSs. Therefore, it can be leveraged for delay-sensitive streaming applications. We plan to investigate the impact of our placement algorithms on more application scenarios, possibly by considering other system-related parameters such as CPU and RAM. We also intend to extend our architecture to measure the incoming data rate to the system and decide on the suitable number of worker nodes to process them.

REFERENCES

[1] Akamai's [state of the internet]: Q1 2017 report. 2017. https://www.bit.ly/3jPSKEP.
[2] Walid Aljoby, Xin Wang, Tom Z. J. Fu, and Richard T. B. Ma. 2019. On SDN-enabled online and dynamic bandwidth allocation for stream analytics. *IEEE Journal on Selected Areas in Communications* 37, 8 (2019), 1688–1702. https://doi.org/10.1109/JSAC.2019.2927062
[3] Amazon EC2 On-Demand Pricing. 2020. https://aws.amazon.com/ec2/pricing/on-demand/.
[4] Apache Flink. 2021. https://flink.apache.org/.
[5] Apache Hadoop YARN. 2021. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.
[6] Apache Kafka. 2021. https://kafka.apache.org/.
[7] Apache Mesos. 2021. https://mesos.apache.org/.
[8] Apache Spark. 2021. https://spark.apache.org/.
[9] Apache Storm. 2021. https://storm.apache.org/.
[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[11] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2017. Optimal operator replication and placement for distributed stream processing systems. *SIGMETRICS Perform. Eval. Rev.* 44, 4 (May 2017), 11–22. https://doi.org/10.1145/3092819.3092823
[12] Javier Cervino, Evangelia Kalyvianaki, Joaquin Salvachua, and Peter Pietzuch. 2012. Adaptive provisioning of stream processing systems in the cloud. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. 295–301. https://doi.org/10.1109/ICDEW.2012.40
[13] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138
[14] Extending the Yahoo Streaming Benchmarks. 2020. https://github.com/dataArtisans/yahoo-streaming-benchmark.
[15] Google cloud: pricing. 2020. https://cloud.google.com/pubsub/pricing.
[16] Google Datacenters. 2021. https://about.google/locations/.
[17] B. Heintz, A. Chandra, and R. K. Sitaraman. 2017. Optimizing timeliness and cost in geo-distributed streaming analytics. *IEEE Transactions on Cloud Computing* (2017), 1–1.

[18] M. Reza Hoseiny Farahabady, Hamid R. Dehghani Samani, Yidan Wang, Albert Y. Zomaya, and Zahir Tari. 2016. A QoS-aware controller for Apache Storm. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. 334–342. https://doi.org/10.1109/NCA.2016.7778638

[19] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. 2019. A survey of distributed data stream processing frameworks. *IEEE Access* 7 (2019), 154300–154316. https://doi.org/10.1109/ACCESS.2019.2946884

[20] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2018. Rethinking adaptability in wide-area stream processing systems. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*.

[21] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2020. WASP: Wide-area adaptive stream processing. In *Proceedings of the 21st International Middleware Conference (Middleware'20)*. 221–235. https://doi.org/10.1145/3423211.3425668

[22] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. 2018. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[23] Kubernetes. 2021. https://kubernetes.io/.

[24] Dhruv Kumar, Sohaib Ahmad, Abhishek Chandra, and Ramesh K. Sitaraman. 2021. AggNet: Cost-aware aggregation networks for geo-distributed streaming analytics. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. 297–311. https://doi.org/10.1145/3453142.3491276

[25] Dhruv Kumar, Jian Li, Abhishek Chandra, and Ramesh Sitaraman. 2019. A TTL-based approach for data aggregation in geo-distributed streaming analytics. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 2, Article 29 (June 2019), 27 pages.

[26] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2020. Sol: Fast distributed computation over slow networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 273–288.

[27] W. Li, D. Niu, Y. Liu, S. Liu, and B. Li. 2019. Wide-area spark streaming: Automated routing and batch sizing. *IEEE Transactions on Parallel and Distributed Systems* 30, 6 (June 2019), 1434–1448.

[28] Xunyun Liu and Rajkumar Buyya. 2020. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Comput. Surv.* 53, 3, Article 50 (May 2020), 41 pages.

[29] Björn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic stream processing with latency guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 399–410. https://doi.org/10.1109/ICDCS.2015.48

[30] Microsoft Azure 2020. https://azure.microsoft.com/en-us/global-infrastructure/locations/.

[31] Mininet. 2020. http://mininet.org/.

[32] Habib Mostafaei, Shafi Afridi, and Jemal Abawajy. 2022. Network-aware worker placement for wide-area streaming analytics. *Future Generation Computer Systems* 136 (2022), 270–281. https://doi.org/10.1016/j.future.2022.06.009

[33] Habib Mostafaei, Georgios Smaragdakis, Thomas Zinner, and Anja Feldmann. 2022. Delay-resistant geo-distributed analytics. *IEEE Transactions on Network and Service Management* (2022), 1–1. https://doi.org/10.1109/TNSM.2022.3192710

[34] ATT network delay. 2021. https://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html.

[35] K. Oh, A. Chandra, and J. Weissman. 2020. A network cost-aware geo-distributed data analytics system. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 649–658. https://doi.org/10.1109/CCGrid49817.2020.00-28

[36] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodík, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. 421–434.

[37] Waleed Reda, Kirill Bogdanov, Alexandros Milolidakis, Hamid Ghasemirahni, Marco Chiesa, Gerald Q. Maguire, and Dejan Kostić. 2020. Path persistence in the cloud: A study of the effects of inter-region traffic engineering in a large cloud provider's network. *SIGCOMM Comput. Commun. Rev.* 50, 2 (May 2020), 11–23.

[38] Redis. 2021. https://redis.io/.

[39] The P4.org Applications Working Group. 2020. In-band network telemetry (INT) dataplane specification v2.1. https://github.com/p4lang/p4-applications/tree/master/docs.

[40] Jan Sipke Van Der Veen, Bram Van Der Waaij, Elena Lazovik, Wilco Wijbrandi, and Robert J. Meijer. 2015. Dynamically scaling Apache Storm for the analysis of streaming data. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*. 154–161.

[41] Ververica. 2022. What is Stream Processing. (2022). https://www.ververica.com/what-is-stream-processing. Accessed on June 1.

[42] Cleverton Vicentini, Altair Santin, Eduardo Viegas, and Vilmar Abreu. 2019. SDN-based and multitenant-aware resource provisioning mechanism for cloud-based big data streaming. *Journal of Network and Computer Applications* 126 (2019), 133–149.

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

[44] Ashish Vulimiri, Carlo Curino, Brighten Godfrey, Konstantinos Karanasos, and George Varghese. 2015. WANalytics: Analytics for a geo-distributed data-intensive world. *CIDR 2015* (January 2015).

[45] Y. Wu, Z. Zhang, C. Wu, C. Guo, Z. Li, and F. C. M. Lau. 2017. Orchestrating bulk data transfers across geo-distributed datacenters. *IEEE Transactions on Cloud Computing* 5, 1 (2017), 112–125.

[46] J. Young and T. Barth. 2017. Web performance analytics show even 100-millisecond delays can impact customer engagement and online revenue. (2017). Akamai Online Retail Performance Report.

[47] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. 2018. AWStream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*. 236–252.

[48] A. C. Zhou, B. Shen, Y. Xiao, S. Ibrahim, and B. He. 2020. Cost-aware partitioning for efficient large graph processing in geo-distributed datacenters. *IEEE Transactions on Parallel and Distributed Systems* 31, 7 (2020), 1707–1723.

Received 31 October 2021; revised 1 September 2022; accepted 3 November 2022