2016 IEEE International Congress on Big Data

Geelytics: Enabling On-demand Edge Analytics Over Scoped Data Sources

Bin Cheng, Apostolos Papageorgiou, Martin Bauer
NEC Laboratories Europe, Heidelberg, Germany

***Abstract*—Large-scale Internet of Things (IoT) systems typically consist of a large number of sensors and actuators distributed geographically in a physical environment. To react fast on real time situations, it is often required to bridge sensors and actuators via real-time stream processing close to IoT devices. Existing stream processing platforms like Apache Storm and S4 are designed for intensive stream processing in a cluster or in the Cloud, but they are unsuitable for large scale IoT systems in which processing tasks are expected to be triggered by actuators on-demand and then be allocated and performed in a Cloud-Edge environment. To fill this gap, we designed and implemented a new system called Geelytics, which can enable on-demand edge analytics over scoped data sources via IoT-friendly interfaces to sensors and actuators. This paper presents its design, implementation, interfaces, and core algorithms. Three example applications have been built to showcase the potential of Geelytics in enabling advanced IoT edge analytics. Our preliminary evaluation results demonstrate that we can reduce the bandwidth cost by 99% in a face detection example, achieve less than 10 milliseconds reacting latency and about 1.5 seconds startup latency in an outlier detection example, and also save 65% duplicated computation cost via sharing intermediate results in a data aggregation example.**

I. Introduction

The Internet of Things (IoT) aims to connect all things widely deployed in our physical environments and further transforms our living world into a smart and safe environment. In a typical large-scale IoT system a large number of sensors and actuators are involved: sensors generate real-time IoT data either periodically or by events, such as event streams, image streams, or video streams; on the other hand, actuators can make actions based on received commands or analytics results. By connecting them with real-time stream data analytics in between, IoT systems are able to bridge sensors and actuators together and then form a *closed control loop* to make fast and automated reactions based on low latency analytics results derived from real-time stream data.

It has been predicted that 50 billion devices will be connected to the Internet by 2020. Given that scale, it is no longer a sustainable and economic model to save and process all IoT data in the Cloud, especially when data must be processed in real time and low latency analytics results are expected by actuators to make fast reactions. On the other hand, we see that network virtualization and softwarization is happening and Telecom operators and vendors are working together to standardize mobile edge computing [1]. Sooner or later, the network infrastructure will be able to widely support edge computing. Under this context, it is a timely shift to move IoT data analytics from the Cloud to network edges so that both bandwidth cost and latency can be largely reduced and more time-critical IoT use cases can be supported economically and efficiently.

In the state of the art, most of the existing stream processing frameworks/platforms like Apache Storm [2], Heron [3], S4 [4], Flink [5], and Spark Streaming [6] are designed to manage stream processing tasks within a cluster or in the Cloud. However they are unsuitable for IoT systems since they did not consider the highly distributed Cloud-Edge environment and the unique characteristics of IoT data and systems. For example, in a Cloud-based stream processing system data sources are relatively centralized, usually provided by a message broker like Kafka [7] or an external No-SQL database like MongoDB [8] or HBase [9]. Differently, in IoT systems data sources are geo-distributed in nature, for example, cameras and environmental sensors can be widely deployed at different locations in a smart city. Also, the workload of IoT systems is highly dynamic as sensors and actuators come and go or move around. For instance, sensors and actuators attached to connected vehicles and mobile phone are movable. These IoT characteristics are unique and crucial for IoT edge analytics, but unfortunately they are not considered by existing stream processing systems.

In this paper we present a method of automatically generating, configuring, and managing stream processing tasks to perform on-demand edge analytics over geo-scoped data sources in a Cloud-Edge based system setting. Based on this method, we implement a new stream processing system called Geelytics, which is tailored to IoT systems to enable on-demand edge analytics with regards to edge computing. In the paper we introduce the major design, algorithms, implementation, and interfaces of Geelytics. As compared with the state of the art, our system can make the runtime stream data processing automatically adapted to scoped subscriptions and the geographical availability and mobility of IoT devices. Based on the implemented Geelytics system, we are able to provide low latency analytics results for actuators with less bandwidth consumption and high resource efficiency in many use cases.

The rest of this paper is structured as follows. Section II introduces some use cases on edge analytics that motivate our work and then analyzes the technology gap by looking into the state of art. Section III presents the main concept, system architecture, design issues, and also interfaces of Geelytics. Section IV introduces the implementation of the Geelytics system prototype and then reports some example applications built on top of Geelytics based on its interfaces. Section V discusses some preliminary evaluation results. Finally the paper is concluded with some future work in Section VI.

978-1-5090-2622-7/16 $31.00 © 2016 IEEE
DOI 10.1109/BigDataCongress.2016.21

II. Background and Motivation

A. Edge Computing

Traditionally, IoT data are collected from geo-distributed sensors and then forwarded to the centralized Cloud for saving and processing, e.g., he CiDAP platform presented in our previous work [10]. However, given the trend that the number of connected devices is going to be 50 billion by 2020 and each of them constantly generates data, sending all sensor data to the Cloud is no longer a sustainable and economic model. That is why recently edge computing [11] (also called fog computing [12]) is attracting more and more attention. With edge computing, more data processing can be moved from the Cloud to the edges that are close to where data is generated and where data analytics results are consumed. This way the bandwidth consumption between Cloud and edges could be largely reduced and also low latency could be achieved to make fast reactions. In terms of deployment, the edges could be base stations, IoT gateways, or even IoT devices, depending on how powerful they are. In terms of networks, as we see lots of network operators and vendors are already working on the standard of mobile edge computing, the future mobile network infrastructure will be able to widely support edge computing.

B. Use Cases for Edge Analytics and Requirement Analysis

As we have described in our previous position paper [13], there are many use cases that require IoT data analytics to be performed at the edges in order to achieve fast response time. They exist in various business domains, uch as smart transportation, smart energy, automotive, public safety, and eHealth. For example, in smart transportation, a traffic alert system can send an alert to vehicles that are heading towards an area where the occurrence of an accident has been detected; in public safety, a suspicious person can be immediately detected and captured with high resolution images by real-time face detection and face recognition at network edges. Based on the analysis of many similar use cases, we identify the following common requirements.

1) The same type of data processing needs to be applied to various data sets within different geo-scopes. For example, for a connected car, a driver expects to be informed whenever there is a car accident ahead, therefore some accident detection algorithm needs to be performed based on the traffic information from the road traffic sensors in the region of several hundreds of meters ahead of the car. In this case, to help all drivers in a smart city, the same accident detection processing must be applied to the IoT data generated in different regions.

2) Even for the same set of data sources, different levels of data processing are required to produce various intermediate results for different scope granularities. For example, in a smart city platform, city authorities require different levels of aggregated real-time information for emergency handling, such as traffic level for each street, section, district, or the overall city. A hierarchical data processing over different layers is demanded.

3) Mobility and geo-location of sensors and actuators must be considered in order to provide low latency results and also to efficiently utilize the computation resource over the central Cloud and the network edges. With the capability of edge computing infrastructure, data processing tasks can be deployed at the network edges. However, as sensors or actuators come and go into the system, or move in and out of a region, the task management to meet the first two requirements becomes even more challenging, with regards to how many data processing tasks are needed, how they should be configured properly, and which compute node they should be allocated.

4) Data and results need to be shared across subscriptions, applications, and users so that resources can be efficiently used to achieve maximal benefits. As illustrated in Figure 1. Assume that there are a large number of road traffic sensors deployed in a smart city platform and the following three applications are expected to be enabled on top of the platform. App1 consists of two tasks: a sampling task and a learning task. The sampling task is to select some samples from all road sensor data and the samples will be forwarded to the learning task for learning a predictive model that can identify anomalies like traffic jam or car accidence. The predictive model is then used by the anomaly detection task within App2 and App3, but App2 and App3 are using it in a different way, with different inputs for the same computation. App2 uses the anomaly detection task to monitor the anomaly situations in each predefined region and then calculates the statistical results for each region and also for the entire city. App3 uses the real-time results from the anomaly detection tasks to inform the drivers about nearby car accidents via a geo-fencing task. As seen in this scenario, final results or intermediate results are expected to be shared across applications and users (e.g., authorities in the city operation center and drivers on the road). Also, in App3 the number of demanded task instances is dynamic, depending on the geographical availability and mobility of cars.

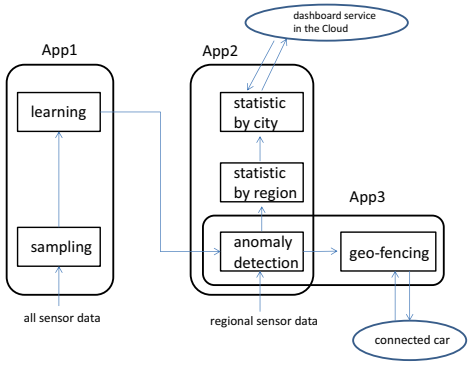

Fig. 1: Illustration of how intermediate results can be shared at various levels of granularity across application topologies

C. Related Work

In the state of the art, lots of systems have been built to enable real-time stream processing in the Cloud, such as Apache Storm [2], Heron [3], Flink [5], Spark Streaming [6], S4 [4], Samza, and BlockMon [14]. However, they are all designed to scale up intensive stream data processing over multiple nodes within a cluster, with different considerations on scalability, programming modeling, fault-tolerance, data communication and routing, and also resource allocation and task scheduling. Some recent studies have been done to extend existing streaming processing platforms from a central data

center to multiple geo-distributed data centers. For instance, Vulimiri et al. [15] [16] focus on orchestrating distributed query execution and adjusting data replication across data centers in order to minimize bandwidth usage in WANalytics, which is implemented based on Hive; Heintz et al. extend Storm to support efficient geo-distributed stream analytics for grouped data aggregation across multiple edges [17]; Pu et al. [18] consider supporting low latency analytics on geo-distributed datasets with optimized data and task placement in Iridium, which is built on top of the Apache Spark framework. All of these studies have already considered data analytics over multiple geo-distributed sites, but they only consider the case where analytics results are derived from the entire dataset and consumed from the Cloud.

As a new trend for IoT systems, edge analytics aims to leverage the power of both edge computing and cloud computing to support real time stream processing. Only a few early stage studies have been done in this area, for example, a recent work from Carnegie Mellon University [19] proposes a cloudlet based platform for performing video analytics at network edges, but it only focuses on video streams and does not consider how to utilize the geographical availability and mobility of IoT devices to do customized and on-demand stream processing on top of the edges and the Cloud. In addition, some industrial systems have been done to explore edge analytics, such as AGT IoT analytics platform [20], Geo-distributed analytics from ParStream [21], Quarks from IBM [22]. However, for the time being all of them still focus on how each edge device can support customized data processing locally and then communicate with the Cloud. The automatic task instance generation and management issue in a Cloud-Edge setting is still not touched by the state of the art.

D. Technology Gap

Although a number of frameworks exist for real-time stream processing in the state of the art, there is still a gap in order to meet the requirements of IoT edge analytics, due to the following reasons. First, the number of task instances and the input data sources of each instance must be determined according to the geographical availability of sensors. Second, task assignment needs to consider the proximity of sensors, actuators, and edge compute nodes, in order to minimize internal data traffic and achieve low latency. Third, intermediate results from each task need to be shared in a meaningful granularity with flexible interfaces, in order to maximize the sharing opportunity of data analytics tasks and increase the efficiency of resource usage. Lastly, there is lack of interaction between stream processing engines and IoT systems. To fill the gap we design and implement a new stream processing system called Geelytics, which is tailored to IoT edge analytics to provide those missing features. Our previous position paper [13] on Geelytics presented its initial architecture design and discussed its use cases, technical challenges, and major design issues, but no detailed solution was provided. In this paper we focus on how to enable on-demand stream processing over scoped data sources in Geelytics and also present its detailed design, interfaces, and core algorithms in the following sections. We also report some performance evaluation results based on the implemented system prototype and application use cases.

III. Geelytics Design

A. System Setting

Geelytics is mainly designed for large scale IoT systems that consist of a large number of geo-distributed data producers, result consumers, and the compute nodes that are located both at the network edge and in the Cloud. Data producers are typically sensors, connected cars, glasses, video cameras, and mobile phones, being connected to the system via different types of edge networks (e.g., WiFi, ZigBee, or 4G, but maybe also fixed networks). They are constantly reporting heterogeneous, multi-dimensional, and unstructured data over time. On the other hand, result consumers are actuators or external applications that expect to receive real-time analytics results from sensor data and then take fast actions accordingly. Both data producers and result consumers could be either stationary or mobile. In between them there are lots of compute nodes geographically distributed at different locations. In general compute nodes are heterogeneous in terms of resource and data processing capability and they can be located at different layers of the network. For example, they could be small data centers at base stations in a cellular network or IoT gateways in factories or shops.

B. Terminology

The Geelytics system is designed as an IoT edge analytics platform that allows consumers to dynamically trigger certain stream data processing either at network edges or in the Cloud to derive real-time analytics results from a set of data producers defined by a geo-scope. At very high level, it works like a distributed pub/sub system to interact with geo-located sensors and actuators, meanwhile having a built-in stream processing engine that can perform on-demand IoT stream data analytics based on the underlying Cloud-Edge system infrastructure. The following terms are defined to explain its main concept.

Physical Location and Logical Location: these two types of information are used by Geelytics to describe the location profile of data producers, result consumers, and compute nodes. We assume that all of them will have a *physical location*, including the following attributes: GPS coordinates (*latitude, longitude*), address of its geographic location comprising *section, district, city, and country*.

In addition, all compute nodes will have another location called *logical location* to express their position in the hierarchical logical network topology that reflects the network correlation between different compute nodes. Compute nodes are distributed at different locations, either at different network edges or in the Cloud, but they are organized in a hierarchical manner. Each compute node is associated with the following logical location information: *layer, site_num, node_num, parent_site_num*. Each compute node has a unique node number but it can share the same site number with a set of other compute nodes located together in the same deployment location, such as in the same cluster. The logical location information can be given manually by system administrators during the setup phase or be assigned automatically by some addition component according to the underlying network topology when a new compute node joins. In general the hierarchical logical network topology should reflect the underlying physical network topology. In addition, each data producer and result

consumer is associated with a nearby compute node when they join the system.

Scoped Tasks: similar to existing stream processing platforms like Apache Storm, a topology in the shape of a directed acyclic graph (DAG) must be first provided by a developer to define the high level data processing logic of an application, consisting of multiple correlated tasks at different layers. However, in order to express the unit of a task instance that will be automatically generated and configured by Geelytics later at runtime, the developer needs to specify a *scope granularity* for each task in the topology based on location attributes. Each specific task instance covers a specific scope at that granularity level. Therefore, the granularity parameter indicates how many instances are needed to cover all the given data sources and how each task instance should be configured with a specific scope value in order to define its input streams. Some visualized examples can be seen in Fig. 5 later. In the end, the following parameters are required to specify a scoped task.

• *name*: a unique name given by the developer to identify a task in the application topology.

• *granularity*: defining the granularity of each task instance, which must be specified based on the location-related attributes, such as section, district, city, or site. For example, if the granularity is section, it means each task instance is responsible for a specific section, covering all data sources associated with that specific section.

• *operation*: the name of the operation to be performed by the task. Note that the operation is some program code to be invoked by the task for performing the actual data stream processing during the runtime. Each operation can take a certain number of streams in, perform some stream data analytics, and then report the calculated results back.

• *input_streams* and *output_streams*: defining the type of input streams and output streams for the task. They are linking the current task to other tasks to form the application topology.

In Geelytics every compute node supports docker containers with the docker engine [23] and the implementation of an operation is wrapped up as a docker image to be deployed and executed within a docker container on any compute node. Docker allows us to package an operation with all of its dependencies into a standardized image unit that can be executed within a very lightweight container. When the task is instantiated during the runtime, it will be configured accordingly and automatically with a set of input parameters and those parameters are further used to configure and launch its operation. Therefore, the configuration of each task instance determines which input streams will be taken by the launched operation to do its internal data analytics.

Scoped Subscriptions: a scoped subscription is sent by a result consumer to trigger the demanded data processing for generating its required result. The scoped subscription consists of a type name and a set of *geo-scope constraints*. The type name in the subscription is the type name of some output streams in the topology, determining the set of tasks to be involved in the demanded data processing logic. The geo-scope constraints within the subscription are a set of constraints to limit the set of selected input streams to be processed by the demanded data processing for generating the requested results.

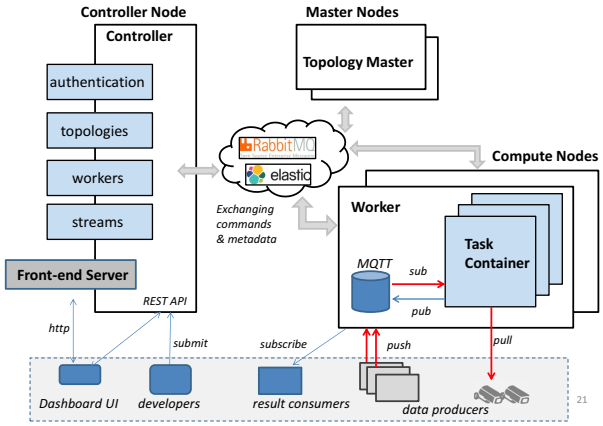

Fig. 2: System Architecture of Geelytics

Each constraint is defined based on the geographic location attributes of data sources.

C. Architecture Overview

Fig. 2 illustrates the overall system architecture of Geelytics. The entire system consists of three major components: a *controller* running on a *controller node*, a large number of *workers* running on geo-distributed *compute nodes* either in the Cloud or at the network edge, a set of *topology masters* that can be running on the controller node together with the controller or separately on their own dedicated *master nodes* (up to the system scale and workload in the real setup).

• *Controller*: supporting web UI based and REST based interfaces for application developers to define and submit operations and scoped tasks to form specific application topologies; monitoring the status of the entire system and presenting them to application developers, including the available streams, workers, topologies, and running task instances. It also checks the health of all other nodes like workers and topology masters based on their periodic heartbeat messages. In addition, as the single entry point of the system, the controller authenticates all data producers or result consumers and also assigns a nearby worker to them when they join the system, based on the location proximity calculated from their GPS attributes. Since all system metadata are saved and updated via the backend repository in Geelytics, the controller can be easily duplicated with multiple instances, just to avoid the single point of failure.

• *Topology Master*: taking the necessary information (application topology saved in repository, status of compute nodes and running tasks reported by each worker, availability of generated streams, and their profile information) from the repository (illustrated by Fig. 3) and then managing task instances accordingly, including generating and configuring task instances, assigning them to running workers, and also maintaining the running task instances to adapt to device mobility. The idea to separate the controller and the topology masters is to avoid overloading the controller for better scalability.

• *Worker*: each compute node runs a worker, which consists of four major components: *MQTT broker* serving pub/sub requests from external data publishers, result subscribers, and also local running tasks; *Executor* receiving configured tasks from the topology masters and then instantiating the tasks in a

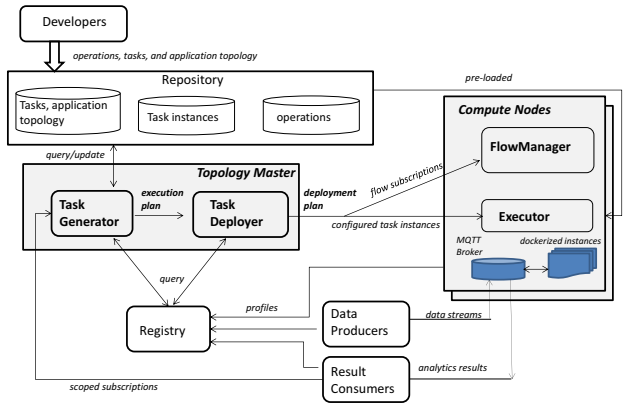

Fig. 3: Main Workflow in Geelytics

docker container; *FlowManager* establishing data flows across different compute nodes when two correlated task instances are assigned to different compute nodes; *Monitor* collecting and reporting the status of the current compute node, such as resource usage, running tasks, and health information. In addition, a local cache is managed by the docker engine to keep the images of all operations, which are prefetched by the workers on all compute nodes when a topology is submitted.

Two external services ElasticSearch and RabbitMQ are used by Geelytics as enablers. ElasticSearch is for indexing all generated streams registered by all compute nodes and supporting efficient and sophisticated stream queries from both workers and topology masters. It is also used as the repository to save the metadata of all other entities, such as operations, tasks, topologies, workers, and topology masters. RabbitMQ is the message bus to exchange messages and commands between different components in the systems with high throughput and low latency.

D. IoT-friendly interfaces

As further illustrated by Fig. 3, once the system is set up, developers first need to define scoped tasks with customized scope granularities to form an application topology, which represents the high-level data processing logic of the IoT applications, and then submit them to be saved in the repository. Except the REST based interfaces for developers or system operators to define and submit operations, scoped tasks, and application topologies, Geelytics also provides the interfaces for checking system status.

Geelytics provides friendly interfaces for both data producers and result consumers to interact with the system. In Geelytics all data producers report their availability, profiles, and data streams to the system, managed by the Registry based on ElasticSearch [24]. The way to fetch the data streams generated by data producers can be *push-based* or *pull-based*. In the push-based approach, data producers publish their stream data to the MQTT broker on the nearby compute node; in the pull-based approach, data producers just announce the URLs of their streams, and later on it is up to task instances to fetch the data directly. For a data producer, it first has to ask the controller to find a nearby worker and then registers its data stream via the nearby worker with the following details: its device ID and location, the generated stream type, and the manner to provide the stream data (push-based or pull-based). A unique ID will be returned to the producer as the global identity of its data stream. If the stream is pull-based (for example, a web camera), a URL must be provided for accessing data as well; if the stream is push-based, using the unique ID as the topic, the producer publishes the generated data to the broker provided by the nearby worker via MQTT.

For a result consumer, it also needs to ask the controller to find a nearby worker first. After that, it sends a scoped subscription to the nearby worker for triggering some real-time data processing over the specified data sources. A subscription ID is returned to the consumer to make a further subscription to the broker. The consumer can receive the subscribed results as soon as they are produced by the triggered task instances in Geelytics. Those running tasks will be terminated once the consumer decides to unsubscribe to the result or its leaving without notice has been detected.

E. Execution Plan and Deployment Plan

As shown in Fig. 3, a geo-scoped subscription will be sent by result consumers to trigger the demanded data processing to produce the required results. The demanded data processing is expressed by an execution plan, which consists of a set of configured tasks generated and configured by the Task Generator based on defined scoped tasks in the topology, the geo-locations of available data producers, and the received subscription. The type name in the geo-scoped subscription determines the set of scoped tasks that are involved in the demanded data processing logic, according to the application topology. On the other hand, the geo-scope constraint determines which data producers should be used to provide input data streams for generating the requested results, according to the latest geo-location information of all available data producers. A top-down approach of generating task instances for a given scoped subscription is further presented in Algorithm 1.

After generating the entire execution plan, the task generator will subtract some existing and reusable tasks from the entire execution plan to generate a minimal execution plan for the current subscription. Different subscriptions can have different scopes to cover data sources in different regions. Therefore, some configured tasks in the entire execution plan might be already triggered by other subscribers in a different subscription before. This can be checked based on the records in the repository. For those configured tasks, their generated results can be reused in the current execution plan and there is no need to spawn them again. So the Task Generator will check the available configured tasks in the previous execution plans and subtract them from the current execution plan for saving more computation resource.

Fig. 4 shows an example of how an execution plan can be dynamically and automatically derived from a pre-defined topology. Assume that there are three scoped tasks A, B, and C within an application topology and an implemented operation called *min*, which is to calculate the min temperature in a set of temperature value streams. All scoped tasks A, B, C are defined based on the same operation *min*, but with different scopes. Task A has the scope section, which means each task A instance is responsible for aggregating data from the data sources in a specific section. The number of task A in

Algorithm 1 $generateTaskInstances(r, S, R)$

Input:
r: the root node of the involved task tree
S: the set of selected streams
R: the set of constraints to select data streams
Output:
F: the forest of constructed task instances

if $(r = null)$ **then return** null ▷ already reach the leaf and no more sub tasks
end if
$F \leftarrow \emptyset$
$g \leftarrow r.granularity$
V = $searchUniqueValues(g, R, S)$ ▷ find all unique geo-attribute values for the given granularity in the selected streams
for each v in V **do**
  $i \leftarrow createOneTaskInstance(r)$
  for each child task c of r **do**
    $\hat{R} \leftarrow R \cup \{g = v\}$
    $i.Children \leftarrow generateTaskInstances(c, S, R)$
    $i.outputs \leftarrow prepareOutputStreams(r.outputs)$
    if $i.Children = null$ **then**
      $i.inputs \leftarrow selectInputStreams(S, \hat{R})$
    else
      $i.inputs \leftarrow collectInputStreams(i.Children)$
    end if
  end for
  $F \leftarrow F \cup \{i\}$
end for
return F

the execution plan can be only determined with the location value of all available data producers. Assume that there are 5 data producers (p1, p2, p3, p4, p5) when a scoped [type = temperature, scope = district: D1] subscription is received, which expects to receive the aggregated results at district level for all data producers within District D1. The geo-location profiles of Data Producers p1, p2, p3, p4, p5 are denoted as [D1, S1, (x1, y1)], [D1, S1, (x2, y2)], [D1, S2, (x3, y3)], [D1, S2, (x4, y4)], and [D1, S3, (x5, y5)] respectively. Given the location information of those data producers, three task A instances and one task B instance will be created and configured accordingly for the entire execution plan.

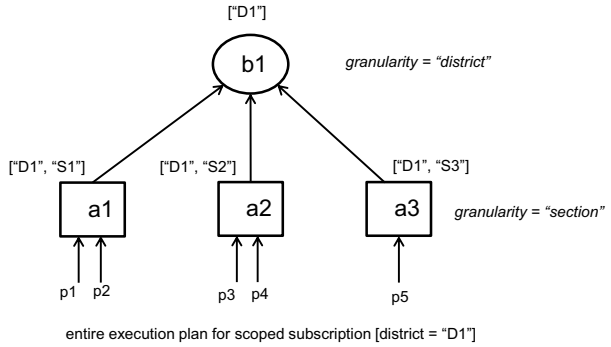

Fig. 4: Example of a generated execution plan

With the generated execution plan, the Task Deployer further figures out an optimized deployment plan based on the logical locations of data producers and result consumer in a hierarchical network topology. The deployment plan includes a set of configured tasks instances that will be assigned to the corresponding workers for executing (done by the Executor module) and also a set of flow subscriptions for the FlowManager module to establish the necessary data flow across different compute nodes. A heuristic algorithm has been used by the Task Deployer to produce an optimized deployment plan, in the purpose of reducing cross-layer and cross-site traffic and data transmission latency. The following information is utilized in Algorithm 2: 1) the hierarchical topology of workers based on their logical location information; 2) capacity of all active workers for accepting new task instances (for the time being, the capacity is calculated in terms of number of running task instaces and each worker has a limited capacity c). The optimization goal is to minimize the cross-layer traffic first and then reduce the cross-site traffic, without overloading any compute nodes. The function $findWorker$ is to search for a worker from the workers associated with its input data sources (for a leaf task instance) or child task instances (for a non-leaf task instance). If no worker has free capacity, the Task Deployer continue to find another worker from the parent site at upper layers in the hierarchical topology.

Algorithm 2 $taskAssignment(r, S, R)$

Input:
F: the forest of all generated task instances
W: the set of active workers
S: the set of selected streams
Output:
F: the forest of all assigned task instances

for each root instance t in the top layer of r **do**
  LowLayerFirst(t, W, S)
end for
procedure LOWLAYERFIRST(t, W, S) ▷ post-order tree travel to assign task instances
  if $t.Children = null$ **then**
    $t.worker \leftarrow findWorker(t.inputs)$
  else
    for each child instance i in $t.Children$ **do**
      LowLayerFirst(i, W, S)
    end for
    $t.worker \leftarrow findWorker(t.Children)$
  end if
end procedure

F. Summary

Based on the design presented above, Geelytics is able to achieve the following features. *1) Enabling on-demand stream processing over scoped data sources*: stream processing tasks are triggered only when their results are demanded by consumers and only the involved data processing tasks in the application topology will be used to construct the required processing logic, rather than the entire topology; consumers can use a particular scope to specify the data streams from which data producers should be fetched by the constructed processing; *2) Providing low latency analytics results with less bandwidth cost*: stream processing tasks should be assigned to the compute nodes close to where data sources are located and where results are consumed so that the latency and bandwidth consumption to develop the required analytics results can be

largely reduced; *3) Providing IoT-friendly interfaces*: various data producers and result consumers can interact with the Geelytics system, which helps IoT systems to easily and dynamically form a closed control loop among sensors, edge analytics, and actuators; *4) Maximizing computation sharing across tasks and topologies*: intermediate analytics results are associated with scopes under various granularity and they can be accessed and shared across processing tasks either in the same topology or across different topologies.

IV. System Prototype

Based on the design and algorithms presented in the previous section, we have prototyped the Geelytics system in the Go language, including the implementation of the controller, topology master, and worker and also the task generation and assignment methods. The system has been deployed and tested in our lab testbed. Three example applications have been implemented based on four implemented operations to verify its interfaces and also to demonstrate its applicability for IoT systems. Note that separating the execution of operations out of the Geelytics system and then communicating them via a pub/sub interface provides better isolation between the Geelytics edge analytics platform and its applications and also more freedom on the implementation of operations. This is because, using a docker container to run an operation, the resource used by the operation can be further configured and controlled by the platform and also developers can implement an operation in any programming language and are free to use any third-party libraries.

Fig. 5 shows the visualization of these three topologies that are constructed from a subset of operations. The first application is *data-aggregation*, to calculate the minimal temperature value in a given time interval for temperature sensors at different geographical levels. The three scope tasks are using the same operation with various scope granularities. The second application is *outlier-detection*, to monitor the total number of temperature outliers for each section. The third application is *crowd-counting*, to count the total number of detected persons for each section via camera-based face detection, where the facecounter operation is implemented in Python using opencv.

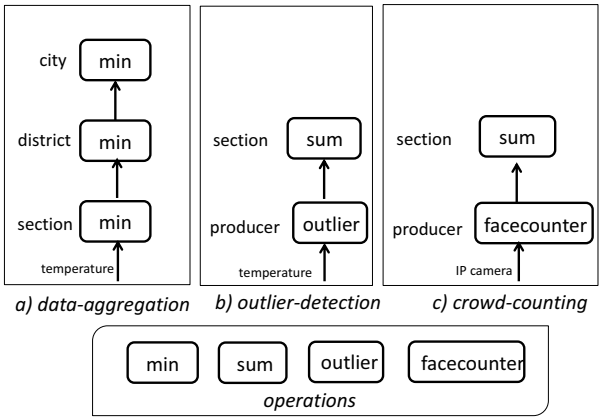

Fig. 5: Application Examples

V. Performance Evaluation

Together with the implemented application examples, we evaluate the system in terms of bandwidth consumption, latency, efficiency, and overhead. All experiments are carried out in a simulated Cloud-Edge environment, including three compute nodes at the edges and two compute nodes in the Cloud. According to the measurement results in [25], the RTT delay between the edges and the Cloud is set to 100ms on average. In the simulated setting, the compute nodes at the edges can accept 60 task instances at most while the compute nodes in the Cloud are more powerful and can handle 200 task instances. Two Raspberry Pis are connected to the system via WIFI to simulate sensors and actuators, generating temperature data and MJPEG streams and also interacting with the Geelytics system. The temperature data are simulated based on the sensor data log collected from the smart city testbed in Santander [10], and the MJPEG streams are fetched from the same connected web camera.

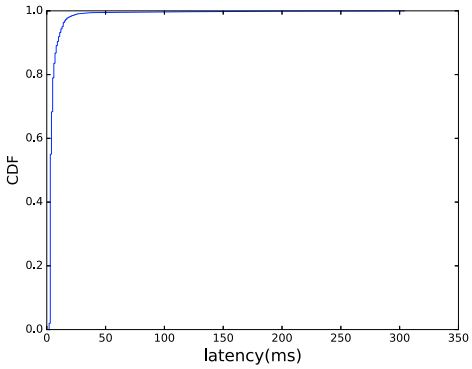

Fig. 6: Reacting latency

Bandwidth Reduction: We first measured the bandwidth consumption that can be reduced at each layer. In the data-aggregation application, the *min* operation is to calculate the minimal temperature value for each 10 seconds at the level of section, district, and city. In the test we simulated 200 temperature sensors from 20 sections in 5 districts of the same city. Assume that a dashboard service running in the Cloud triggers a subscription to request the real-time aggregation result for all temperature sensors in the entire city, we can reduce the bandwidth consumption between edges and the Cloud by 80%, as compared to the pure Cloud-based solution. In the example of crowd-counting, 5 camera sensors are simulated and the bandwidth cost can be reduced by 99%, because 5 facecounter task instances are created and assigned to the compute nodes at the edge and each of them just needs to produce a counting number from a MJPEG video stream.

Latency: We then measured two types of latency, *reacting latency* and *startup latency*. The reacting latency means the delay from the time when an event is reported by a sensor to the time when an actuator receives the notification and can start to act on it. The startup latency means the delay from the time when an actuator starts to join the system to the time when it receives the first result from the system. Fig. 6 shows the reacting latency measured from the outlier-detection example, in which an outlier is defined as events that report a temperature out of the normal range. We simulated an actuator to subscribe the outlier in a specific section and ran it on

the same Pi as the temperature sensor that reports abnormal temperature values. The result shows that about 90% of outlier events can be detected within 10ms and only less than 1% of them cannot be detected within 50ms. This is much lower than 100ms, the average RTT between devices and the Cloud. In addition, the startup latency for the actuator is in the range of 800ms to 2000ms, 1.5 seconds on average. This is the result when the actuator only triggers one task instance at the edges. Of course, as the number of involved task instances grows, the startup latency will increase. However, this is just one-time cost for result consumers.

Efficiency: it is simply calculated by the percentage of shared task instances as compared with the case where no task can be shared across different subscriptions. In the data-aggregation example, to server the subscriptions from the dashboard service for generating the statistical results for each section, district, and the overall city, only 25 task instances are required in the case of with task sharing, leading to about 65% reduction against the 71 task instances in the case of without task sharing.

Overhead: There is certain overhead to run an operation within a docker container. Based on our measurements, it increases less than 5% execution time and there is no clear additional memory and bandwidth cost as compared with native process. For example, the latency and throughput in the crowd-counting are nearly the same as using native process. However, as compared with using virtual machines, the cost is much lower. For example, launching a virtual machine could take a few minutes but in Geelytics it takes less than 2 seconds to launch a task instance within a docker container on average. Also the good deployability of using docker containers is a big plus, especially for edge analytics tasks that require advanced third-party libraries, such as the image processing library opencv or the machine learning library R.

VI. Conclusions and Future Work

This paper presents a method of automatically generating, configuring, and deploying stream data processing tasks in a Cloud-Edge based system environment. The key concept, *scoped tasks and subscriptions*, can enable on-demand edge analytics over scoped data sources. The concept has been realized in a system prototype called Geelytics. We present its system architecture, major design issues, and core algorithms. As compared to the state of the art, Geelytics is tailored for large scale IoT systems to do customized edge analytics. It not only provides low latency analytics with optimized bandwidth consumption, but also achieves better resource efficiency.

One of the remaining issue in Geelytics is to fully support device mobility, meaning the generated execution plan and optimized deployment plan can be updated as IoT devices are moving. This requires live migration of running task instances and also a mobility manager to monitor the mobility pattern and changes of IoT devices. Other issues will be to enhance system security, resource allocation, and auto-scaling of compute nodes at network edges.

VII. Acknowledgment

The work presented in the paper has been partially funded by the European Union's 7th Framework Programme for Research (FP7/2013-2016) under Grant Agreement No.609062.

References

[1] (2016) The mobile-edge computing initiative. [Online]. Available: http://www.etsi.org/technologies-clusters/technologies/mobile-edge-computing

[2] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "StormTwitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 147–156.

[3] S. Kulkarni, N. Bhagat, M. Fu, and et al., "Twitter Heron: Stream Processing at Scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 239–250.

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Proceedings of IEEE International Conference on Data Mining Workshops (ICDMW)*, 2010, pp. 170–177.

[5] Apache flink. [Online]. Available: https://flink.apache.org

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

[7] Kafka. [Online]. Available: http://kafka.apache.org

[8] Mongodb. [Online]. Available: https://www.mongodb.org

[9] Apache hbase. [Online]. Available: https://hbase.apache.org

[10] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs, "Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander," in *IEEE Big Data Congress 2015*, June 2015.

[11] S. Yi, C. Li, and Q. Li, "A Survey of Fog Computing: Concepts, Applications and Issues," in *Proceedings of the 2015 Workshop on Mobile Big Data*, 2015, pp. 37–42.

[12] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, 2012, pp. 13–16.

[13] B. Cheng, A. Papageorgiou, F. Cirillo, and E. Kovacs, "Geelytics: Geo-distributed edge analytics for large scale iot systems based on dynamic topology," in *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, Dec 2015, pp. 565–570.

[14] F. Huici, A. Di Pietro, B. Trammell, J. M. Gomez Hidalgo, D. Martinez Ruiz, and N. d'Heureuse, "Blockmon: A High-performance Composable Network Traffic Measurement System," *ACM SIGCOMM Computer Communication Review*, vol. 42, pp. 79–80, 2012.

[15] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, and et al., "WANalytics: Geo-Distributed Analytics for a Data Intensive World," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1087–1092.

[16] A. Vulimiri, C. Curino, B. Godfrey, J. Padhye, and G. Varghese, "Global Analytics in the Face of Bandwidth and Regulatory Constraints," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA*, 2015.

[17] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing Grouped Aggregation in Geo-distributed Streaming Analytics," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 133–144.

[18] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica, "Low Latency Analytics of Geo-distributed Data in the Wide Area," in *Proceedings of ACM SIGCOMM*. ACM, 2015.

[19] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge Analytics in the Internet of Things," *IEEE Pervasive Computing*, vol. 14, pp. 24–31, Jun. 2015.

[20] "AGT IoT Analytics Platform," https://www.agtinternational.com/iot-analytics/iot-analytics/analytics-the-edge/, 2015.

[21] "ParStream Geo-distributed Analytics," https://www.parstream.com/product/parstream-geo-distributed-analytics/, 2015.

[22] Quarks. [Online]. Available: http://quarks-edge.github.io

[23] "Docker," https://www.docker.com.

[24] "ElasticSearch," https://www.elastic.co.

[25] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 139–152.