FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities

Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa

***Abstract*—Smart city infrastructure is forming a large scale Internet of Things (IoT) system with widely deployed IoT devices, such as sensors and actuators that generate a huge volume of data. Given this large scale and geo-distributed nature of such IoT systems, fog computing has been considered as an affordable and sustainable computing paradigm to enable smart city IoT services. However, it is still a major challenge for developers to program their services to leverage benefits of fog computing. Developers have to figure out many details, such as how to dynamically configure and manage data processing tasks over cloud and edges and how to optimize task allocation for minimal latency and bandwidth consumption. In addition, most of the existing fog computing frameworks either lack service programming models or define a programming model only based on their own private data model and interfaces; therefore, as a smart city platform, they are quite limited in terms of openness and interoperability. To tackle these problems, we propose a standard-based approach to design and implement a new fog computing-based framework, namely *FogFlow*, for IoT smart city platforms. FogFlow's programming model allows IoT service developers to program elastic IoT services easily over cloud and edges. Moreover, it supports standard interfaces to share and reuse contextual data across services. To showcase how smart city use cases can be realized with FogFlow, we describe three use cases and implement an example application for anomaly detection of energy consumption in smart cities. We also analyze FogFlow's performance based on microbenchmarking results for message propagation latency, throughput, and scalability.**

***Index Terms*—Edge computing, Internet of Things (IoT), parallel programming.**

I. Introduction

NOWADAYS cities are becoming more and more digitalized and connected as numerous sensors have been widely deployed for various purposes. For example, deployments in smart cities include CO_2 sensors for measuring air pollution, vibration sensors for monitoring bridges, and cameras for watching out potential crimes. Those connected devices form a large scale of Internet of Things (IoT) system with geographically distributed endpoints, which generate a huge volume of data streams over time. Potentially, the generated data can help us increase the efficiency of our city management in various domains, such as transportation, safety, and environment (e.g., garbage management). However, to utilize the data efficiently we need to have an elastic IoT platform that allows developers to easily program various services on top of a shared and geo-distributed smart city IoT infrastructure.

In the past, most of the existing city IoT platforms are built only based on cloud, such as CityPulse [1] and our previous city data and analytics platform [2]. However, this is no longer a sustainable and economical model for the next generation of IoT smart city platforms, given that many city services (e.g., car accident detection) require ultralow latency and fast response time. Moreover, bandwidth and storage costs can be substantially high if we send all sensor data, such as video frames to the cloud. Recently there is a new trend to offload more computation from the cloud and device layer to the middle layer components which are IoT gateways and edge/core networks, called *fog computing* [3]. While fog computing perfectly fits the geo-distributed nature of smart city infrastructure, it is still challenging for smart city IoT platforms to adapt to this new computing paradigm. The heterogeneity, openness, and geo-distribution of the new cloud-edge environment raise much more complexity on the management of data and processing tasks than the centralized cluster or cloud environments. Therefore, we need a sufficient and flexible programming model with open interfaces that allow developers to implement various IoT services on top of the cloud-edge environment without dealing such complexities.

The current state of art on fog computing, such as Foglets [4], has been mainly focused on how to optimize task deployment over distributed edges in terms of saving bandwidth and reducing latency. However, there is not much work that has been done to explore the programming model for fog computing. The existing studies either just reuse the programming models from existing frameworks (e.g., Apache Storm and Spark) or come up with their own programming models with nonstandardized interfaces. In this paper we argue that both solutions are not suitable for smart city IoT platforms to adopt fog computing in terms of *openness, interoperability, and programmability*.

To tackle these problems, we take a standards-based approach and propose a next generation service interface (NGSI)-based programming model to enable easy programming of IoT services over cloud and edges. In this paper we introduce the overall architecture of our new fog

Manuscript received January 31, 2017; revised July 16, 2017; accepted August 4, 2017. Date of publication August 30, 2017; date of current version April 10, 2018. This work was supported in part by the European Union's Horizon 2020 Research and Innovation Programme within the CPaaS.io project under Grant 723076 and in part by the FIESTA-IoT Project under Grant 643943. *(Corresponding author: Bin Cheng.)*

B. Cheng, G. Solmaz, F. Cirillo, and E. Kovacs are with NEC Laboratories Europe, 69115 Heidelberg, Germany (e-mail: bin.cheng@neclab.eu).

K. Terasawa and A. Kitazawa are with NEC Solution Innovators, Ltd., Tokyo 8666, Japan.

Digital Object Identifier 10.1109/JIOT.2017.2747214

2327-4662 © 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

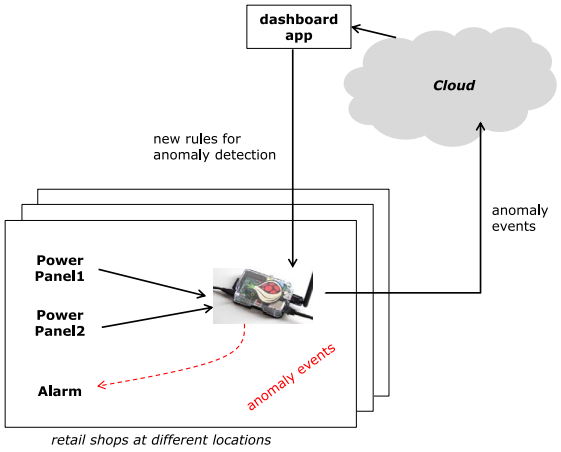


Fig. 1. Detecting abnormal electricity usage in retail stores.

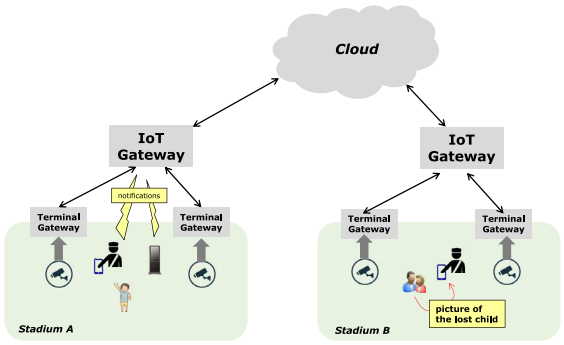


Fig. 2. Video surveillance in stadiums.

computing framework, namely *FogFlow*, and also report its core technologies for supporting the proposed programming model. Furthermore, we introduce some concrete application example to showcase how IoT services can be easily realized on top of our NGSI-based programming model. The main contributions of this paper are highlighted as follows.

- *Standard-Based Programming Model for Fog Computing:* We extend dataflow programming model with *declarative hints* based on the widely used standard *NGSI*, which leads to two benefits for service developers: 1) fast and easy development of fog computing applications, this is because the proposed hints hide lots of task configuration and deployment complexity from service developers and 2) good openness and interoperablity for information sharing and data source integration, this is because NGSI is a standardized open data model and API and it has been widely adopted by more than 30 cities all over the world.
- *Scalable Context Management:* To overcome the limit of centralized context management, we introduce a distributed context management approach and our measurement results show that we can achieve much better performance than existing solutions in terms of throughput, response time, and scalability.

II. Use Cases and Requirements

In this section, we shortly describe three smart city use cases which require programming of IoT services over cloud and edge nodes.

A. Smart City Use Cases

Use Case 1 (Anomaly Detection of Energy Consumption): The first use case study is for retail stores to detect abnormal energy consumption in real-time. As illustrated in Fig. 1, a retail company has a large number of shops distributed in different locations. For each shop, a Raspberry Pi device (edge node) is deployed to monitor the power consumption from all PowerPanels in the shop. Once an abnormal power usage is detected on the edge, the alarm mechanism in the shop is triggered to inform the shop owner. Moreover, the detected event is reported to the cloud for information aggregation. The aggregated information is then presented to the system operator via a dashboard service. In addition, the system operator can dynamically update the rule for anomaly detection.

Use Case 2 (Video Surveillance in Stadiums): The second use case is for providing stadium security with video surveliance and real-time analytics. Fig. 2 illustrates this use case based on three layers: 1) terminal gateway; 2) IoT gateway; and 3) cloud. In the lower layer, terminal gateway devices are deployed to process the video streams captured by cameras. In the upper layer, each stadium has an IoT gateway to perform further data processing. Terminal gateways and the IoT gateway are connected to the local area network of the stadium. In the top layer, all IoT gateway devices are connected to the cloud via the Internet. The following services are expected to be enabled.

- *Crowd Counting:* Aggregation of the number of people extracted from the captured video streams and the total number of people at each area to show the stadium crowdedness in real-time.
- *Finding Lost Child:* When a child gets lost in a stadium, their parents ask the staff for help. Based on the picture provided by the parent, video analytics tasks are launched dynamically on demand at the edge nodes to identify the lost child in real-time. Once the child is found, the staff is notified and a digital signage close to the child is actuated in order to ensure the safety of the child.

Use Case 3 (Smart City Magnifier): The last use case is an application for visualization of smart cities which we named as smart city magnifier (SCM). SCM provides a user interface for displaying the results from environmental monitoring, critical situations, such as safety alerts, as well as the view of city-wide deployments. Fig. 3 illustrates SCM for environmental monitoring. Environmental monitoring results include traffic, air pollution, and crowd levels. Overall condition of the situations for each of them are shown with green, yellow, or red lights (left). Furthermore, these levels are also displayed with graphs (right). The dashboard includes the data analytics results from past measurements (historical data), current (real-time) measurements, as well as future predictions based on the current trends.

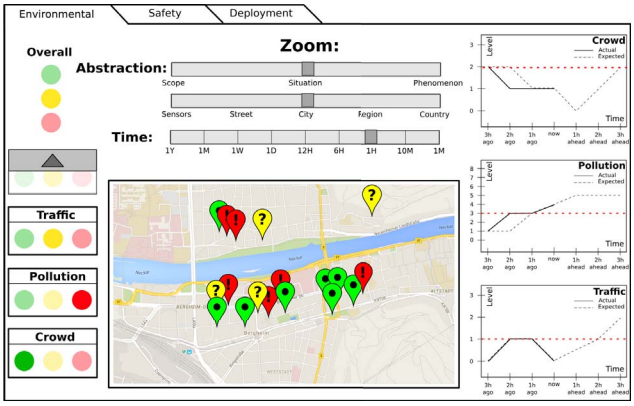


Fig. 3. Illustration of SCM.

As shown in Fig. 3, the visualization tool can be set based on three parameters: 1) *space* for specifying the geographic scope (the map view); 2) *time* for the evaluation time window or the forecasting horizon; and 3) *abstraction* for defining the level of detail of the view. The level of detail varies based on the analytics results.

For instance, for air pollution results, lower abstraction can be CO_2 levels, while higher abstraction can be overall air quality. Furthermore, abstraction depends on the geographic scope such that it varies if the scope is a building, a street, or a city.

B. High Level Requirements

Let us briefly discuss the main requirements of the three use cases. The general requirements for such systems include *system interoperability and openness* since different IoT system components, such as things, edge nodes, middlewares (e.g., context broker), and developers need to connect through interfaces. Considering the variety of "things" in the IoT, this is a challenging task. Other than those there exist certain performance requirements. The IoT services for smart cities require certain data processing capabilities including offline big data analytics through frameworks, such as Apache MapReduce as well as real-time stream processing through frameworks such Apache Spark. Therefore, a major performance requirement is *dynamic orchestration of the data processing tasks*. Various data processing tasks (e.g., video analytics and air quality measurements) need to be performed on the shared cloud and edge resources.

III. FogFlow: Programming IoT Services Over Cloud and Edges Based on NGSI

To meet the openness and interoperability requirements of IoT smart city platforms, we propose a standards-based approach for designing and implementation of the FogFlow programming model based on the two standards: 1) Dataflow (de facto standard) and 2) NGSI (official standard).

Dataflow is a popular programming model to decompose applications which are widely used by cloud service developers for big data processing in cloud environments. Google Cloud Dataflow and Amazon Data Pipeline are among the data processing services that are built based on the dataflow programming model. As a unified programming model for both batch and stream processing, Dataflow is still suitable for defining fog services; however, it is missing certain features or extensions to adapt to the challenges introduced by fog computing. For example, from the underlying infrastructure perspective, fog computing requires more geo-distributed, dynamic, and heterogeneous infrastructure than cloud computing. From the application perspective, fog services usually require low latency and location awareness. This is especially true for smart city applications. Thus, we introduce *declarative hints* and extend the traditional dataflow programming model for enabling efficient fog computing.

In the dataflow programming model, the data processing logic for a service is usually decomposed as multiple *operators*. Operators form a directed acyclic graph (DAG) called *topology* through the linked inputs and outputs among different operators. Traditionally operators are defined as functions with certain APIs, but this is no longer a suitable model to define operators for fog computing due to its bad isolation and limited flexibility and interoperability. Conversely, FogFlow requires service developers to define operators as dockerized applications based on NGSI [5] (details to be discussed in Section III-A).

Let us introduce the overall system achitecture of FogFlow first with some background information on the NGSI standard. We then present the detailed design of our NGSI-based programming model and how such a programming model is supported by scalable context management and dynamic service orchestration in FogFlow.

A. Next Generation Service Interface

As an open standard interface from Europe, open mobile alliance NGSI [6] is currently used in industry and academia as well as in large scale research projects, such as FIWARE [7] and Wise-IoT [8]. In 2015 an open and agile smart cities initiative [9] has been signed by 31 cities from Finland, Denmark, Belgium, Portugal, Italy, Spain, and Brazil, for adopting the NGSI open standard in their smart city platforms. It is strategically important to design our programming model based on NGSI in order to achieve openness and interoperability in the areas of IoT and smart cities.

From the technical perspective, NGSI defines both the *data model* and *communication interface* to exchange contextual information between different applications via context brokers. The NGSI data model characterizes all contextual information as context entities, where each entity must have an ID and a type. Entities also optionally have a set of attributes and metadata related to domains and attributes. Typically metadata includes the source of information, observation areas, and the location of the IoT device. The NGSI communication interface defines a lightweight and flexible mean to publish, query, and subscribe to context entities. *NGSI10* and *NGSI9* are, respectively, designed for managing the data values of context entities and their availability (e.g., discovery of entities). As

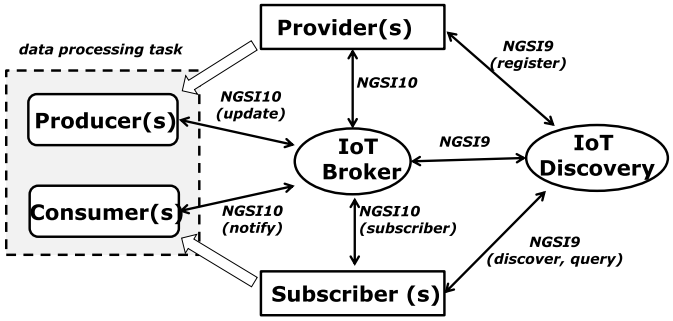


Fig. 4. Typical NGSI-based interactions and system diagram.

opposed to the existing message brokers (e.g., MQTT), NGSI not only defines a unified data model to express contextual data (both raw sensor data and derived intermediate results) but also provides missing features that are highly demanded by geo-distributed fog computing. For instance, geoscope-based resource discovery and subscription are needed by our service orchestrator for dynamic configuration and management of the data processing tasks.

Fig. 4 illustrates the usage of NGSI in different scenarios. Typically an IoT Broker (e.g., Aeron [10] and Orion [11]) middleware is deployed between context provider(s) and context consumer(s) to allow them to exchange NGSI-based context entities. Meanwhile, an IoT Discovery component creates an index for the availability of all registered context entities. Providers register the availability of their context data via NGSI9 to make them discoverable. To subscribe or query any context entities via NGSI10, consumers must find out which provider offers the requested context entities via NGSI9 request to IoT Discovery. In FogFlow, each data processing task acts as a provider to publish its outputs, while it also acts as a consumer to receive its input streams. However, for easy programming of operators, the FogFlow framework dynamically registers the generated outputs via NGSI9 and subscribes the inputs via NGSI10 on behalf of data processing tasks. In the end, developers only need to deal with NGSI10 update and notify when they implement operators. More details can be seen in Section III-C.

B. Architecture Overview

The system architecture of FogFlow is illustrated in Fig. 5. The figure includes the FogFlow framework, *geo-distributed infrastructure resources*, and FogFlow's connection with the users (system operator and service developers) and external applications through its API and interfaces. Infrastructure resources are vertically divided as *cloud*, *edge nodes*, and *devices*. Computationally intensive tasks, such as big data analytics can be performed on the cloud servers, while some tasks, such as stream processing can be effectively moved to the edge nodes (e.g., IoT gateways or endpoint devices with computation capabilities). Devices may include both computation and communication capabilities (e.g., tablet computer) or only one of them (e.g., beacon nodes advertising Bluetooth signals). The FogFlow framework operates on these geo-distributed, hierarchical, and heterogeneous resources, with three logically separated divisions: 1) service management; 2) data processing; and 3) context management.

The service management division includes task designer, topology master (TM), and docker image repository, which are typically deployed in the cloud. Task designer provides the Web-based interfaces for the system operators to monitor and manage all deployed IoT services and for the developers to design and submit their specific services. Docker image repository manages the docker images of all dockerized operators submitted by the developers. TM is responsible for service orchestration, meaning that it can translate a service requirement and the processing topology into a concrete task deployment plan that determines which task to place at which worker.

The data processing division consists of a set of workers $(w_1, w_2, \ldots, w_m)$ to perform data processing tasks assigned by TM. A worker is associated with a computation resource in the cloud or on an edge node. Each worker can launch multiple tasks based on the underlying docker engine and the operator images fetched from the remote docker image repository. The number of supported tasks is limited by the computation capability of the compute node. The internal communication between TM and the workers is handled via a advanced message queuing protocol-based message bus, such as RabbitMQ to achieve high throughput and low latency.

The context management division includes a set of *IoT Brokers*, a centralized *IoT Discovery*, and a *Federated Broker*. These components establish the data flow across the tasks via NGSI and also manage the system contextual data, such as the availability information of the workers, topologies, tasks, and generated data streams. IoT Discovery handles registration of context entities and discovery of them. This component is usually deployed in the cloud. IoT Brokers are responsible for caching the latest view of all entity objects and also serving context updates, queries, and subscriptions. In terms of deployment, IoT Brokers are distributed on the different nodes in the cloud and on the edges. They are also connected to the other two divisions (workers, task designer, TM, and external applications) via NGSI. Federated Broker is an extended IoT Broker used as a bridge to exchange context information with all other Federated Brokers in different domains. For instance, Federation Broker enables communication from one deployment in an European smart city (e.g., Domain A: Heidelberg) to another in a Japanese smart city (e.g., Domain B: Tokyo). These deployments are considered as two different domains. Within the same domain, all IoT Brokers and Federated Broker are connected to the same IoT Discovery.

C. NGSI-Based Programming Model

In FogFlow, an IoT service is represented by a *service topology* which consists of multiple *operators*. Each operator receives certain types of input streams, performs data processing, and then publishes the generated results as output streams. The FogFlow programming model defines the way of how to

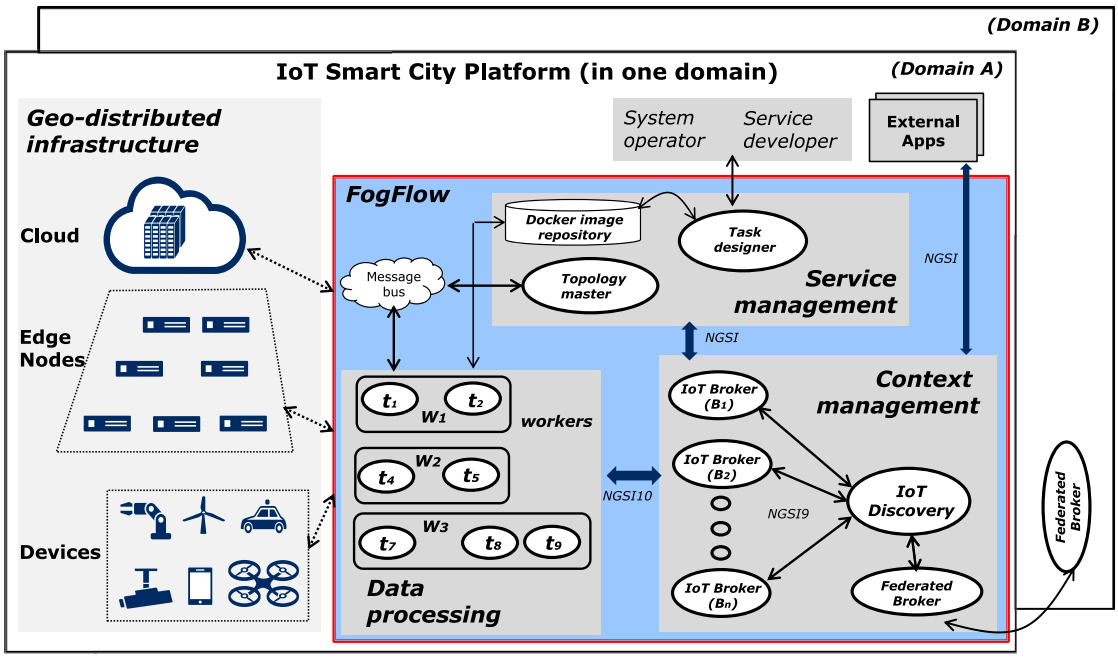


Fig. 5. System architecture of FogFlow.

```
name: AnomalyDetector
operator: anomaly
groupBy: shopID
input_streams:
    - type: PowerPanel
      shuffling: unicast
      scoped: true
    - type: Rule
      shuffling: broadcast
      scoped: false
output_streams:
    - type: Anomaly
```

Fig. 6. Example of task specification written in YAML.

specify a service topology using declarative hints and how to implement operators based on NGSI.

1) Declarative Hints: Developers decompose an IoT service into multiple operators and then define its service topology as a DAG in JSON format to express the data dependencies between different operators. This is similar to the traditional dataflow programming model. On the other hand, the FogFlow programming model provides *declarative hints* for developers to guide service orchestration without introducing much complexity. Currently, it requires developers to specify two types of hints in the service topology: 1) granularity and 2) stream shuffling.

Granularity hint is associated with each operator in the service topology and represented by the "groupBy" property, as shown by a task specification example (written in YAML language) in Fig. 6. The granularity hint is defined using the name of one stream attributes. In FogFlow, every data stream is represented as a unique NGSI context entity generated and updated by either an endpoint device or a data processing task. Different types of metadata are created by FogFlow on the fly to describe the data stream. For instance, metadata includes which device or task is producing the data stream, the location of the producer, which IoT Broker is providing the stream, and so on. Regarding the geo-distributed nature of the underlying infrastructure, some common granularity hints are geo-location related attributes, such as "section," "district," "city," or "ProducerId." These granularity hints are later used as an input by TM to decide the number of task instances to be created and configured for each operator during system runtime.

Stream Shuffling hint is associated with each type of input stream for an operator in the service topology, represented by the "shuffling" property. TM uses this hint as additional information to decide how to assign matched input streams to task instances. Based on the granularity hint, multiple task instances could be instantiated from the same operator, but they can be configured with different set of input streams. For each operator, the type of its input streams determines which type of streams should be selected to configure its task instances, but its shuffling property can further decide how the selected streams should be assigned to the task instances as their inputs. The value of the shuffling property can be either "broadcast" or "unicast." The broadcast value means the selected input streams should be repeatedly assigned to every task instance of this operator, while the unicast value means each of the selected input streams should be assigned to a specific task instance only once.

Fig. 7 shows a concrete example of how these two types of hints are used by TM to create and configure data processing tasks for service orchestration. The left side illustrates a service topology with two simple operators, A and B, which is designed for use case 1 in Section II-A. The right side illustrates the execution plan generated by TM. Operator B is named as "AnomalyDetector" and its detailed specification

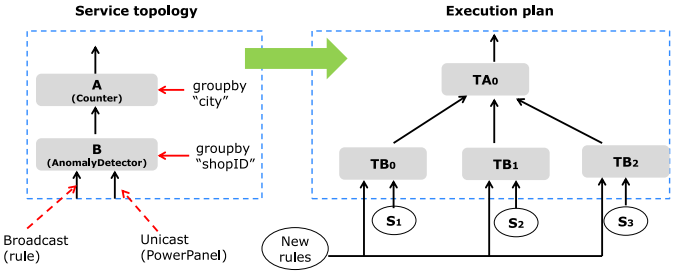


Fig. 7. Left: service topology example with declarative hints and right: execution plan generated by TM.

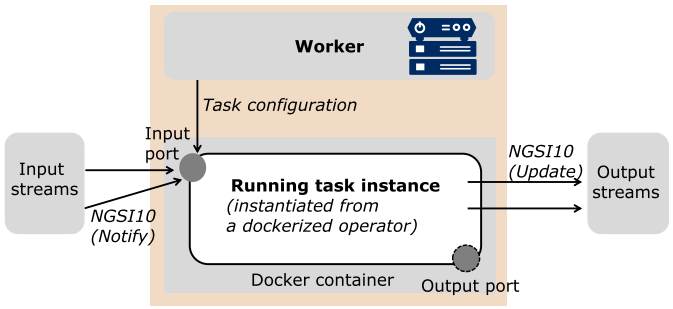


Fig. 8. Interactions of the task instance with FogFlow.

is listed in Fig. 6. The goal of operator B is to detect the abnormal usage of electricity for each shop based on a given rule. Based on this requirement, the granularity of operator B is defined by "shopID," meaning that TM needs to create a dedicated AnomalyDetector task instance for each shop. Operator B has two types of input streams: 1) the anomaly detection rule and 2) the measurement from power panel. Assume that three power panel devices are from three different shops S1, S2, and S3. In this case three task instances (TB0, TB1, and TB2) must be created because its operator granularity is based on shopID. Also, each task instance is assigned with the stream from a specific power panel but they all share the same detection rule as another input. This is because the shuffling property of the rule input stream is broadcast while the shuffling property of the PowerPanel devices is unicast.

2) NGSI-Based Operators: Developers need to implement each operator in a service topology as a dockerized application. As illustrated in Fig. 8, once a worker instantiates a dockerized operator application (a task instance running in a docker container), the task instance interacts with the FogFlow framework via the following steps.

First, before starting its internal processing logic, the task instance receives a JSON-based configuration from the worker through environment variables. The initial configuration includes which IoT Broker the task instance should talk to and also the metadata of its input and output streams. Later on, if there is any configuration change, those changes can be sent to the task instance via a listening port (input port). In FogFlow, the important stream metadata required by the task instance include: 1) the entity type and entity ID of the associated stream that can be used by the task instance to know which entity to subscribe as inputs and which entity to update as outputs and 2) the way of how the stream data is provided from the producer to consumers, which can be PUSH-based or PULL-based.

More specifically, PUSH-based means that the stream entity will be updated by its context producer actively and context consumers can receive the updates from IoT Broker via subscriptions, while PULL-based means that the stream entity only represents the information of the context producer and the actual stream data must be pulled by the task instance from a service URL, which is part of the stream entity. For example, a temperature sensor that actively sends temperature observations periodically can be represented as a PUSH-based stream entity; a webcam that sends captured images or video streams on request can be represented as a PULL-based stream entity.

Second, after the task instance is launched and configured, it will start to get its input streams and process the received data. If the stream is PUSH-based, the task instance can receive all input stream data as NGSI10 notify via the input port without sending any subscritpion, because the worker issues NGSI10 subscriptions to the IoT Broker on behalf of the task; if the stream is PULL-based, e.g., video streams from an IP camera, the task instance needs to fetch the input stream data from a provided URL in the stream metadata.

Lastly, once some results are generated from the received stream data, the task instance publishes or announces them as output streams. If the output stream is PUSH-based, the task instance sends the generated outputs to the IoT Broker as NGSI10 update under the specified entity type and ID; if the output stream is PULL-based, the worker can register the output stream on behalf of the task. With this design we allow the worker to handle more management complexity in order to dynamically configure and establish the data flows cross different task instances. Therefore, we can try to reduce the complexity of the implementation of dockerized operators and reduce the required effort from developers; on the other hand, we can provide enough flexibility for various application use cases and also comply with the NGSI standard.

D. Scalable Context Management

The context management system is designed to provide a global view for all system components and running task instances to query, subscribe, and update context entities via the unified NGSI. Our NGSI-based context management has the following additional features. These features are different from the ones provided by traditional pub-sub message brokers, such as MQTT-based Mosquitto or Apache Kafka, but they play an important role to support FogFlow's NGSI-based programming model.

- It provides separate interfaces to manage both context data (via NGSI10) and context availability (via NGSI9). This feature enables flexible and efficient data management with standardized open APIs.
- It supports not only ID-based and topic-based query and subscription but also geoscope-based query and subscription. This feature enables FogFlow to efficiently manage

all geo-distributed resources, such as workers, tasks, and generated streams.
- It allows a third-party to issue subscriptions on behalf of the subscribers. This feature provides the chance to achieve the minimized complexity within the operators and the maximized flexibility of the operators.

Context availability represents the outline of context data. Usually context availability changes less frequently than context data over time. For example, the following availability information is used to register context entities: 1) context type; 2) attribute list; and 3) domain metadata (e.g., provider information). The FogFlow programming model benefits from these separated interfaces because of two reasons. First, FogFlow can automatically manage context availability information on behalf of tasks so that we reduce the complexity of operator implementation for developers. Second, context availability and context data updates are forwarded to task instances via separate channels; therefore, we do not have to feed the unchanged context availability information to the tasks repeatedly. This can significantly reduce the bandwidth consumption of cross-task communication.

In FogFlow, whenever a service topology is triggered, a large number of geo-distributed task instances are created, configured, and instantiated on the fly in a very short time. This introduces two challenges to the context management system: 1) it must be fast enough to discover available resources in a specified scope and 2) it must provide high throughput to forward context data from one task to another. In addition, we assume that data processing tasks can only be instantiated from a service topology within a single FogFlow-enabled smart city IoT platform. However, they should also be able to share and reuse context data from other smart city IoT platforms as long as these platforms are compatible with NGSI. In terms of terminology, each smart city IoT platform is represented by a domain and the FogFlow framework can be duplicated to realize other smart city platforms for different domains. Different domains can be different cities or business domains, such as transportation and e-health in the same city.

Currently, the Orion Context Broker developed by Telefonica [11] is the most popular message broker supporting NGSI; however, it is not scalable due to the lack of distributed solutions and federation support. To achieve a scalable context management system, we apply the following two approaches.

Scaling Light-Weight IoT Broker Up With Shared IoT Discovery: As illustrated in Fig. 5, within each smart city platform domain a large number of IoT Brokers work together in parallel with a shared IoT Discovery. The centralized IoT Discovery provides a global view of context availability of context data and provides NGSI9 interfaces for registration, discovery, and subscription of context availability. Each IoT Broker manages a portion of the context data and registers data to the shared IoT Discovery. However, all IoT Brokers can equally provide any requested context entity via NGSI10 since they can find out which IoT Broker provides the entity through the shared IoT Discovery and then fetch the entity from that remote IoT Broker.

Connecting Different Domains via Federated Broker: In each domain there is one Federated Broker responsible for announcing what the current domain provides and fetching any context data from the other domains via NGSI10. Within the domain, Federated Broker informs IoT Discovery that it can provide any context data out of the current domain. Federated Broker needs to coordinate with the other Federated Brokers in different domains. The coordination can be done using different approaches, such as table-based, tree-based, or mesh-based. In the table-based approach, all Federated Brokers can know which Federated Broker is responsible for which domain via a shared and consistent table that is maintained and updated by a bootstrap service. In the tree-based (hierarchical) approach, a hierarchical relationship between different domains is configured manually or maintained automatically by a root node. In the mesh-based approach, each Federated Broker maintains a routing table based on its partial and local view and relies on a next hop from the routing table to locate its targeted domain. In practice, which approach to take is up to the actual scale of domains. Due to a limited number of domains in our current setup, FogFlow takes the table-based approach and looks up the Federated Broker for a target domain directly from the shared table.

E. Dynamic Service Orchestration

Once developers submit a specified service topology and the implemented operator docker images, the service data processing logic can be triggered on demand by a high level processing requirement. The processing requirement is sent (as NGSI10 update) to the submitted service topology entity. It is issued either by the system operator via task designer or by a subscriber via an external application. The following three inputs are necessary for TM to carry out service orchestration.
- *Expected Output:* It Represents the output stream type expected by external subscribers. Based on this input parameter, TM decides which part of service topology should be triggered. This allows FogFlow to launch only part of the data processing logic defined in the service topology.
- *Scope:* It is a defined geoscope for the area, where input streams should be selected. This allows FogFlow to carry out the selected data processing logic for the selected area, such as a specific city or a polygon area.
- *Scheduler:* It Decides which type of scheduling method should be chosen by TM for task assignment. Different task assignment methods lead to different service level agreements because they aim for different optimization objectives. For instance, we provide two methods in FogFlow: one for optimizing the latency of producing output results and the other for optimizing the internal data traffic across tasks and workers.

For a given processing requirement, TM performs the following steps (illustrated in Fig. 9) to dynamically orchestrate tasks over cloud and edges.
- *Topology Lookup:* Iterating over the requested service topology to find out the processing tree in order to produce the expected output. This extracted processing tree represents the requested processing topology which is further used for task generation.

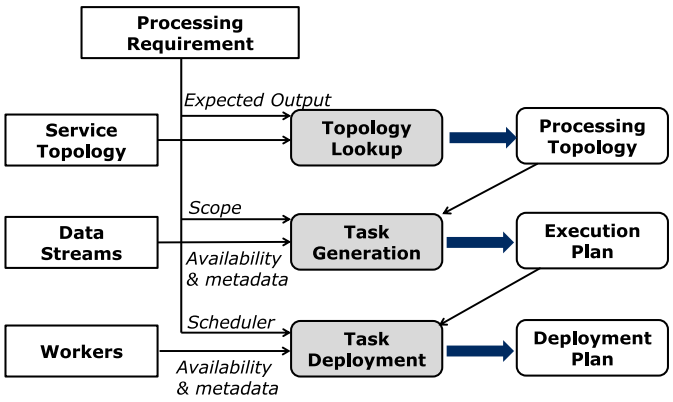


Fig. 9. Major steps of service orchestration.

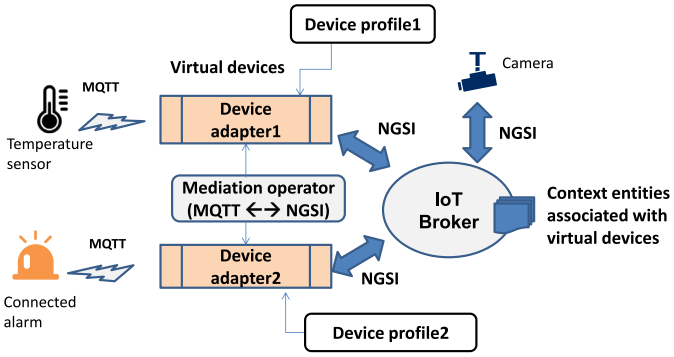


Fig. 10. Example to illustrate the concept of virtual devices.

- *Task Generation:* First querying IoT Discovery to discover all available input streams and then deriving an execution plan based on this discovery and the declarative hints in the service topology. The execution plan includes all generated tasks that are properly configured with right input and output streams and also the parameters for the workers to instantiate the tasks.
- *Task Deployment:* Performing the specified scheduling method to assign the generated tasks to geo-distributed workers according to their available computation capabilities. The derived assignment result represents the deployment plan. To carry out the deployment plan, TM sends each task to the task's assigned worker and then monitors the status of the task. Each worker receives its assigned tasks and then instantiates them in docker containers. Meanwhile, worker communicates with the nearby IoT Broker to assist the launched task instances for establishing their input and output streams.

Since the focus of this paper is on the NGSI-based programming model and its supporting system framework, we skip the algorithms for task generation and task assignment. More details can be found in our previous GeeLytics platform [12] study.

F. Virtual Sensor

In FogFlow we dynamically composite multiple data processing tasks to form the data processing flow of each IoT service based on the standardized NGSI data model and service interface. However, to interact with sensors and actuators, we still need to deal with the diversity of various IoT devices. For example, some devices might not be able to talk with the FogFlow system via NGSI due to their limited upload bandwidth; some existing devices might only support other protocols, such as MQTT or COAP; or some devices need to be turned into a sleep mode from time to time in order to save their battery lifetime. To handle these issues, we introduce *virtual device* to unify the communication between FogFlow and IoT devices.

Fig. 10 illustrates this concept. Any physical device that already supports NGSI can be integrated to the FogFlow system directly by interacting with a nearby IoT Broker. On the other hand, a physical device that does not support NGSI must be integrated into the FogFlow system as a virtual device via a *device adapter*. As shown in Fig. 10, the device adapter is a proxy that is triggered and initialized from a device profile in order to mediate the communication between the device and the FogFlow system. The proxy is a task instance instantiated from a specific *mediation operator*, which handles the detailed mediation procedure for different types of devices. For conversion between different interfaces, different mediation operators must be developed.

By adding a device profile, we can trigger an adapter task to integrate a non-NGSI device into the FogFlow system. The *device profile* provides which operator should be used and the necessary configuration information to interact with the device. In the end, the physical device is presented as an NGSI context entity via its device adapter. All FogFlow services just need to interact with the NGSI context entity associated with the physical device, such as sending context updates or subscribing to some attribute changes. Using this virtual device approach, we handle the availability and reliability issues of IoT devices within the device adapters.

IV. Use Case Validation

In this section, we discuss our implementation of an example application which realizes the first use case (described in Section II-A): anomaly detection of energy consumption in retail stores. The service topology (illustrated in Fig. 11) is defined to meet the requirements of the use case. Two data processing operators are defined as follows.

- *Anomaly Detector:* This operator is to detect anomaly events based on the collected data from power panels in a retail store. It has two types of inputs: 1) *detection rules* which are provided and updated by the operator and 2) *sensor data* From power panel. The detection rules input stream type is associated with broadcast, meaning that the rules are needed by all task instances of this operator. The granularity of this operator is based on shopID, meaning that a dedicated task instance will be created and configured for each shop.
- *Counter:* This operator is to count the total number of anomaly events for all shops in each city. Therefore, its task granularity is by city. Its input stream type is the output stream type of the previous operator (Anomaly Detector).

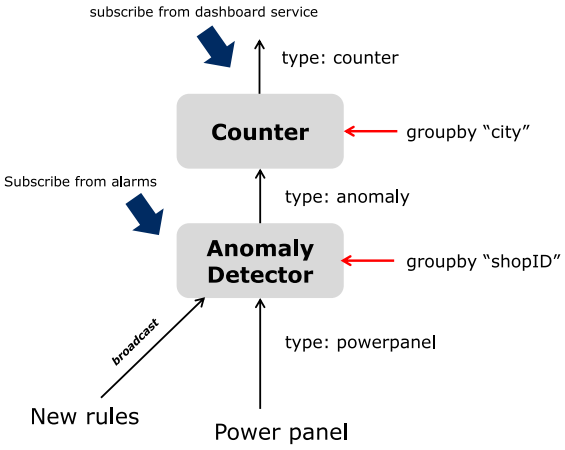


Fig. 11. Illustration of how intermediate results are shared at various levels of granularity across application topologies.

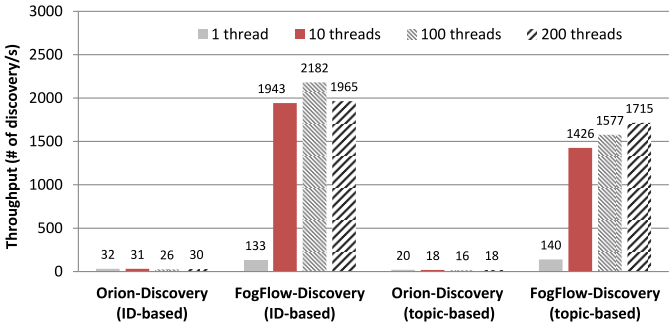


Fig. 12. FogFlow-Discovery and Orion-Discovery throughputs for a matched ID- and topic-based query among 10 000 entities.

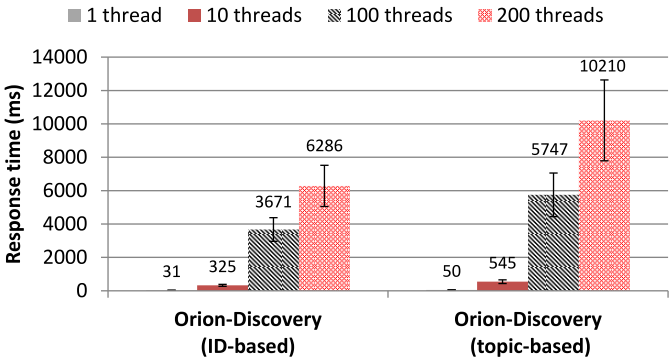


Fig. 13. Orion-Discovery response times for ID- and topic-based queries.

There are two types of result consumers: 1) a dashboard service in the cloud, which subscribes to the final aggregation results generated by the counter operator for the global scope and 2) the alarm in each shop, which subscribes to the anomaly events generated by the Anomaly Detector task on the local edge node in the retail store.

The second and third use cases can be realized in a similar way by defining a processing topology based on their specific requirements.

V. Performance Evaluation

This section includes our experimental evaluation of NGSI-based context management systems of the FogFlow framework. Our analyses include the efficiency of context availability discoveries and context transfers in the smart city scale. Moreover, we analyze the scalability of FogFlow using multiple IoT Brokers. Our metrics are throughput (number of messages per second) and response time/message propagation latency. The results show the performance of the IoT Brokers (our FogFlow-Broker and Orion-Broker) as well as the IoT Discoveries (our FogFlow-Discovery and Orion-Discovery).

The query for discovery and update requests are generated using Apache JMeter performance testing tool. We analyze three types of queries for context discovery: 1) ID-based; 2) topic-based; and 3) geoscope-based. In ID-based queries, a match (discovery) occurs when the queried entity ID is already registered as available. In topic-based (pattern-based) queries, a match happens when there is a registered entity ID of the similar pattern defined by a regex (e.g., searching pattern "Room.*" for the registered entity ID "Room12"). Geoscope-based queries match when the entity is registered with the location that is inside the defined geographical area (defined by latitude/longitude and radius values). IoT Brokers mainly forward updates from the context producers to the context consumers. We analyze the throughput and propagation latency of updates under various cases.

Considering the shared discovery in the FogFlow architecture, we conduct laboratory experiments for IoT Discovery using single server instance, which has 12 CPUs, 128 GB memory, and 256 GB disk storage. For IoT Brokers, we conduct experiments on AWS cloud using multiple server instances for context updates. Since IoT Brokers in FogFlow can be widely deployed on edge nodes, such as IoT Gateways, we use only micro instances in our tests, where each micro instance has 1 CPU and 1 GB memory. We consider various number of threads (clients) in a smart city (1, 10, 100, 200 entities) accessing the context management system at the same time. The threads may represent devices, such as sensor nodes in a smart city or applications accessing the system.

Let us first start with the ID- and topic-based query performance for discovery. Fig. 12 shows the average throughput of query for FogFlow-Discovery and Orion-Discovery. The query always returns 1 match out of 10 000 registered entities. For 1 client, Orion-Discovery has throughput of less than 50 discoveries per second, while FogFlow-Discovery serves more than 100 discoveries. With the increased number of threads, throughput of FogFlow-Discovery significantly increases to more than 1900 in ID-based and more than 1400 for topic-based queries. On the other hand, when the number of threads increase, Orion-Discovery has even worse performance, showing that using Orion-Discovery can cause a bottleneck in certain scenarios, where multiple IoT Brokers query at the same time.

The average response times of Orion-Discovery are shown Fig. 13. We observe that Orion-Discovery returns fast responses in the case of one thread and ten threads. However, the response times dramatically increase in the case of 100 or 200 threads. In the case of 200 threads in topic-based queries, the average response time is more than 10 s. On the other

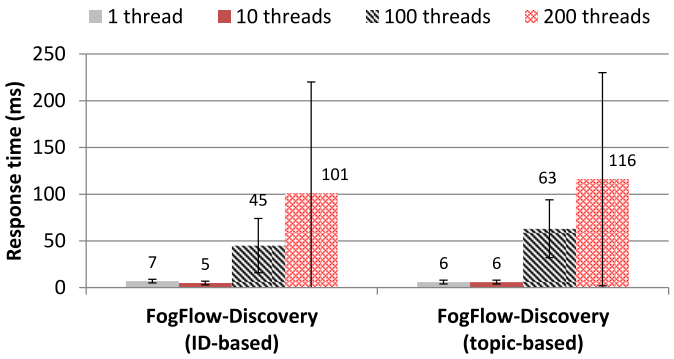


Fig. 14. FogFlow-Discovery response times for ID- and topic-based queries.

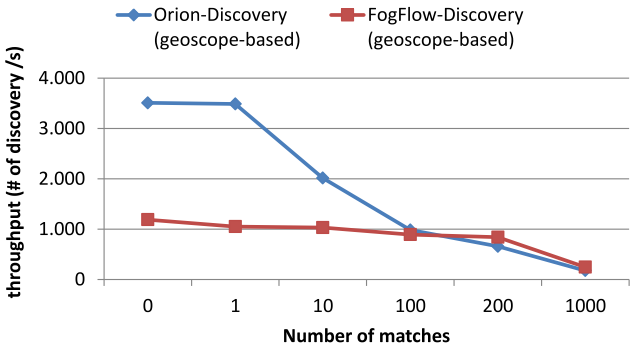


Fig. 15. FogFlow-Discovery and Orion-Discovery throughputs in geoscope-based queries.

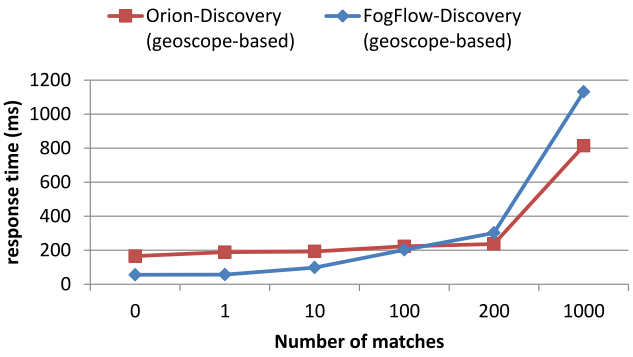


Fig. 16. FogFlow-Discovery and Orion-Discovery response times in geoscope-based queries.

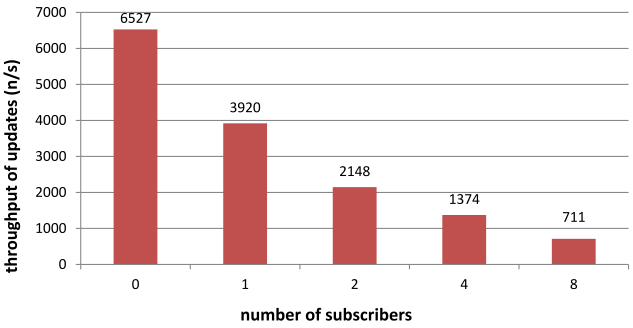


Fig. 17. FogFlow-Broker and Orion-Broker throughput of updates between publishers and subscribers.

hand, as can be seen in Fig. 14, the response times of FogFlow-Discovery are shorter in all cases. Moreover, for more than 100 threads, FogFlow-Discovery still provides high performance with an average response time around 100 ms per query. Overall, considering a smart city with multiple IoT Brokers querying for ID- and topic-based discoveries, we find FogFlow-Discovery clearly a more reliable component to handle such loads. While Orion-Discovery provides the same functionalities, its performance is not sufficient for such scenarios.

Let us now discuss the geoscope-based query performances of the discovery components. We compare the response times for different numbers of matched entities in Fig. 16 among 10 000 registered entities using 200 threads. For 0-match case, where IoT Brokers query for an entity which is located outside of any registered areas, Orion-Discovery has significantly better performance compared to FogFlow-Discovery. Same performance difference exists for 1-match case. On the other hand, this performance gap diminishes with the increased number of matches, where the volume of transferred data increases. FogFlow-Discovery produces slightly higher throughput for more than 100 matches, as can be seen in Fig. 15. Furthermore, both components provide a reliable service for queries up to 1000 matches. Fig. 16 shows the response times of the geoscope-based queries. Both discovery components achieve short response times in most cases. Only exception is seen in the case of 1000 matches, which produces a certain load in the network. In that case, Orion-Discovery performs slightly better (≈800 ms) than FogFlow-Discovery (≈1000 ms).

Orion-Discovery is a built-in feature of Orion Context Broker while in FogFlow-Discovery is a stand-alone component separated from FogFlow-Brokers. With this design, FogFlow is able to scale up brokers to handle data transfer between different tasks in parallel. We now look at the performance of FogFlow-Brokers. Fig. 17 shows the throughput of one FogFlow-Broker as the number of subscribers increases. When there is no subscriber, FogFlow-Broker's throughput reaches 6500 updates per second while Orion-Broker can only achieve 2200 updates per second. We observe that FogFlow-Broker performs much better than Orion-Broker in terms of update throughput. This is mainly because FogFlow-Broker keeps the latest updates and all subscriptions in memory while Orion-Broker has to save them into the database (MongoDB). Furthermore, we find that the update throughput decreases as the number of subscribers increases as FogFlow-Broker becomes busy with forwarding received updates to all subscribers.

We also test the propagation latency of updates from publishers to subscribers when FogFlow-Brokers are not overloaded. To calculate the latency, we run a program to simulate both the publisher and the subscriber on the same cloud instance. The latency is defined as the time difference between when a update message is sent out by the publisher and when the update is received by the subscriber. Table I lists the propagation latency of updates in three different situations: 1) both the publisher and the subscriber contact with the same broker; 2) they communicate with two different brokers located at the same data center; and 3) they communicate

TABLE I
PROPAGATION LATENCY OF UPDATE

Different test cases	Propagation latency	
	Avg. (ms)	Std. (σ)
Same broker	0.7	0.7
Different broker, same data center	<50	<50
Different broker, different data center	430.8	194.2

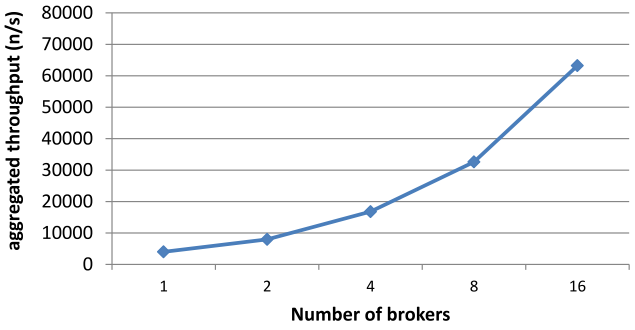


Fig. 18. Aggregated throughput of updates for different number of parallel brokers.

with two different brokers located at two different data centers.

The results shows that the propagation latency via the same broker is very low (less than 1 ms on average). On the other hand, if the data flow between publishers and subscribers is established via two different brokers, the propagation latency increases. In this case the latency depends on, where these two brokers are located. If they are located at the same data center, the propagation latency can still be less than 50 ms on average; however, it becomes unpredictable since we do not know whether our cloud instances are at the same rack. If the two brokers are located at different data centers, the average propagation latency significantly increases (≈500 ms). This result indicates that we need to carefully select and configure a proper broker for each running task in order to minimize data propagation latency for any time critical services.

We do further experiments for the scalability of FogFlow-Brokers when they work on different topics in parallel, but still share the same discovery component. As can be seen in Fig. 18, the aggregated throughput of updates increases linearly with the increased number of brokers. Note that the number of brokers increase two times for each result. This result shows that FogFlow-Brokers can scale up very well without overloading the shared discovery component. This is due to the fact that the coordination with FogFlow-Discovery is only needed for subscriptions and initial updates to decide which stream should be provided to which subscribers. After that, the workload triggered by frequent value updates can be easily handled by brokers in parallel. Hence, by separating broker and discovery components, FogFlow is able to achieve scalability of forwarding context data between publishers and subscribers.

VI. RELATED WORK

There have been many studies related to the IoT smart city platforms. However, most of the efforts focus only on the cloud environment. For example, as an open software platform, FIWARE is helping service providers to quickly and cost-effectively build their cloud-based applications and services by providing various open-source generic enablers (GEs). Nevertheless, none of the GEs offered by FIWARE enable flexible fog computing. In the FIWARE community, Orion Context Broker has been extensively used to enable the interoperability between different GEs, whereas Orion provides centralized context management. This is not a scalable solution considering large scale scenarios; in particular, when we consider exchanging real-time context information at edges or across various domains.

Most of the existing programming models focus on supporting batch and real-time data processing efficiently in a cluster or cloud environment. For instance, MapReduce has become the de facto standard for batch data processing in Apache Hadoop framework. Apache Spark is a distributed batch processing framework, while it also supports stream processing based on micro-batching. Other frameworks involve Apache Storm which supports event-based stream processing and Apache Flink which enables both batch and stream processing with its unified APIs. Recently, due to the requirement of having a unified programming model for both batch processing and stream processing, the generic dataflow programming model is mostly preferred over MapReduce to support cloud-based data processing in many new frameworks, such as Apache Beam, MillWheel, and Google Cloud Dataflow. All of these frameworks are only tailored to the cloud environment and they are unsuitable for fog computing due to their limited considerations on the heterogeneity, geo-distribution, openness, and interoperability requirements of future fog computing infrastructures.

In 2015 Cisco formed the OpenFog Consortium [13] together with partners from industry and academia, trying to accelerate the adoption of open fog computing in various domains. OpenFog Consortium emphasize the importance of openness and interoperability of fog computing infrastructure in their blueprint architecture document, while they do not provide any concrete proposal to achieve these two goals. The existing studies on fog computing mainly focus on optimization of resource and task allocations. For instance, Foglets [4] and MCEP [14] support live task migrations with their own APIs in a cloud-edge environment for location-aware applications. Mobilefog [15] provides a programming model for fog computing applications based on its own APIs. FogHorn is a commercial edge computing infrastructure with the focus on complex event processing at edge devices. Different from those existing fog computing frameworks, FogFlow is not designed to invent a completely new programming model for fog computing with private APIs. FogFlow, on the other hand, extends cloud-based dataflow programming model with standard-based APIs and make it suitable for the cloud-edge environment. In this way, our programming model can be quickly adopted by cloud service providers to build their fog computing services without much learning effort.

VII. CONCLUSION

In this paper, we propose the FogFlow framework which provides a standards-based programming model for IoT

services for smart cities that run over cloud and edges. The FogFlow framework enables easy programming of elastic IoT services and it supports standard interfaces for contextual data transfers across services. We showcase three use cases and implement an example application for smart cities. Furthermore, we analyze the performance of context management using NGSI interfaces to see feasibility of our standard-based approach in the smart city scale.

As a future work, we plan to develop algorithms that supports mobility-aware optimization for edge computing. Moreover, we intend to improve FogFlow to provide fault tolerance in extreme conditions, such as natural disasters. Lastly, for external integration with IoT systems, we plan to develop a semantic mediation gateway which converts context information on the fly from various information models to NGSI.

REFERENCES

[1] (2017). *CityPulse*. [Online]. Available: http://www.ict-citypulse.eu

[2] 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 *Proc. IEEE Big Data Congr.*, New York, NY, USA, Jun. 2015, pp. 592–599.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. 1st Ed. MCC Workshop Mobile Cloud Comput.*, Helsinki, Finland, 2012, pp. 13–16.

[4] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, "Incremental deployment and migration of geo-distributed situation awareness applications in the fog," in *Proc. 10th ACM Int. Conf. Distrib. Event Based Syst.*, Irvine, CA, USA, 2016, pp. 258–269.

[5] (2017). *NGSI 9/10 Information Model*. [Online]. Available: http://www.openmobilealliance.org/release/NGSI/

[6] M. Bauer *et al.*, "The context API in the OMA next generation service interface," in *Proc. 14th Int. Conf. Intell. Next Gener. Netw.*, Berlin, Germany, Oct. 2010, pp. 1–5.

[7] (2017). *Fiware*. [Online]. Available: https://www.fiware.org/

[8] (2017). *Wise-IoT*. [Online]. Available: http://wise-iot.eu/en/home/

[9] (2017). *Open & Agile Smart Cities*. [Online]. Available: http://www.oascities.org/open-agile-smart-cities/

[10] (2017). *Aeron Broker*. [Online]. Available: https://github.com/Aeronbroker/Aeron

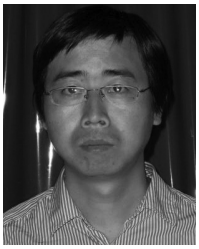
[11] (2017). *Orion Broker*. [Online]. Available: https://fiware-orion.readthedocs.io/

[12] B. Cheng, A. Papageorgiou, and M. Bauer, "Geelytics: Enabling on-demand edge analytics over scoped data sources," in *Proc. IEEE Int. Congr. Big Data*, Jun. 2016, pp. 101–108.

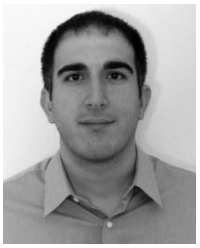
[13] (2017). *OpenFog Consortium*. [Online]. Available: https://www.openfogconsortium.org/

[14] B. Ottenwälder *et al.*, "MCEP: A mobility-aware complex event processing system," *ACM Trans. Internet Technol.*, vol. 14, no. 1, pp. 1–24, Aug. 2014.

[15] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the Internet of Things," in *Proc. 2nd ACM SIGCOMM Workshop Mobile Cloud Comput. (MCC)*, Hong Kong, 2013, pp. 15–20.



Bin Cheng is a Senior Researcher with NEC Laboratories Europe, Heidelberg, Germany. His current research interests include edge computing, big data analytics, and serverless computing for Internet of Things.



Gürkan Solmaz is a Research Scientist with NEC Laboratories Europe, Heidelberg, Germany. His current research interests include Internet of Things, human mobility, wireless ad hoc and sensor networks, and smart cities.

Flavio Cirillo is a Research Scientist with the Cloud Services and Smart Things Group, NEC Laboratories Europe, Heidelberg, Germany. His current research interests include Internet of Things analytics and platforms, especially scalability and federation aspects and semantics enablement.

Ernö Kovacs is a Senior Manager for the Cloud Services and Smart Things Group, NEC Laboratories Europe, Heidelberg, Germany. His group works on cloud computing, Internet of Things (IoT) data analytics, edge computing, and context aware services. He is currently contributing to the FIWARE, FIESTA-IoT, and AUTOPILOT projects.

Kazuyuki Terasawa is a Chief Engineer with the NEC Corporation, Tokyo, Japan. He is leading a team to provide smart city solutions in Japan. His current research interests include federated data management and data sharing platforms across multiple business domains.

Atsushi Kitazawa is a Chief Engineer with NEC Solution Innovators, Ltd., Tokyo, Japan, where he is leading a division in charge of big data and extending its reach to Internet of Things and edge computing.