

# *FogFlow* - computation organization for heterogeneous Fog computing environments

Joanna Sendorek<sup>1</sup>, Tomasz Szydło<sup>1</sup>, Mateusz Windak<sup>1</sup>, and Robert Brzoza-Woch<sup>1</sup>

AGH University of Science and Technology,  
Department of Computer Science, Krakow, Poland

**Abstract.** With the arising amounts of devices and data that Internet of Things systems are processing nowadays, solutions for computational applications are in high demand. Many concepts targeting more efficient data processing are arising and among them edge and fog computing are the ones gaining significant interest since they reduce cloud load. In consequence Internet of Things systems are becoming more and more diverse in terms of architecture. In this paper we present *FogFlow* - model and execution environment allowing for organization of data-flow applications to be run on the heterogeneous environments. We propose unified interface for data-flow creation, graph model and we evaluate our concept in the use case of production line model that mimic real-world factory scenario.

**Keywords:** IoT, fog computing, stream processing, data-flow graphs

## 1 Introduction

Internet of Things (IoT) is becoming more and more present in our reality with the development of intelligent buildings, smart cities[18] or mobile communications[2]. We are being surrounded by increasing number of electronic devices including our everyday objects, like watches or mobiles, and whole systems managing our environment, such as traffic monitoring or intelligent surveillance. Majority of those devices is able to be network-connected and this number is still growing with hardware advancements causing increasing capabilities of simplest appliances. This can be observed especially in the sensors area which gives foundations for sensor networks solutions and architectures[1]. All of those devices are potential sources of data which may be shared or gathered thanks to the network access.

Constantly growing amount of real-time data has been causing a shift towards data-orientation in the systems architectures as well as programming paradigms over the last few years[9]. Even with computing clouds becoming recently more developed and mature, processing data we have access to is still posing challenges. Although cloud solutions provide a way to achieve capabilities required to handle significant amounts of data, they bring in response delays, coming from

communication overhead, which are often unacceptable in the real-time applications. Therefore, current research in IoT domain is often focused on finding a balance between responsiveness and robustness, resulting for instance in edge computing[8] or fog computing[3] concepts. Mentioned ideas assume that some of the computations can be moved closer to the data sources and in this scenario cloud infrastructure may be responsible only for heavy analytic and global information handling. Also, it is beneficial to move computations responsible for local decision making closer to the devices as it reduces response time. In order to fully make use of available resources, cloud, edge and fog computing should be treated as complimentary ones since each of them is most advantageous in different aspect of whole system[10].

In the area of cloud and data processing a lot have been done recently with multiple mature analytic solutions such as *Apache Spark*<sup>1</sup>, *Google Cloud Dataflow*<sup>2</sup> or *Apache Flink*<sup>3</sup>. However, since cloud and fog should be inter-operating as mentioned, it is desirable to have a generic development model for data-flow applications enabling to span fog and cloud[4]. Such a concept, but limited only to multiple cloud solutions, is realized by *Apache Beam*<sup>4</sup> - unified model which enable defining data streams and running them on different back-ends. In this paper, we take the idea of unification further and propose *FogFlow* - the model and engine enabling defining data processing pipelines able to be decomposed into the sub-pipelines that span fog or edge and the cloud. We base our concept on graph model, commonly used in data-flow applications[16], [13]. Scientific contribution of this paper can be summarized as follows: (i) we define unified graph-based model for abstract definition of data processing applications; (ii) we discuss methods of decomposing and translating the graph model into the set of processing nodes appropriate for given infrastructure; (iii) we propose methodology and provide example of design and implementation data-processing application able to be run in heterogeneous IoT environments.

Organization of the paper is as follows. Section 2 describes the related work and section 3 discusses *FogFlow* - its components, model and implementation. Section 4 describes the evaluation, while section 5 concludes the paper.

## 2 Related work

To our best knowledge *FogFlow*, as the model aiming at unifying data processing applications definition and enabling their execution in heterogeneous environments, is a novel solution bridging the gap of existing concepts. However, while developing *FogFlow* we relied on current research in the area of organizing modern applications centred around IoT data processing.

Since many of nowadays systems are focused on data processing, the data-flow paradigm is adopted altogether with IoT systems in many current works

<sup>1</sup> [spark.apache.org](https://spark.apache.org) access for 15.03.19

<sup>2</sup> [cloud.google.com/dataflow/](https://cloud.google.com/dataflow/) access for 15.03.19

<sup>3</sup> [flink.apache.org](https://flink.apache.org) access for 15.03.19

<sup>4</sup> <https://beam.apache.org/get-started/beam-overview/> access for 02.02.2019

and research. For example, in [7] authors describe scalable IoT framework to design logical data-flow using virtual sensors. In this solution, operators for data processing are defined and then processed in the logical data-flow. The whole modelling is based on the graph that is being executed in the proper topological order. Data-flow architecture is also commonly used for application design - in [5] author proposes data-flow architecture for smart city applications. In this solution operations can manipulate data coming from different flows.

The necessity for designing and managing strategies for IoT services placement dedicated for IoT and Fog computing has been acknowledged lately as the natural consequence of heterogeneity in the IoT environments. Since components of both IoT and Fog system infrastructures are often highly distributed and they vary in the context of resources availability and computing capabilities, efficient resource management strategies have to be implemented in order to maximize whole system response while minimizing overall solution cost and latency. *IFogSim* [6] is a simulator for IoT and fog environments which enable assessment of system efficiency due to the chosen metrics, such as energy consumption, network latency and operational cost. *IFogSim* allows for system modelling and evaluation of system focusing on resource management techniques. It is worth noting that *IFogSim* uses graph model as the system representation and makes usage of typical graph algorithm, Floyd-Warshall for all shortest path problem, in order to carry out data transmission simulation. In very recent work [11] authors present an extension to iFogSim allowing for the design of data placement strategies based on 'divide and conquer' approach. There are other methods such as [12], where authors propose a strategy of data placement in fog infrastructure and it is based on graph partitioning. They use graph as the model of infrastructure with edges standing for data-flows numbers passing between the nodes.

The concept similar to the one presented in the paper is the Distributed Dataflow (DDF) - the programming model described in [4] which aims at being used with infrastructures of both fog and cloud. DDF framework is based on the *Node-RED*<sup>5</sup> tool, which allows for creating high-level application definition constructed as a graph. In a similar way to *FogFlow* application parts can be placed on different elements of the infrastructure, but the same execution environment (*NodeRed*) is required on all of the devices. In contrast to that, *FogFlow* enables multiple execution environment and this is achieved, among all, via source code generation.

### 3 *FogFlow* structure

We propose the *FogFlow* model and execution environment, which enables the design of applications able to be decomposed onto heterogeneous IoT environments according to the chosen decomposition schema. We achieve this flexibility of running applications in the multiple environments by abstracting out the

<sup>5</sup> <https://nodered.org/>

application definition from its architecture and the target implementation. In order to provide one unambiguous, well-defined model of computations, we rely on graph representation. We aim at fulfilling the following requirements for data-driven IoT applications:

1. the application definition should be infrastructure-independent and contain only logic of data processing;
2. execution of the application should be possible on different set of devices.

Those requirements are able to be fulfilled with the assumption that each data-processing step is stateless and all information required for further processing is contained in the messages passed with data. Independence between application definition and its execution is achieved by three-layered application model corresponding with three-layered module architecture. Fig 1 depicts conceptual diagram of *FogFlow* - both its architecture and application model. First and most high-level layer of *FogFlow* is *FogFlow API*, which enables creating application definition. Our API is organized in the functional style, enabling for creation of data processing pipeline reflecting successive steps. Definition created is then being **transformed** into data-flow graph representation managed by the next layer - graph module. Core function of using graphs as the intermediate application state is to provide one, unambiguous model which can be used in the further processing. In the module discussed the data-flow graphs are fetched into the form appropriate for the **modification**. This term includes both adjustments made to remove ambiguities introduced by user in the application definition and those aiming at decomposing graph as preparation for multiple devices execution. The last phase is **translation** happening in the execution module which results in executables ready to be run on the provided devices. The same data-flow definition is able to be run in the chosen environment via appropriate **modifications** in the graph module and **translation** preparing graphs to be executed.

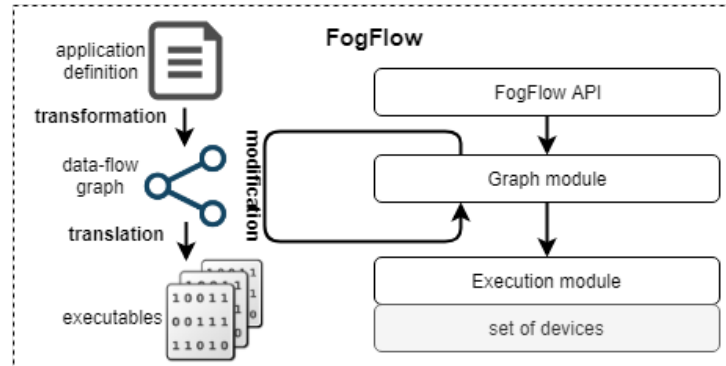


Fig. 1: Conceptual diagram of *FogFlow*

### 3.1 FogFlow API

*FogFlow API* is the part of *FogFlow* enabling for creation of data processing pipelines in the functional style. There are two main entities defined in the API:

1. **Flow** - representing the whole data processing pipeline;
2. **StreamData** - representing data stream containing data of the specified type at the given state of pipeline.

We decided upon using *Java 8* for *FogFlow API* implementation. It enables type-checking at compile time and ensuring that consecutive data processing steps are valid. Table 1 presents list of methods defined for each of the entities mentioned.

Table 1: *FogFlow API methods.*

StreamData<T>		Flow	
type	name	type	name
StreamData<U>	<i>map</i>	void	<i>setUpEnvironment</i>
StreamData<U>	<i>aggregateOnCount-SlidingWindow</i>	void	<i>executeFlow</i>
StreamData<U>	<i>aggregateOnCount-BatchWindow</i>	StreamData<T>	<i>createStreamEntry</i>
List<StreamData<T>>	<i>splitStream</i>	StreamData<T>	<i>createAsStreamsJoin</i>
StreamData<T>	<i>filter</i>		
void	<i>sink</i>		

### 3.2 Data-flow graph representation





In the *FogFlow*, data-flow is represented as a graph, where nodes correspond to processing functions while the vertices represent the flow of the messages between functions. The aforementioned *FogFlow API* is used to programmatically construct the data-flow graph representation. Based on the number of incoming and out-coming edges, the following types of graph nodes can be described:




- *One-to-one* nodes - those are nodes corresponding with processing functions able to modify particular data stream;
- *One-to-many* nodes, also called **split nodes** - such nodes can execute splitting the incoming data stream into a few, based on the defined condition;
- *Many-to-one* nodes, also called **join nodes** - those are able to reverse action of split by merging few data streams into one.
- *Zero-to-one* nodes, also called **sources** - entries for data streams;
- *One-to-zero* nodes, also called **sinks** - nodes representing termination of data processing.

The described differentiation requires additional comment regarding split and join nodes. First of all, multiple out-coming edges may occur implicitly when user calls multiple one-to-one processing functions on particular stream therefore creating multiple one-to-one nodes. The difference between this situation and application of split node is that in the first case, all created nodes are assigned **same** copy of the stream as the incoming edge while the usage of split node is inevitably related with creating **different data streams** being substream of the initial one. Second issue worth noting is that **join nodes** are the only ones that enable multiple incoming edges and in consequence those are the nodes responsible for the whole graph being **directed acyclic graph** and not a **tree** in particular. From the perspective of computation results equivalence, joining streams and next creating *one-to-one* node corresponds to multiple *one-to-one* nodes being created for each of join terms. However, such a solution would be both less intuitive for user and less effective to evaluate.

Taking into account, the specific data processing that the node can carry out, we propose the following types of intermediate vertices gathered in table 2.

Table 2: Types of graph nodes and their representation

<i>Node</i>	<i>In-edges</i>	<i>Out-edges</i>	<i>Symbol</i>
<b>split</b>	1	mutiple	
<b>join</b>	multiple	1	
<b>source</b>	0	1	
<b>sink</b>	1	0	

<i>Node</i>	<i>In-edges</i>	<i>Out-edges</i>	<i>Symbol</i>
<b>window</b>	1	1	
<b>map</b>	1	1	
<b>filter</b>	1	1	

### 3.3 Graph modifications

As discussed previously, the choice of representing consecutive data-flow transformations via graphs has been driven, among others, by the convenience of adapting the whole graph analysis and algorithms to structure computations efficiently. Treating each transformation of data as a separate graph node allows for placing them on different devices according to the acquired policy. Moreover, graph structure allow for straightforward simplifications with tree-like balancing, pruning or path reductions.

In this section, we present two examples illustrating decomposition onto two devices and simple pruning. Figure 2 depicts fragment of data-flow consisting initially of four nodes: *map* and *split* node being source for *filter* and *data sink*. The most basic pruning algorithm used in *FogFlow* is described by the pseudocode 1 and it removes all of the branches from the graph that are not finished by a sink node. In the example, the upper-side branch of split is unused since it ends in the filter node. Therefore it is removed at this state of modification.

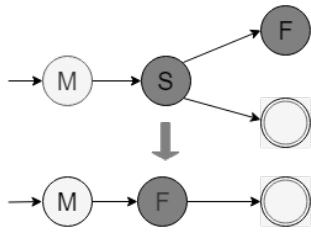


Fig. 2: Graph pruning

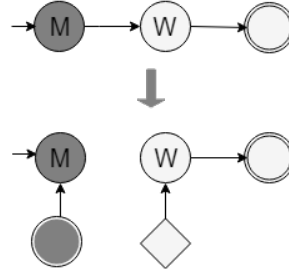


Fig. 3: Decomposition

Since the *sink node* on the bottom side of *split* has been assigned only part of the data and now this is the only successor of the *split*, in order to maintain unambiguous description, the *split node* should be now converted to *filter node*. The graph after the pruning is equivalent to the one before in the sense that it will give the same data transformation results. Nodes not participating directly in the pruning (coloured in white) remain unchanged. The question may arise when pruning will be beneficial in the practical sense. The issue with an API styled in the functional way is that even though it is intuitive to use, it lacks mechanisms imposing ambiguity. On the contrary, the graph description allows for detecting any redundancies and reducing graph to the concise form. Apart from concept illustrated by the example, other possible ways for graph data-flow simplification would be: analyses of parallel edges, avoiding duplicating vertices or path reduction. All of the mentioned modifications are valid under the assumption that nodes are stateless as mentioned earlier.

---

**Algorithm 1** Tree pruning algorithm.
 

---

```

1: function PRUNE( $G$ )
2:    $usedNodes \leftarrow \emptyset$ 
3:   for  $graphNode$  in  $G.V$  do
4:     if  $graphNode$  is sink then
5:        $usedNodes \leftarrow usedNodes \cup graphNode$ 
6:   for  $sinkNode$  in  $usedNodes$  do
7:      $usedNodes \leftarrow usedNodes \cup ancestors\ of\ sinkNode$ 
8:    $unusedVertices \leftarrow G.V \setminus usedNodes$ 
9:   return  $unusedVertices$ 

```

---

Figure 3 illustrates decomposition of the graph fragment onto two devices. Flow fragment consists of three nodes being structured as one pipeline. Additionally, each of the nodes has been assigned to the one the two available devices - dark nodes to one of the devices (let it be called device A) and white nodes to the other (device B). The choice of exact algorithm or policy used for node

tagging and device assignment is the whole broad topic and out of scope of this paper, but there are a few approaches we consider:

- one of the algorithms described in 2 may be adopted;
- dynamic system analysis may be conducted in order to determine the devices usage and balancing may be implemented;
- heuristic aiming at moving majority of processing as close to data source as possible at each processing phase may be used - we discussed it in [14].

Regardless of the approach taken, in the example the nodes were tagged in a way resulting in the *window* and succeeding *sink* node being to be moved onto device B, while assigning all of the rest to the device A. The decomposition onto two devices in the example is crossing one edge - the one connecting *map* with the *window* node. Algorithm 2 describing the process of decomposition on the given edge. Since the effect of decomposition should be two sub-graphs - representing two independent data-flows able to be run on separated devices, this edge is being removed. In order to maintain valid graph structure and provide a way to pass data from device A to device B, two new nodes are inserted into graph: sink node on device A and corresponding source node on device B. The exact implementation of those so-called *communication nodes* depends on the devices and the established protocol, but it does not affect the decomposition itself.

---

**Algorithm 2** Decomposition on edge algorithm.

---

```

1: function DECOMPOSE( $G, edge, deviceA, deviceB$ )
2:    $sourceVertex \leftarrow edge.source$ 
3:    $targetVertex \leftarrow edge.target$ 
4:    $newSourceVertex \leftarrow newSource(deviceB)$ 
5:    $newSinkVertex \leftarrow newSink(deviceA)$ 
6:    $G.E \leftarrow G.E \setminus edge$ 
7:    $G.V \leftarrow G.V \cup \{newSourceVertex, newSinkVertex\}$ 
8:    $G.E \leftarrow G.E \cup \{E(sourceVertex, newSinkVertex),$ 
       $E(targetVertex, newSourceVertex)\}$ 

```

---

### 3.4 Execution module and implementation

Execution module is responsible for translating data-flow graphs into executables ready for running on the given devices set. Up to this point, we have used term 'devices set' to describe multiple possible infrastructures that may be existent in the IoT system. However, in order to translate data-flow graphs into executables for heterogeneous target environments, we distinguish three types of infrastructure summarized in the table 3. At current state of work, *FogFlow* provides execution with one runtime per type - further development regarding this area is described in 5. While implementation details are out of scope for this



Table 3: Types of infrastructure distinguished for *FogFlow*

Infrastructure	Description		Runtime
<i>cloud</i>	<b>features</b>	- focus on high-availability - scalability, - possible contenerization	<i>Apache Flink</i>
	<b>use cases</b>	- running resource-demanding algorithms - correlating data from different sources - data persistence	
<i>edge</i>	<b>features</b>	- ability to run local analytics, - limited resources and small footprint, - having application processors	<i>Apache Edgent</i>
	<b>use cases</b>	- reducing data sent for further analytics - local decision making and classification - immediate events reaction	
<i>MCUs</i>	<b>features</b>	- resource constrained embedded microcontrollers, - impossibility of porting high-level libraries	<i>C based</i>
	<b>use cases</b>	- acquiring sensor-data - basic preprocessing and reformatting	

paper in which we focus on concept and methodology, we will briefly describe execution module structure.

Cloud environments are the ones that are characterized by resource capabilities and global knowledge of the whole environment. Therefore, application parts best suited to be run in the cloud are the ones that correlate data from multiple sources or are highly demanding. In *FogFlow* we currently provide support for execution with *Apache Flink* framework to run such application components. We chose *Flink* because of its efficiency (in-memory speed) and interoperability with majority of commonly used cluster environments. On the contrary to cloud, edge devices lack resources required to run distributed computations, but are able to execute local analysis. We distinguish this group of devices by presence of application processors allowing for high-level programs execution. Paramount example of edge devices would be gateways passing data from lower-level devices up to the cloud. They can be used to process and filter data streams and conduct non complex local analysis leading to rapid system reaction. For edge, we decided upon using *Apache Edgent*<sup>6</sup> runtime to execute data-flow graphs. The lowest-level of infrastructure consists of MCU's - resources constrained embedded microcontrollers. They lack ability to port any high-level libraries and using two Apache technologies mentioned would be impossible. Data-flow parts placed on such devices could contain data-gathering logic and basic preprocessing such as changing format or cutting off unnecessary information. Data-flow may be able to be executed with C based runtime.

Execution module is divided into submodules (implementation providers) corresponding with the three types of runtime. Each submodule contains code

<sup>6</sup> [edgent.apache.org](http://edgent.apache.org) access for 15.03.2019

that translates data-flow graph into the given target executable. When translating, we rely on the source code generation approach giving promising results when small footprint processing running on the edge is concerned [15]. Both *Flink* and *Edgent* rely on the slightly different graph models, are based on Java programming language and have application programming interfaces allowing for basic functional-style operations on data streams. With *FogFlow* it is possible to translate data-flow graphs created with *FogFlow API* into their source code in following steps:

1. data-flow graph and chosen implementation provider are passed to the executing module;
2. data-flow graph is being translated into source code in the topological order using provider implementation of *FogFlow API*
3. all of the external objects such as custom sources or sinks are serialized and attached to the code;
4. proper java archive file is created including only dependencies of *FogFlow API* and chosen implementation provider.

Translation targeting C code is far more complex and lower-level issue, with many aspects needed to be taken into account: inter-language typing system, objects serialization between technologies, constraints of C language or threads execution in C. We do not discuss this broad and hardware related topic in the scope of this paper and our use case does not include data-flow graph execution on MCU's.

## 4 Use case and evaluation

In this section we present an example usage of *FogFlow* to organize computations for vibrations detection of the working assembly line in our laboratory. Schema of our environment has been depicted in figure 4. The goal of the presented use case is to measure vibrations generated by the working assembly line. The *micro-machined microelectromechanical system (MEMS)* accelerometer is gathering three dimensional data allowing for vibration measurement and it is read by connected embedded device. For each 128 samples gathered in the batch windows we calculate *Root Mean Square (RMS)*[17] defined as  $\sqrt{\frac{\sum_{i=j-\gamma}^j x_i^2}{\gamma}}$ , where  $\gamma$  states for the window size and in the case described equals to 128. In the local network we also have *Raspberry Pi* gateway. Another component of environment is the private cloud with database accessible from both the assembly line and gateway.

Results of the vibration analysis are then written to the database in the cloud for further analysis. Processing pipeline is defined as follows:

1. data is received from the device with accelerometer using MQTT protocol;
2. data is parsed - only the component aligned with x axis is being extracted from three dimensional data and interpreted as floating-point number;

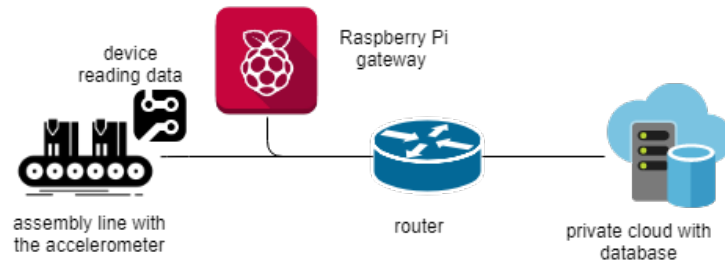


Fig. 4: Laboratory environment schema

3. data is gathered in 128-elements batch windows and for each window  $RMS$  is calculated;
4. results are written to the database.

Listing 1.1 presents application definition corresponding with the described pipeline created with *FogFlow API*.

```

flow . setUpEnvironment ();
StreamData<String> receivedMessage = flow .
    createStreamEntry (new MqttExternalReceiver ());
StreamData<Double> x = receivedMessage . map (new
    MessageParser ());
StreamData<Double> rms = x . aggregateOnCountBatchWindow
    (128, new RmsLambda ());
rms . sink (new DatabaseWriter ());
flow . executeFlow ();

```

Listing 1.1: Assembly line data processing pipeline in *FogFlow API*.

Created application definition is then **transformed** into data-flow graph. At this point graph nodes are assigned to the particular devices in our laboratory and then graph is decomposed accordingly. Given the infrastructure described, we decompose data processing application with the two schemas:

1. cloud-based data processing - where all of the data from the accelerometer is received in the cloud and processed there;
2. cloud database operations with edge computing pre-processing - where all of the data is received on gateway, pre-processed there and send to cloud for further steps.

In the scenario 1, the whole pipeline is aimed at executing with *Apache Flink* and in scenario 2, the underlying data-flow graph is decomposed into two sub-graphs. This process is illustrated in figure 5. With cloud based data processing, *Apache Flink* is used for the whole execution. One thing requires discussion at this point - the impact of possible number of production lines on the processing

efficiency. In such a simple scenario, we propose creating separate pipeline for each of the lines. Therefore, execution of all of them in one environment, such as *Apache Flink*, is an embarrassingly parallel problem with only potential bottleneck being database writing. Since cloud allows for high distribution of computations, significant processing capabilities may be achieved. However, major drawback of this scenario is that of all the data from accelerometer is sent to the cloud generating great amount of network traffic leading to unnecessary delays and greater costs.

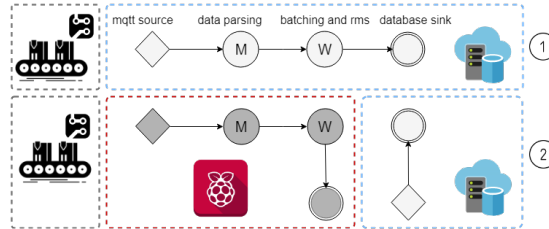


Fig. 5: Data-flow graph decomposition with two scenarios.

We measured data traffic which is being processed at different stages of data-flow. First of all, accelerometer is sending data as fast as it gets consecutive measurements, which gives approximately *950 messages per second (msg/s)* which with the MQTT protocol for communication gives a traffic of *149 596 B/s*. After batch window reduction is applied this is reduced to approximately *584 B/s*. In the second scenario, reduction of data in the batch windows is conducted by *Raspberry Pi gateway with the Apache Edgent*.

Table 4: Processing capabilities for *Apache Edgent*

	simple data source	MQTT data source
<i>Intel Core i5 2 cores</i>	462 929 $\frac{msg}{s}$	41523 $\frac{msg}{s}$
<i>Raspberry Pi</i>	20 145 $\frac{msg}{s}$	9861 $\frac{msg}{s}$

We conducted artificial tests of *Apache Edgent* processing capability where we replaced MQTT data source with the data spawner. In table 4 we gathered results obtained both with MQTT source and spawner in different setups. It is worth noting that even when running on *Raspberry Pi* it can process *9861 msg/s* with the MQTT client as the data source. That means that in this particular setup the amount of data being processed is limited only by sending speed of the device with *MEMS accelerometer*, which is determined by communication protocol. It demonstrates that even devices with relatively constrained resources are able to significantly reduce amount of data processed in the cloud. *FogFlow* enables their seamless integration with whole IoT system infrastructure.

## 5 Summary and future work

In this paper we have presented *FogFlow* which allows for defining data processing applications able to be run with different technologies with the unified interface. We have described concept of modelling such applications and decomposing them onto given set of devices. At current state of work we have focused on designing *FogFlow* and developing end-to-end process for applications design and execution. Among the goals that are our current priority there is extending existing implementation of execution module with C-based one. We plan also to create toolkit of lambdas and data providers/sinks in order to facilitate *FogFlow API* usage. At the moment we are also working on developing more sophisticated ways for graph modifications and decomposition.

## Acknowledgment

The research presented in this paper was supported by the National Centre for Research and Development (NCBiR) under Grant No. LIDER/15/0144 /L-7/15/NCBR/2016.

## References

1. Akyildiz, I.F., , Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. *IEEE Communications Magazine* **40**(8), 102–114 (Aug 2002). <https://doi.org/10.1109/MCOM.2002.1024422>
2. Aloï, G., Caliciuri, G., Fortino, G., Gravina, R., Pace, P., Russo, W., Savaglio, C.: Enabling iot interoperability through opportunistic smartphone-based mobile gateways. *Journal of Network and Computer Applications* **81**, 74 – 84 (2017). <https://doi.org/https://doi.org/10.1016/j.jnca.2016.10.013>, <http://www.sciencedirect.com/science/article/pii/S1084804516302405>
3. Bellavista, P., Berrocal, J., Corradi, A., Das, S.K., Foschini, L., Zanni, A.: A survey on fog computing for the internet of things. *Pervasive and Mobile Computing* **52**, 71 – 99 (2019). <https://doi.org/https://doi.org/10.1016/j.pmcj.2018.12.007>, <http://www.sciencedirect.com/science/article/pii/S1574119218301111>
4. Giang, N.K., Blackstock, M., Lea, R., Leung, V.C.M.: Developing IoT applications in the Fog: A Distributed Dataflow approach. In: 2015 5th International Conference on the Internet of Things (IOT). pp. 155–162 (Oct 2015). <https://doi.org/10.1109/IOT.2015.7356560>
5. Grgoire, J.: A data flow architecture for smart city applications. In: 2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN). pp. 1–5 (Feb 2018). <https://doi.org/10.1109/ICIN.2018.8401639>
6. Gupta, H., Vahid Dastjerdi, A., Ghosh, S.K., Buyya, R.: ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience* **47**(9), 1275–1296 (2017). <https://doi.org/10.1002/spe.2509>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509>

7. Kim-Hung, L., Datta, S.K., Bonnet, C., Hamon, F., Boudonne, A.: A scalable iot framework to design logical data flow using virtual sensor. In: 2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob). pp. 1–7 (Oct 2017). <https://doi.org/10.1109/WiMOB.2017.8115775>
8. Mao, Y., You, C., Zhang, J., Huang, K., Letaief, K.B.: A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys Tutorials* **19**(4), 2322–2358 (Fourthquarter 2017). <https://doi.org/10.1109/COMST.2017.2745201>
9. Milutinovic, V., Kotlar, M., Stojanovic, M., Dundic, I., Trifunovic, N., Babovic, Z.: DataFlow Systems: From Their Origins to Future Applications in Data Analytics, Deep Learning, and the Internet of Things, pp. 127–148. Springer International Publishing, Cham (2017). [https://doi.org/10.1007/978-3-319-66125-4\\_5](https://doi.org/10.1007/978-3-319-66125-4_5), [https://doi.org/10.1007/978-3-319-66125-4\\_5](https://doi.org/10.1007/978-3-319-66125-4_5)
10. Munir, A., Kansakar, P., Khan, S.U.: Ifciot: Integrated fog cloud iot: A novel architectural paradigm for the future internet of things. *IEEE Consumer Electronics Magazine* **6**(3), 74–82 (July 2017). <https://doi.org/10.1109/MCE.2017.2684981>
11. Naas, M.I., Boukhobza, J., Parvedy, P.R., Lemarchand, L.: An extension to ifogsim to enable the design of data placement strategies. In: 2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC). pp. 1–8 (May 2018). <https://doi.org/10.1109/CFEC.2018.8358724>
12. NAAS, M.I., Lemarchand, L., Boukhobza, J., Raipin, P.: A graph partitioning-based heuristic for runtime iot data placement strategies in a fog infrastructure. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 767–774. SAC '18, ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3167132.3167217>, <http://doi.acm.org/10.1145/3167132.3167217>
13. Sena, A.C., Vaz, E.S., Frana, F.M.G., Marzulo, L.A.J., Alves, T.A.O.: Graph templates for dataflow programming. In: 2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW). pp. 91–96 (Oct 2015). <https://doi.org/10.1109/SBAC-PADW.2015.20>
14. Szydło, T., Brzoza-Wońc, R., Sendorek, J., Windak, M., Gniady, C.: Flow-based programming for iot leveraging fog computing. In: 2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). pp. 74–79 (June 2017). <https://doi.org/10.1109/WETICE.2017.17>
15. Szydło, T., Sendorek, J., Brzoza-Wońc, R.: Enabling machine learning on resource constrained devices by source code generation of the learned models. In: Shi, Y., Fu, H., Tian, Y., Krzhizhanovskaya, V.V., Lees, M.H., Dongarra, J., Sloot, P.M.A. (eds.) *Computational Science – ICCS 2018*. pp. 682–694. Springer International Publishing, Cham (2018)
16. Teranishi, Y., Kimata, T., Yamanaka, H., Kawai, E., Harai, H.: Dynamic data flow processing in edge computing environments. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC). vol. 1, pp. 935–944 (July 2017). <https://doi.org/10.1109/COMPSAC.2017.113>
17. Trkay, S., Akay, H.: A study of random vibration characteristics of the quarter-car model. *Journal of Sound and Vibration* **282**(1), 111 – 124 (2005). <https://doi.org/https://doi.org/10.1016/j.jsv.2004.02.049>, <http://www.sciencedirect.com/science/article/pii/S0022460X04002974>
18. Zanella, A., Bui, N., Castellani, A., Vangelista, L., Zorzi, M.: Internet of things for smart cities. *IEEE Internet of Things Journal* **1**(1), 22–32 (Feb 2014). <https://doi.org/10.1109/JIOT.2014.2306328>