Prevent overcrowding in exhibitions using FogFlow to distribute people

Lorenzo Siega Battel
Politecnico di Milano
lorenzo.siega@mail.polimi.it

***Abstract*—The organizers of an exhibition with several booths are trying to avoid overcrowded booths so visitors could be distributed smoothly and enjoy the exhibition. They mount sensors at each booth so to know the number of people currently at each booth. Each booth has an electrical board which, based on the information from other booths their distance, suggests visitors the closest booth to visit next in a way that the population is smoothly distributed. The overall number of visitors of each booth is reported to the cloud for information aggregation to be queried for the crowded time of the day.**

I. Introduction

During an exhibition people can stop in front of a booth and create congestion. By providing them informations about the next more suitable booth to visit, we can distribute them more efficiently and so avoid overcrowding.
People at each booth are counted using cameras or some other types of sensors and then an algorithm suggest, for each both, a booth to visit by displaying it on an "electrical board".
In this document we will address the problem by leaving the choice of the sensor and the algorithm to the final people who finally implement the system.
During the development of such a system two main problems will have to be addressed:

- Required analysis effort: As we have to analyze a stream of data from a large number of sensors, we can't compute it on a centralized component.
- Algorithm: Algorithm works on the entire set of booths and triggering it on each update received from a booth, as FaaS computing does, will generate an high computation and data load.

FogFlow can address to both problems, by filling these gaps:

G2 Function triggering: from per event to per selected entities: In existing serverless computing frameworks, functions are invoked per event with limited execution time and memory size. This is not suitable for data-intensive IoT services.[...]

G3 Function execution: from data $\rightarrow$ code or code $\rightarrow$ data to code $\leftrightarrow$ data: Existing serverless computing frameworks separate data management from the function execution environment, always moving data into the execution environment for function execution (data to code pattern). On the other hand, existing fog computing frameworks such as Azure IoT Edge move cloud functions to the data located at the edges.[...]

G4 Function composition: from event-oriented or edge-oriented to data-centric: In existing serverless computing frameworks such as OpenWhisk, service developers need to customize a series of event triggers and rules to link multiple functions together. Existing fog computing frameworks allow service developers to link functions at each edge by manually configuring the topic-based data routing path between them.[...]

[1]

II. Context producers and context consumers

In FogFlow, as it is a data-centric framework, our first goal is to produce data, process it and then utilize that data to make decisions or let others do that.
In this document we will refer at data as context. In fact, in FogFlow, data is handled as a context and not as a single entity, so multiple instances of a single entity can create a context. For example you can think of multiple sensors that count people who are entering in a building, a single sensor can produce a context called "Ingoing at entrance 2-A" while all the sensors placed at the entrances of a building can produce a context called "People flow in building 2". FogFlow can handle this different types of context producing different other context updates based on where they will be "deployed". For example it can reduce the flow at a single entrance or cut it when a building is becoming full of people.

A. Context producer aka the sensor

We cannot know *a priori* the pyhisical structure of an exhibition so we do not provide any constraint on the type of the sensor. Sensor's data will be processed on site and then sent to the nearest broker adoptingits NGSI10 APIs.

B. Context consumer aka the electronic board

As for the sensor we cannot know how informations will be displayed so an intermediate device is needed to receive data from FogFlow and then show it.
It will expose an HTTP server which will be used by the broker to send context updates.

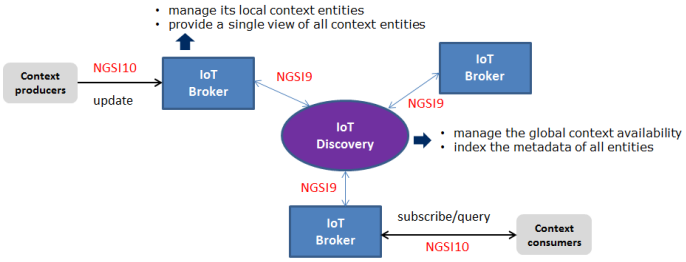

Fig. 1. Context prod/cons topology

III. Operator and Fog Function

An operator contains the Fog Function code and must be in the form of a docker image and available on Docker Hub. FogFlow enables serverless edge computing, meaning that developers can define and submit a so-called fog function and then the rest will be done by FogFlow automatically, including:

- Triggering the submitted fog function when its input data are available
- Deciding how many instances to be created on workers according to its defined granularity
- Deciding where to deploy the created instances

The instances in the above text refer to the task instances which run a processing logic within them and this processing logic is given by the operator.

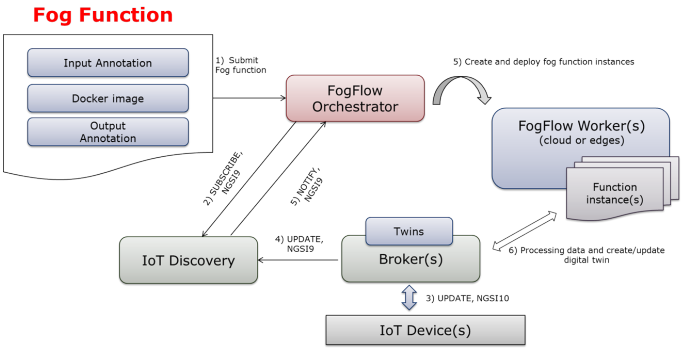

Fig. 2. FogFlow and Fog Functions

IV. Data visualization and querying

To better provide insights on how people move through the exposition and to allow the organizers to perform queries on the data, a scraping tool is mandatory. Prometheus is the best choice for two reasons:

- It allows users to perform queries on its data directly and not by using other tools
- It's widely supported by data visualization tools like Grafana

Perhaps the broker must be modified to expose a web service that matches the syntax accepted by Prometheus. Please notice that from now on we will see three values for each counter (count, male and female). These values are chosen by the developer (the broker exposes the values uploaded by the context producers) and so they are not fixed. Below an example of this syntax and in the next page the Go code developed.

```
PeopleCounter_1{name="count"} 56
PeopleCounter_1{name="male"} 37
PeopleCounter_1{name="female"} 19
PeopleCounter_10{name="count"} 45
PeopleCounter_10{name="male"} 37
PeopleCounter_10{name="female"} 8
PeopleCounter_20{name="count"} 17
PeopleCounter_20{name="male"} 14
PeopleCounter_20{name="female"} 3
```

A. Querying

To perform queries we must use a language called PromQL that lets the user select and aggregate time series data in real time. In the following examples, of PromQL, we assume there is a counter for each booth.

- Affluence for each booth:

```
{__name__=~"PeopleCounter_.*", name="count"}
```

- Total affluence, per gender:

```
sum({__name__=~"PeopleCounter_.*", name="male"})
sum({__name__=~"PeopleCounter_.*", name="female"})
```

- Affluence at a specific booth:

```
PeopleCounter_10{name="count"}
```

B. Visualization

To visualize data generated by the sensor, in some different graphs, and to provide an immediate point of view in a dashboard, Grafana is the best tool to use. In fact it allows us to use directly the data produced by the PromQL queries for build some interesting graphs.

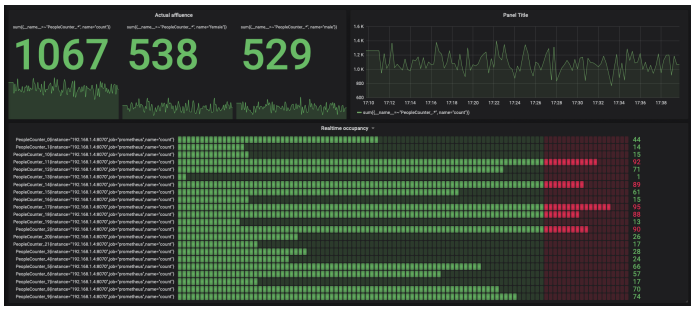

Fig. 3. A dashboard created using the queries in the example above

References

[1] Bin Cheng et al. *Fog Function: Serverless Fog Computing for Data Intensive IoT Services*. 2019. arXiv: 1907.08278 [cs.DC].

```
func (apisrv *RestApiSrv) getEntitiesProm(w rest.ResponseWriter, r *rest.Request) {
  for _, entity := range apisrv.broker.getEntities() {
    for _, attribute := range entity.Attributes {
      val, err := strconv.ParseFloat(fmt.Sprintf("%v", attribute.Value), 64)
      if err == nil {
        w.(http.ResponseWriter).Write([]byte(fmt.Sprintf("%s{name=\"%s\"} %v\n",
            strings.ReplaceAll(entity.Entity.ID, ".", "_"), attribute.Name, val)))
      }
    }
  }
}

func (apisrv *RestApiSrv) getEntityProm(w rest.ResponseWriter, r *rest.Request) {
  var eid = r.PathParam("eid")

  entity := apisrv.broker.getEntity(eid)
  if entity == nil {
    w.WriteHeader(404)
  } else {
    for _, attribute := range apisrv.broker.getEntity(eid).Attributes {
      val, err := strconv.ParseFloat(fmt.Sprintf("%v", attribute.Value), 64)
      if err == nil {
        w.(http.ResponseWriter).Write([]byte(fmt.Sprintf("%s{name=\"%s\"} %v\n",
            strings.ReplaceAll(entity.Entity.ID, ".", "_"), attribute.Name, val)))
      }
    }
  }
}

func (apisrv *RestApiSrv) getAttributeProm(w rest.ResponseWriter, r *rest.Request) {
  var eid = r.PathParam("eid")
  var attrname = r.PathParam("attr")

  attribute := apisrv.broker.getAttribute(eid, attrname)
  if attribute == nil {
    w.WriteHeader(404)
  } else {
    val, err := strconv.ParseFloat(fmt.Sprintf("%v", attribute.Value), 64)
    if err == nil {
      w.(http.ResponseWriter).Write([]byte(fmt.Sprintf("%s{name=\"%s\"} %v\n",
          strings.ReplaceAll(eid, ".", "_"), attribute.Name, val)))
    }
  }
}
```