SLIDE 1

SLIDE 2

Edge Computing is a paradigm in which nodes are placed at the Internet’s edge near mobile devices or IoT sensors. So "edge" can be any computing resource placed between data sources and cloud data centers.

This paradigm can address many concerns…

\* An improved latency can be provided thanks to the proximity between the edge server and the client.

\* Mobile devices' battery life can be saved by offloading the computation to the nearest edge server, instead of computing it locally.

\* And also bandwidth costs can be saved thanks to reduced usage of the network and by allowing to run compression techniques directly at the edge near the client.

SLIDE 3

In the subject of data processing on the edge, we found a gap in both the industry and in the literature.

The main frameworks of the industry that can perform data processing on the edge can only support use cases of caching, forwarding or with frequent-reads. They do not provide support for use cases with high write throughput.

In the literature this absence is often covered with the concept of stream processing, where long-running operators are placed in the network and data is bound to be flowing through these operators.

But using long-running operators in a geo-distributed network composed of hundreds of nodes is often problematic and inefficient. In fact, all the available frameworks use the Functions as a service paradigm, since it allows to reach high efficiency and since functions do not consume resources if not used and they can be easily deployed on hundreds of distributed locations.

Instead the topic of serverless platform is now becoming more relevant in the literature. And it is, in fact, on this topic that our work stands, since our solution wants to expand the current offering of edge serverless frameworks.

SLIDE 4

For example a use case that is not covered by the available framework can be the following in which we have thousands of video cameras recording multiple road sections.

We can analyze the footage of the camera to understand how much traffic that section of the road has and then devices can request the traffic data of certain areas.

>>> With a more general view this is valid for any use case where producers are creating geo-localized data that may need some processing, and consumers request geo-localized aggregations.

To make another example we can think of a geo-localized trending section of an application, where it is displayed the trending items in a certain region.

In these use cases there is an absence of the need for a fully global view. In fact, if a global view is needed, of course a core-centric approach would be preferred since with all the information in one point it becomes easier to create an aggregate result.

SLIDE 5

So the novel contributions that we bring are the following:

>>> We identified and classified use cases in the literature and used them to understand the requirements that needs to be fulfilled;

>>> Then we studied the current solutions provided by the industry and saw that the use cases collected cannot be fulfilled by these frameworks.

>>> So we found room for a new solution that can focus on specific characteristics of the use cases collected.

>>> We designed and then implemented the solution and used the implementation as a first evaluation method.

>>> But since emulating an edge network requires a huge amount of resources we developed also a discrete-event simulation to better evaluate our platform in a more realistic scenario.

SLIDE 6

The high-level concepts of our solution consist in splitting the infrastructure in hierarchical levels and allowing the developers to specify on which levels the clients save and access the data.

The goal is to perform the processing as close as possible to the client and then aggregate the result geographically.

>>> So, first our API allows to specify the hierarchy of the infrastructure. In serverless setups this should be done by the web infrastructure companies.

>>> Then it allows to deploy geo-distributed functions exactly where needed >>> and have a geographically-partitioned stateful support, where each location has its own set of data

>>> The developer can only focus on the levels where the data should be saved, then the actual location is obtained by the framework.

As can be seen in the image clients can interact with a specific level, and clients can both send data or request data. In the picture the act of sending data is represented with a purple arrow, while the act of requesting data is represented with a green arrow. The developer then is free to specify any behavior for the function, and with the stateful support provided the framework takes care of aggregating the data geographically. The yellow arrows represent the communication between servers to perform this geographic aggregation

SLIDE 7

By having such a vast and heterogeneous network the first step in our approach is to abstract away the difficult management of the deployment to the many edge locations.

In a traditional cloud setup the developer specifies individually on which data center to deploy, this cannot be done efficiently for a vast edge network because the developer would have to specify hundreds of specific deployments.

Therefore, to allow flexibility in the deployments and to allow the geographical aggregations we can organize the various machines running in the data centers and cloudlets in a hierarchy with multiple levels.

SLIDE 8

The developer should be able to deploy on a specific level of the hierarchy, only in a certain area and should be able to exclude a specific subsection from this area.

This can be done with our API by using the “in-every” and “in-areas” parameters, with them the developer can specify the level and the areas of the deployment, and with the except-in parameter instead some subsections can be excluded.

For example the function that can process the video footage in our road traffic monitoring use case can be deployed at the lowest level of the hierarchy and only in the areas needed

SLIDE 9

In the prototype we implemented two types of triggers for the functions: an HTTP trigger, where the function gets activated by a simple HTTP request, and a cron trigger, where the function is automatically called periodically based on the current time.

Devices can use the HTTP trigger to send or request data to the servers, while the cron trigger can be used to perform some periodical analysis on the data.

The developer can also specify RAM and CPU usage limit for each function.

>>> For the stateful support, the two "read" APIs allow only to read the values that the current location contains, so if the developer wants to access "continent" level data, the developer will need to deploy a function at the "continent" level and perform a get operation in that function.

While the two write APIs allow saving the data on one or multiple levels.

>>> In every write action there must also be specified a Time-To-Live that will be applied to that value, this forces developers to not accumulate data in the stateful support. Accumulating data should be avoided due to the more bounded resources present at the edge of the network.

SLIDE 10

Let’s see the concepts I presented applied to the specific scenario of the Road Traffic Monitoring use case.

As said we have cameras that can be used to extract traffic information, and devices that for example can use the aggregations of the data to find the shortest path between two points.

>>>

A first function can be implemented with the objective of processing the footage and extracting a value representing the traffic. With our API this data can be easily saved locally and propagated to upper levels.

>>>

Then a second function can be implemented to answer requests of traffic information in a certain area. The function can be deployed on multiple levels and the function can read the local stateful support to get the information needed.

So the device has to contact the right level based on the aggregation needed. If the device wants to know the fastest path between two points in a city, it can contact the cloudlet positioned at the city level.

SLIDE 11

For what concerns the actual implementation of our API we used the following architecture.

>>>

For each location in the hierarchy we have the same architecture, with a server running the required software. >>> For the prototype we used a container-orchestration system called Kubernetes to manage the various containers, >>> and in these containers we run two systems: a key-value database called Redis, and a serverless architecture framework called OpenFaas.

The key-value database provides stateful support for the functions, that instead are run and managed by OpenFaas.

>>> Then the developer can easily make a deployment to the hundreds edge servers in the network with the Command Line Interface that we developed. For each one of these servers OpenFaas is invoked with a request of deployment.

>>> Smart devices can invoke with an HTTP trigger the functions that are running on top of OpenFaas >>> and these function can store and read data on the local level or forward data to upper levels in the hierarchy.

SLIDE 12

We performed a first assessment on the working prototype running on an emulation of a small edge network composed of multiple Virtual Machines.

It was not possible to emulate a real edge network with hundreds of nodes due to the amount of resources needed.

But even with a small network we were able to test some of the attributes of our prototype.

By executing thousands of sequential calls to a single location we were able to test how the prototype scales up or scales down based on the load.

And also we implemented some of the use cases easily like the one of traffic monitoring that we saw or the one of trending items in a region for a mobile app.

Instead a negative impact on the prototype is the common problem of the Function-as-a-Service paradigm where a cold start time is needed to load the function for the first time.

SLIDE 13

Now I’ll present some results of the discrete-event simulation that we implemented.

We used this simulation to understand how the solution can perform in a real edge network composed of thousand of devices that interact with hundreds of geo-distributed nodes.

We of course considered the processing power of an edge server to be lower than what is available in a cloud data center. And we computed the transmission time as being predominantly affected by the travel distance

SLIDE 14

We found that by using our framework we get immense benefits in terms of reduced traffic in the network compared to a cloud solution, even when using in our framework an aggregation at a very high level like a continent aggregation.

We can understand why this happens with an example, in the road traffic monitoring use case we send the footage to the nearest edge server when using the framework, instead when using a cloud solution this footage is sent to the central data-center consuming a lot of bandwidth in the travel.

SLIDE 15

The solution also allows faster reads when the data aggregation needed is not central since the request is answered from a server closer to the client.

As can be seen in this Figure where we show how requesting data to lower levels of the hierarchy results in an improved latency.

SLIDE 16

Some use cases still can’t be fulfilled by our solution because it does not offer session consistency, meaning that this solution is impractical in dynamic contexts where a session needs to be preserved even when the clients change its geographical location.

Therefore a possible improvement and a possible research direction could be session consistency in the context of stateful serverless computing on the edge.

>>>Another problem with our solution that we noticed during our simulations is the possibility for edge locations to be overwhelmed due to random spikes in requests targeting a specific location: this is because our solution does not support the offload of the computation from an overloaded node.

>>>And finally in the context of serverless computing a common problem is the phenomenon of cold-start, which as we saw impacts processing latency. There exist solutions that mitigated the problem reaching milliseconds cold-start latencies, but unfortunately these solutions are currently proprietary.

SLIDE 17

So, after analyzing the state of the art in the literature, we collected and organized the use cases. With the use cases at hand we studied the current frameworks provided by the industry and we noticed that some of the use cases were left out and couldn't be fulfilled by the available frameworks.

Therefore we proposed a new solution which supports the characteristics of the use cases left out. We designed and then implemented a prototype for this solution.

And finally we evaluated our solution by using both the implementation and a discrete-event simulator, the results of the evaluation confirmed the power and effectiveness of the proposed solution.

SLIDE 18

Thank you :)