PAPS: A first step into the implementation phase. Partitioning a network.

Fabio Losavio, Oscar Francesco Pindaro

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, Italy
{fabio.losavio,oscarfrancesco.pindaro}@mail.polimi.it

10 - June - 2020

https://github.com/OscarPindaro/partition
https://github.com/OscarPindaro/KubePAPS

Abstract

The emergence of latency-sensitive and data-intensive applications requires to move the computational power closer to users on nodes at the edge of the network (*edge computing*). This work starts from PAPS, a framework which aims to tackle the complexity of edge infrastructures by means of decentralized self-management and serverless computing. This paper shows how the first of the four PAPS phases, *partitioning*, is implemented and integrated with a common open-source system: Kubernetes,to easily deploy the system in an already existing network. The partitioning is mainly focussed on the division of the network in multiple communities in order to reduce the scope of the following parts. The division is performed using the SLPA algorithm, which uses a label spreading technique to assign each node to a community. Once all the nodes are assigned, the Kubernetes node will be modified using the API labeling it in order to easily recognize and manage each community.

Keywords: Edge computing · Partition · Kubernetes · PAPS · Community division · SLPA

1 Introduction

Edge computing [3] is an open platform that integrates network, computing, storage, and application core capabilities on the edge of the network that is physically close to the data source [4]. It provides a computing model for edge

intelligence services. The location where the edge calculation occurs is called the edge node, which can be any node between the data generation source and the cloud center that has computing resources and network resources. The amount of devices connected to the internet, and as a consequence the amount of data shared, is constantly [5] growing and this introduced a number of different problems that have to be addressed.

1. First of all the high amount of traffic flowing in the network is making bandwidth a bottleneck for the traditional cloud computing approach.

2. With the popularity of smart home, video data for many families in the house installation network camera, privacy and data security will be a problem since data will be collected and directly uploaded to the cloud computing increasing the risk of disclosure of users personal data.

3. As more and more user applications run on cloud servers, the demand for energy consumption in large-scale data centers will be difficult to meet in the future. The existing research on energy consumption in cloud computing centers focuses on how to improve energy efficiency [7]. However,improving energy efficiency alone will not solve the enormous energy consumption problem of the data center, which will be more prominent in the environment of all things.

In response to this, the development of the application requirements of the Internet of Everything has spawned an edge computing model. The edge computing model is a new type of computing model that performs calculations at the edge of the network. In the edge computing model, the edge device has the processing capability of performing calculation and data analysis, and migrates some or all of the computing tasks performed by the original cloud computing model to the network edge device, reducing the computing load of the cloud server, slowing down the pressure of the network bandwidth, and improving the processing efficiency of data in the era of Internet of Everything. Edge computing is not to replace the cloud, but to complement the cloud, providing a better computing platform for related technologies such as mobile computing and the Internet of Things.

Existing research generally divides the architecture of edge computing from the central network to the edge of the network into three layers: the cloud computing layer, the edge computing layer, and the ending layer, as shown in Figure 1 [6, 8, 9]. Different layers are generally divided according to their computing and storage capabilities. The computing and storage capabilities of the ending layer, the edge computing layer, and the cloud computing layer are sequentially increased. In order to achieve intra-layer and cross-layer communication, each communication technology can be connected by various communication technologies, including wired communication (such as Ethernet, optical fiber), wireless communication (such as Bluetooth, LTE, ZigBee, NFC, IEEE802.11a/b), /c/g/c, satellite link or a combination of two technologies [10]. Edge computing

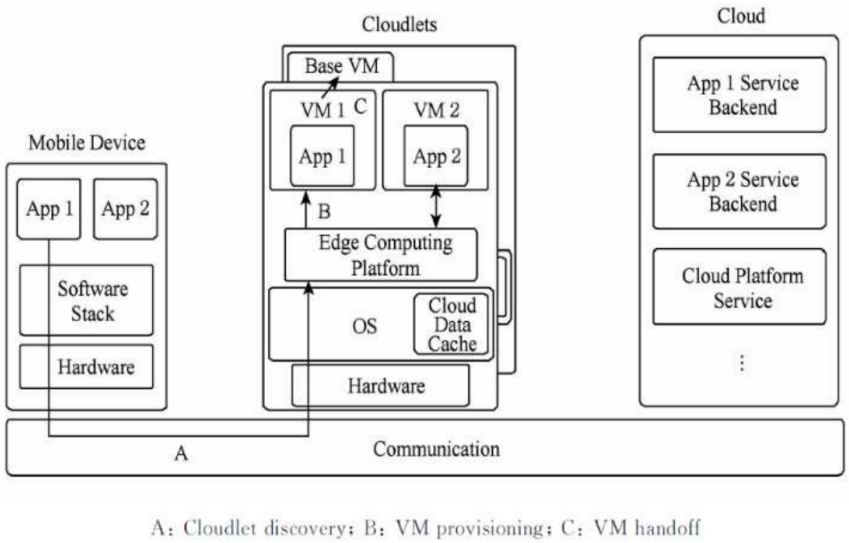


Figure 1: Edge computing infrastructure.

extends cloud services to the edge of the network by introducing an edge layer between the terminal device and the cloud.

Since some applications can not deal with the high latency from devices to cloud data centers, this approach can be useful to respect SLA requirements.
At the same time, *serverless computing*, also known as Function-as-a-Service (FaaS), offers a new execution model, which enables users to run their code (a small application dedicated to a specific task; e.g. a single-unit function) without being concerned about operational issues. This way, the user is relieved from any involvement in the server provisioning and resource management. Serverless applications are intended to be event-driven and stateless. For example, FaaS could be instantiated (triggered by the described condition) to execute a predefined function and shut down when finished. In a commercial system, a user is charged on a per-invocation basis, without paying for unused or idle resources. Being supported by unlimited Cloud resources, serverless computing represents a form of utility computing with access to elastic scaling via available on-demand computational resources.[13]
Serverless computing is becoming more and more popular, since it allows the developer to focus on the development of an application while ignoring the infrastructure on which the function will be deployed. With this paradigm, the main system can dynamically allocate machine resources according to the specific needs of the final application and being sure that the application SLA will be respected.

To easily manage cloud applications, virtualization results to be the best technology mainly in the form of *containerization.* Often, the management of cluster of containers becomes essential and the orchestration of the construction

and deployment becomes a central problem.
A container holds packaged self-contained, ready-to-deploy parts of applications and, if necessary, middleware and business logic (in binaries and libraries) to run the applications. Tools like Docker are built around container engines where containers act as portable means to package applications. This results in the need to manage dependencies between containers in multi-tier applications. An orchestration plan can describe components, their dependencies and their lifecycle in a layered plan. To interconnect multiple containers, an *orchestrator* is used. We define orchestration as constructing and continuously managing possibly distributed clusters of container-based software applications. Container orchestration allows users to define how to coordinate the containers in the cloud when a multi-container application is deployed. Container orchestration defines not only the initial deployment of containers, but also the management of the multi-containers as a single entity. It takes care of availability, scaling and networking of containers. Essentially cloud-based container construction is a form of orchestration within the distributed cloud environment.[14]

In this context, PAPS (Partitioning, Allocation, Placement and Scaling), a framework that tries to tackle the challenges related to the management of edge computing infrastructures, is trying to automate the allocation of serverless function and the management of large scale edge topologies. It should be able to give an optimal allocation for a given workload, but also to react to unpredictable workload fluctuations that characterize this type of networks. An initial prototype of the framework was already produced, as presented in PAPS [1] paper, it uses PeerSim [1] to simulate a network behavior and in which the serverless functions are implemented as Java threads. The actual workload of the system is then simulated using a probability distribution. In this paper we will present our work on PAPS, it mainly consists on a research phase in which we analyzed the documentation of some support frameworks, such as OpenFaas, OpenWhisk and Kubernetes, with the aim to simplify the deployed of a real world implementation. Based on the initial prototype we identified a general idea of how all PAPS phases will need to be implemented and integrated with the external frameworks and in the end we also developed an implementation of the partition phase.

The rest of the paper is organized as follows. Section 2 presents the theoretical structure of the framework which was the starting point of our work. Section 3 describes how frameworks and tools used in the actual implementation have been chosen among all the possible choices. Sections 4 and 5 go more in detail about the general implementation idea for the overall framework and more in details how a prototype for the partition phase has been created. Section 6 describes the tests made on our implementation. Section 7 concludes the paper.

[1] http://peersim.sourceforge.net/

2 PAPS

As already said, the importance of edge computing is growing, because some applications can not deal with the high latency from devices to cloud data centers. In this context, the way in which applications are allocated and distributed on edge nodes is crucial in order to have high performance and respect the SLA requirements of these applications.
At the same time, serverless computing is a new cloud computing paradigm that is becoming more and more popular, since it allows the developer to focus on the development of an application while ignoring the infrastructure on which the function will be deployed.

PAPS [1] is a framework that tries to tackle the challenges related to the management of edge computing infrastructures.
Its main objective is to automate the allocation of serverless function and the management of large scale edge topologies. It should be able to give an optimal allocation for a given workload, but also to react to unpredictable workload fluctuations that characterize this type of networks.
This is done by dividing this problem into four sub-problems, Partition, Allocation, Placement and Scaling.
The Partition module divides the topology in sub-networks to reduce the complexity. The Allocation module works at two different levels, community and node. At the node level new containers are allocated in order to deal with sudden changes in the workload, while at the community level its considered the aggregated demand of the whole community.
The Placement module handles the placement of containers on the nodes of the topology using the information calculated by the Allocation on the nodes of the topology. The Allocation module is supported by the Scaling module, that performs both horizontal and vertical scaling.

PAPS uses a distributed approach instead of a centralized one. In fact, centralized solutions suffer from network latency, because they have to communicate to the whole network the solution proposed, and at the same time they make very time consuming decisions and are not able to scale with the size of the network. As we will see, PAPS considers a *centralization withing decentralization*, since it divides the nodes in sub-networks called communities. Each community elects a leader that has the role of supervisor, that will ensure that the communities remain consistent and will periodically calculate an allocation plan. An advantage of this approach is that it does not need to define a communication protocol between communities, since every solution applies only to its community.
This framework focuses on MEC topologies [11, 12], composed of geo-distributed nodes which access the system through cellular base stations. In a typical topology is possible to identify two different networks: the *fronthaul network*, which connects normal nodes to the MEC stations, and the *backhaul network*, which interconnects the MEC stations. PAPS aims to reduce the complexity of the problem working at three different levels: *system, community and node*.

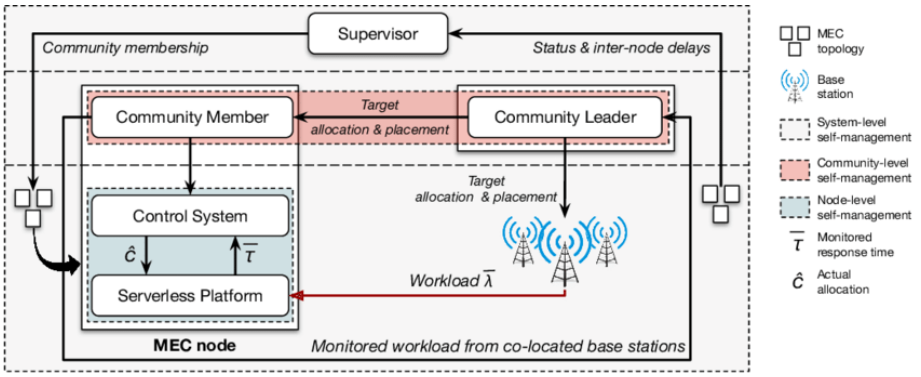


Figure 2: PAPS general structure

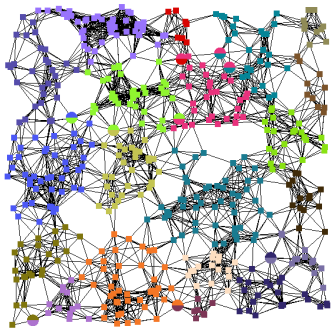
System level

Since MEC topologies can be really large and densely connected, the first step is to *partition* the network in multiple delay-aware sub-networks called *communities*. Each community is composed by a set of nodes whose propagation delay from one another is below a given threshold. PAPS assumes the availability of a *supervisor* that has a global view of the MEC topology and uses the SLPA [2] algorithm to create the communities.

SLPA is an algorithm which uses label spreading techniques to decide to which community each node will belong. It is composed of a variable number of iterations (stable solutions are obtained with 20) where nodes spread their labels around to their neighbors. The main component of this algorithm is the memory inside each SLPA node, in which it will store all the labels received.
In each iteration, all the nodes will become one at a time *listener* and will collect a label, selected from the memory, from any nearby node (*speaker*). Once all the labels are collected, the listener selects the most popular node (or selects a node with any other listening rule) and adds it to its memory. An iteration ends when all the nodes have been listener once.
Since the initial idea of this algorithm was developed to divide social networks profiles in communities, it can create overlapping node which will belong to more than one community. In PAPS a node that is a member of multiple communities will have to balance its resources in order to address the requests coming from different community leaders. Each community will elect a *leader* which will be in charge of managing the community and the communication between the nodes.

```
Algorithm 1 : SLPA(T, r)
  [n,Nodes]=loadnetwork();
  Stage 1: initialization
  for i = 1 : n do
    Nodes(i).Mem=i;
  Stage 2: evolution
  for t = 1 : T do
    Nodes.ShuffleOrder();
    for i = 1 : n do
      Listener=Nodes(i);
      Speakers=Nodes(i).getNbs();
      for j = 1 : Speakers.len do
        LabelList(j)= Speakers(j).speakerRule();
      w=Listener.listenerRule(LabelList);
      Listener.Mem.add(w);
  Stage 3: post-processing
  for i = 1 : n do
    remove Nodes(i) labels seen with probability < r;
```

(a) SLPA Pseudocode



(b) An example of communities division

Community level

Communities aims to minimize the likelihood of SLA violations so that the MEC nodes can operate under feasible conditions.
A community leader has to supervise its community, by ensuring that every node it's still up and running. A heart-bit signal for each node is checked inside the community. Each community leader will manage the *allocation and placement* phases by looking at the aggregate demand to each service and the node capacities in order to decide how to distribute resources among the nodes in its community. This is done solving a *mixed integer programming (MIP)* problem whose goal is to minimize the overall community delay. The MIP is solved periodically by the community leader which will then contact the other members to communicate the new configuration. Informed load balancers will use the information contained in this plan to route the workload coming from different base stations to the actual destination.

Node level

The Node-Level Self-Management system aims to keep the response time for a given function fixed, by deploying and removing containers accordingly to the measured workload. In order to achieve this, every node has a controller for each function that is hosting; all the controllers run in parallel and independently. Each controller accepts as input the number of container allocated for its function and the measured arrival rate, and outputs the response time. The objective is to keep the response time at the control set point, which must be lower than the SLA.

Currently PAPS is a prototype that works on a simulated network in PeerSim, in which the serverless functions are implemented as java threads. The prototype runs on a single machine and simulates the workload by drawing from a set of probability distributions. The goal of our project is to integrate

PAPS with real world frameworks, in order to test and have a better idea about the feasibility of this project on a real multi-node environment.

3 Research

Framework research

Since PAPS focuses on serverless function, initially the research looked for a framework that would allow the user to create in a immediate way functions and would give the opportunity to manage their deployment and placement inside the network. In this way, the creation of containers is sped up, because the framework automates all the process of creating a container and eliminates the need of adding useless boilerplate code.

The first candidate was OpenWhisk [2], a serverless platform able to run functions on top of an existing infrastructure. The idea was to integrate it on top of a Kubernetes [3] cluster, running a series of docker [4] containers.
Searching on the internet information about the integration of those platforms, other two options were available: OpenFaas [5] and Kubeless [6]. For what concerns the functionalities the three frameworks are pretty similar one to another. They all provide a mechanism to deploy functions that can be written in various programming languages. These functions accept arguments and give a response in the form of a json file, whose structure is determined by the programmer. In this way the mechanisms that deploy and call the functions are language agnostics.

Function calls are controlled with Triggers, components that reacts to certain events inside the ecosystem (such as HTTP request or the call of another function). OpenWhisk offers a very detailed protocol on how to design triggers, functions, and rules that connect these two elements. On the other hand, OpenFaas is less focused on this functionality and more oriented on HTTP calls.

Kubeless stands out since it has a built-in synergy with Kubernetes. In order to support the creation of functions, it extends the Kubernetes API by creating a CustomResourceDefinition, a RESTful resource that is saved inside the Kubernetes system. This new type of resource can be accessed like any other normal object inside Kubernetes by making REST calls (via CLI or HTTP request).

The main issue with OpenWhisk and Kubeless is that they are written in Scala and Akka; OpenWhisk also has a complex structure and for that reason is not optimal for a project like this. For what concerns OpenFaas, it is written in Go and offers some additional features like a metrics system and the possibility to customize the scheduling policy used by the framework.
During the research process one main problem came out: all the frameworks

[2] OpenWhisk website: https://openwhisk.apache.org/
[3] Kubernetes website: https://kubernetes.io/
[4] Docker website: https://www.docker.com/
[5] OpenFaas website: https://www.openfaas.com/
[6] Kubeless website: https://kubeless.io/

hide to the final user the underlying infrastructure, making the integration of PAPS functionalities hard. To solve this issue, the best decision was to build something directly on top of Kubernetes using the provided API, available for a large variety of languages, to create, manage and delete containers when needed. For what concerns the framework, OpenFaas came in handy to provide a frontend module useful for PAPS users which will be easily able to upload the functions that PAPS will manage and distribute. We chose OpenFaas since it offers both a simple UI and the possibility to be managed by a CLI. It was also the only one that offered the possibility to install a statistics collection module, which can be used to monitor how the system is working in a real environment.

OpenFaas

OpenFaas is a serverless framework that does not manage the infrastructure, since it delegates this task to other IaaS frameworks (such as Kubernetes). It works at the application level, easing the creation, management and deployment of functions on the cluster.

A function is a piece of software that provides some functionality, receiving request and sending response in a json file format. It will be deployed in the cluster in the form of multiple containers, on which the user has no direct control or knowledge. This allows the user to have a high level view of its functions, without having to worry about the complexity of the management of the infrastructure, since functions and containers are decoupled.

For what concerns the architecture, the main component of OpenFaas is the **API-gateaway**. It exposes the REST-API that allows the user to manage its functions. Both function calls and management requests are sent to the gateaway, which in turn will provide to contact the right component in order to perform the desired action. The API-gateaway can be invoked via CLI, UI or HTTP requests.
When a function is deployed, on or more pods/containers are created. The **WatchDog** is the component that is responsible for starting and monitoring functions. It becomes an init process with an embedded HTTP server, and can support concurrent requests, timeouts and healthchecks.
The **Alert Manager** controls the traffic to which the function are subjected. This is done monitoring the metrics produced by Prometeus. When the current configuration cant keep up with the current workload or too much containers are allocated, the Alert Manager fires an alert to the API-gateaway, that will provide to change accordingly the number of containers.
All these components are builded on the cluster as standalone containers, and so they can be also duplicated depending on the cluster administrator needs.

Kubernetes integration

Kubernetes is an orchestration framework that automates deployment, scaling and management of containerized applications on a multi-node cluster. The containers are bundled in particular entities, Pods, that contain all the depen-

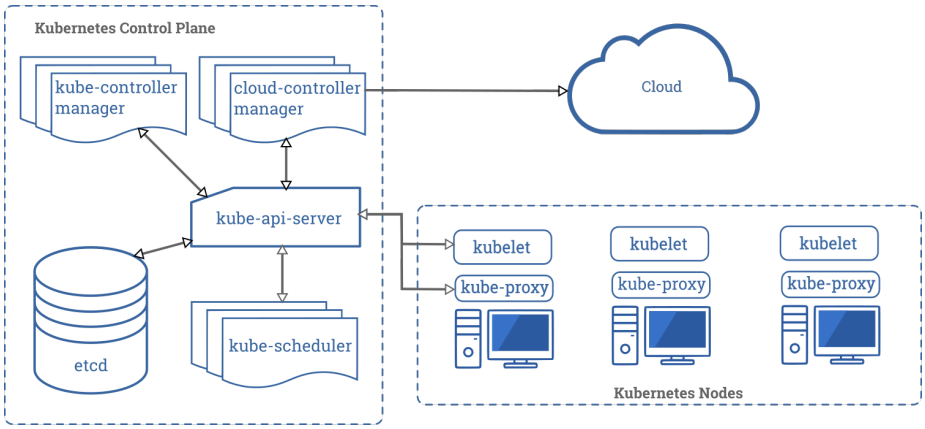


Figure 3: Components of Kubernetes.

dencies and additional metadata about the deployment. A set of controllers allows the user to automate in a flexible way the deployment of these pods.
Kubernetes offers to the user REST API, through which all object can be created, read, updated and deleted. This object are stored inside the system and describe the desired state of the cluster.
All objects have a metadata field, in which labels can be added and removed. Labels can be used to mark an object, and their meaning is defined by the programmer that created it. Labels allows filtering while asking for resources, and so they can be used to assign a particular role or attribute to a certain object.

The infrastructure is managed with the help of some key components, that are divided into two categories: Control Plane components and Kubernetes Node components.

The **Control Plane components** has the role of managing the cluster, by making global decisions. It detects and responds to cluster events. These components can be run on any machine, but usually they are placed on a node called Master. The **kube-api-server** is the frontend for the Kubernetes Control Plane. It implements Kubernetes' REST API, that can be accessed with a CLI or an HTTP request. It allows to create and manage resources, giving fine grained information about the cluster.
The **etcd** is a key-value based storage, in which all the data about the cluster are saved.
The **kube-scheduler** is notified when a pod has been created. Its function is to place this pod on a node, taking into account some factors such that individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, and inter-workload interference. The default scheduler can be replaced with a custom scheduler. A schedule routine is composed of two phases. The *Filtering* phase has the objective to identify a set of candidates nodes on which the pod can be scheduled, while in the *Scoring* phase a score for each node is calculated with the help of

some heuristics. The node with the highest score is the one that will host the given pod. This two phases are ignored if the current pod has already specified in its fields the node in which it will be scheduled.
The **kube-controller-manager** runs all controller processes that manage both nodes and pods. The *Node controller* checks the status of the nodes in the cluster, responding accordingly if a node fails. The *Replication controller* controls all the pods, and reacts to changes of the pod number by allocating or deallocating accordingly to the desired state and the workload. If a node fails, all the pods that were active on it will be placed again thanks to its routines.

The **Kubernetes Node components** are placed on the node of the cluster and handle the real creation and destruction of the containers. **Kubelet** makes sure that containers are running in a Pod. It receives from the kube-api-server a PodSpec (an object that describes all the dependencies and characteristics of the pod that will run on the node) and checks if the containers described are running and healthy. Kubelet does not manage containers that have been created outside of Kubernetes.
The **kube-proxy** allows nodes to communicate with the api server. It maintains network rules, allowing pods to communicate with one another.
The **Container Runtime** is the software responsible for running containers.

4 General Implementation Idea

Thanks to the results of the research phase, we decided to use mainly kubernetes functionalities and extension possibilities to implement PAPS phases and use OpenFaas as a frontend module to allow an easy deployment of the serverless functions.

The first of the four PAPS phases is the **partition**, in this phase the original network topology on which PAPS will need to work is divided into delay-aware communities in order to reduce the scope of the following parts. To do that the idea is to take as input a delay matrix describing the measured delay between the nodes of the networks and other additional parameters that can be useful to describe the desired behavior of the partitioner. This input can be provided as a file input or even through the OpenFaas GUI and will than be parsed by the program to fill some data structures. The partition will first of all connect to kubernetes API manager and do a call to obtain the list of registered nodes to check if they match with what has been parsed from the input. This operation needs to be done in order to avoid the presence of nodes in the topology that are not actually managed by kubernetes and so that can't be modified and labeled according to the specific needs. After this check, the program can start dividing the network into delay aware communities, using SLPA, according to some additional parameters also provided as input. With this algorithm there is the possibility to obtain an *overlapping node*, a node belonging to more than one community. In the original PAPS idea this is handled by dividing the resources that has to be assigned to that node among the overlapping communities, but, for the sake of simplicity, in our implementation this is solved by assigning that

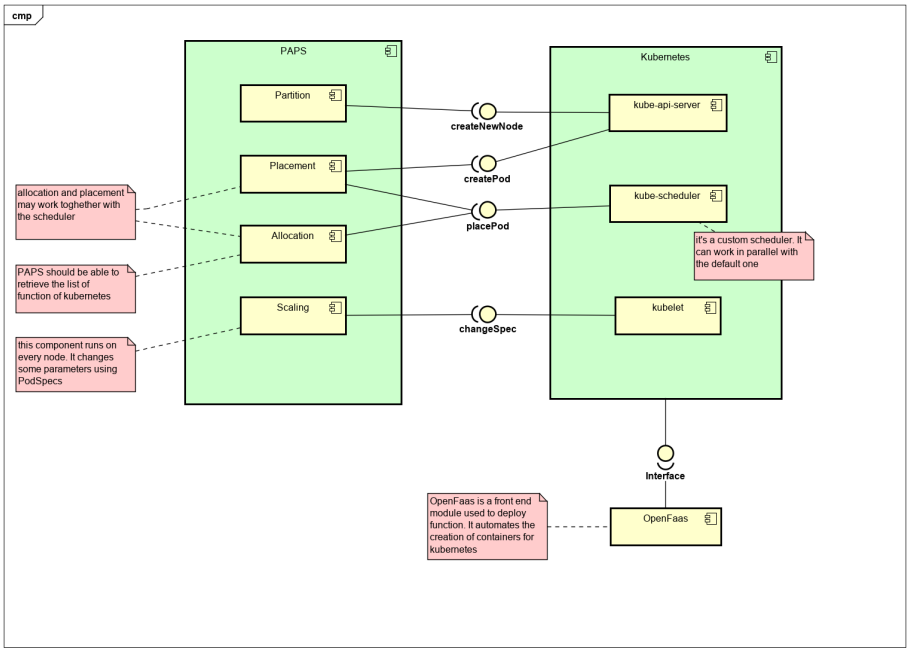


Figure 4: Component diagram of PAPS

node to only one of the overlapping communities. Since all the labels kept at the end of the process are valid ones, there are several ways to assign that node to one of its communities. The simplest idea is to just randomly choose one of them or, with some more processing, is also possible to pick the most popular one among all the remaining labels. In our implementation we select the first one in the list, which is equal to picking a random one since all the nodes are shuffled at each round, but it can be easily changed by changing few lines in the SLPA code.

As an additional division tool the Partition keeps the size of the communities under control applying Round Robin to oversized communities to split them in smaller ones and have all communities under a given threshold size.
The Kubernetes label feature can be used to perform the partition of the network. Every Node can be marked with two labels, "COMMUNITY" and "ROLE". The former assigns to the node the PAPS community to which it belongs, while the latter specifies if the host is the leader of the community or just a member that executes the leader's allocation plan. By filtering on these labels, all the nodes of a community can be retrieved, easing the tasks performed by the Allocation and Partition modules.

Once the network is split in communities, in each one the **allocation** phase

will take place. There the community leader will compute the resource allocation to assign to each node according to the current load of the network and to respect the SLA of each function deployed. To do that a MIP model is solved internally, using the IBM solver [7], stored in a dedicated container inside the leader node. Since it is inside the community leader, it will need to have access to some performance metrics regarding the other nodes of its community and this can be easily done through the kubernetes API by filtering with the correct community label. The result of the allocation will tell how many pods of each function will be scheduled on each node in order to meet the SLA for all the received request at that moment. The allocation phase will be periodically triggered to check if the provided allocation schema is still valid or needs to be changed.
All the nodes will also implement an internal allocation phase which will change the amount of resources assigned to each scheduled pod inside the node itself since between two allocation cycles the amount of resources needed to meet the SLA of each function will slightly change. To do that a pod for the allocation will be always deployed in each node and it will contain a function which works only on local resources.

The allocation phase will work in combination with the **placement** phase using a *kube-scheduler* module to actually deploy the pods necessary to realize the computed model. The kube-scheduled will be a custom module running in the network and will contain some rules on how to deploy pods according to the MIP result. To correctly schedule pods, the filtering rules of the scheduler will be modified to filter by community and for a particular node in the community. Once the correct node is identified, the scheduler will actually deploy the pod in the node and from now on it will be up and running and ready to be used.

The last phase is the **scaling** which will use a PID controller to keep meeting the SLA of all the functions on each node. Since this is a node level function, it will be deployed in a pod on each node, maybe together with the allocation part, and will be always up to check the resource utilization. This part will trigger the kubelet module of kubernetes by changing the PodSpec relative to the deployed function if needed to keep the performance under the given SLA. If a more drastic change is identified, it will make a call to the leader and ask for a new MIP evaluation even before the normal control period.

[7] IBM ILOG CPLEX Optimization Solver: https://www.ibm.com/products/ilog-cplex-optimization-studio

5 Partition Implementation

The scope of this project is to develop the first of the four PAPS phases: the network partitioning. This part of the framework has the role of dividing the network into delay aware communities in order to reduce the number of nodes on which the next phases will need to work. The first piece needed to partition the network is the network itself: it will be provided as an input in the form of a delay matrix, from a node to any other node, and the number of nodes which are part of that network.
Once the input is parsed and data are acquired, the program checks if the nodes specified in the input are really present in the cluster. To do that the program interrogates Kubernetes, through a call to the API, to obtain the list of objects (V1Node) related to all the nodes registered in the cluster and checks if the names given in the input matches with real cluster nodes. If all the matches are positive, each selected node is wrapped in a SLPA node to allow an effective partition of the network. This wrapping adds to the initial Kubernetes structure a memory and a list of nearby nodes and a label which it will try to spread.
To run SLPA, the program takes as inputs, the number of iterations, a probability threshold and a maximum size for the communities. To improve the randomness of the label spreading, both the list of the listeners and the list of the speakers are shuffled at each iteration. For each listener in the topology, its neighbors will become the speakers and from them it will obtain a list of labels and the most received one will be added to the memory (*listening rule*). Every time the memory is updated, the node will select the label that he will spread according to the *speaking rule*. In our implementation the speaking rule uses a *ranking selection* algorithm which randomly picks a node according to a probability distribution based on the occurrences of each label in the memory. If a node is present many times in the memory it will be select with an higher probability.
After the given number of iterations, the memory of each node will be post-processed, the most popular node in the memory will determine its community. In case of multiple labels with the same number of occurrences, the node will be assigned to multiple communities and will be considered an *overlapping node*. Since PAPS doesn't want to deal with those kind of nodes, the overlap will be solved selecting randomly one of the overlapping communities.
With this procedure the network will be effectively divided into communities but, to keep the size even more under control, oversized communities will be split to better fit the size limit (also provided as input).
Now that all the nodes are assigned to a community, another API call will modify the existing node that will add a label in order to easily recognize and manage each community.

6 Testing

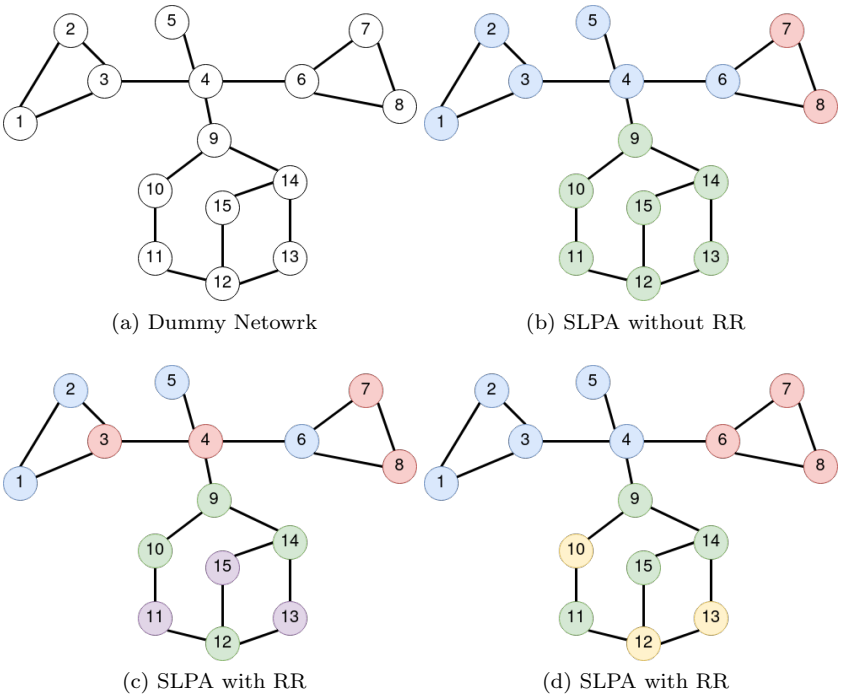
Our prototype has been written in Java and makes use of the REST API of Kubernetes. This component was initially a Java executable that operated outside a Kubernetes cluster, but at the end it has been shipped inside a container using the building functionality of OpenFaas. The testing will check two main characteristics: functional requirements and performance.

Functional Requirements

First of all we needed to check if the Kubernetes API was reachable from our code. The first executable was configured to access a cluster from outside. This was done by loading the administrator configuration file in the API. The limit of this method was that the component had to be launched on the administrator machine with a JVM and that it had full control of the cluster.
These problems were solved by building a container with our executable by using OpenFaas and deploying it in the cluster.
In Kubernetes permissions are granted to pods by creating new Roles. These are objects that tell which API calls can be made by an agent inside the cluster, and can specify which objects can be affected by a call.
Since a running pod does not have access to the old configuration file (because it's not deployed on the administrator machine), it doesn't have the permission to read and updated nodes' labels, and so we had to link it to a Role. After doing this procedure, the behavior of the component inside and outside the cluster is identical, since it does not affect the SLPA code.

For what concerns SLPA, the algorithm in the container runs only if the nodes that are given as input are registered inside the Kubernetes cluster. Since this check does not have any effect on the execution of the algorithm and requires the presence of real hosts in a network, we decided to unit-test the algorithm locally outside the container. We created a dummy network (5d) by filling the delay matrix and giving it to the algorithm. The delay matrix is a NxN matrix, where N is the number of nodes of the network. The cell M[i,j] contains the delay between the node i and j. If this cell is set to -1, there is no arc that links the two nodes. A delay threshold given as input will set to -1 all the cells that have a delay greater than that. In this example, the network shown is the one obtained after deleting all of these arcs that have a too high delay. The delays are not shown in the figure in order to keep the visualization simple and because SLPA does not consider the value of the arcs but only their presence. Note that in a real world scenario this delay would be measured for example by pinging the target node.

Figure 5b shows the result of the algorithm without Round Robin. In this output the size of the different communities varies a lot. Since SLPA makes some decision based on chance, reapplying the algorithm on the same network will give different results, creating each time different communities. Figure 5c and 5d shows the effects of the Round Robin on the communities that were



(a) Dummy Netowrk

(b) SLPA without RR

(c) SLPA with RR

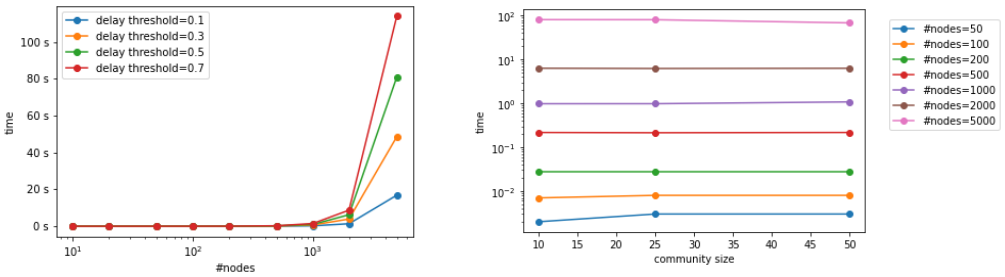
(d) SLPA with RR

Figure 5: Example Networks

exceeding in size. In this examples, the maximum size of a community was set to 5. In figure 5c SLPA returned two communities, one that included all the nodes from 1 to 8 while the other one included the remaining ones. These were broken down by the RR, creating 4 communities that cannot be separated in 4 different partitions. Figure 5d shows a luckier iteration of SLPA, that returned only one oversized community (from node 9 to node 15).
The fact that some nodes are not directly connected to the others of the same community is not a problem, since sometimes also SLPA outputs communities with a structure similar to the one returned by the Round Robin in these examples. However, if SLPA returns a very large community, with size much larger than the maximum size allowed, a set of communities with disconnected nodes could be created, since there is no logic that tries to group closer nodes. This result would be not optimal, since for sure exists another configuration that can have lower overall delay between the nodes of a community.

Perfomance

The performance of the SLPA algorithm has been measured locally outside the container, due to all the consideration done in the Functional Requirements section. The following tests have been done in *Ubuntu 18.04.4 LTS* on a *AMD*



(a) Time in function of nodes

(b) RR impact on performance

Figure 6: Performance graphs

*Ryzen 7 1700 Eight-Core*Processor.
The algorithm has been tested on the following inputs: number of nodes in the network, delay threshold and maximum community size. The number of nodes determines the size of the delay matrix. We tested networks with 10, 20, 50, 100, 200, 500, 1000, 2000 and 5000 nodes. As previously said, the delay threshold is a constraint to which all cells of the delay matrix are subjected to. Every arc with and edge value grater than the delay threshold is cancelled (the corresponding cell in the matrix is set to -1). We tested with the following thresholds: 0.1 , 0.3, 0.5, 0.7. The maximum community size limits the number of nodes that can be in a community and we tested the same values reported in the PAPS testing phase (10, 25, 50).
For each combination of these inputs a NxN delay matrix is built, where N is the number of nodes in the network. Every cell of the matrix has values between 0 and 1 drawn from a uniform probability distribution. In this way, the delay threshold can be used to maintain a certain percentage of edges (a delay threshold of 0.3 will roughly maintain the 30% of the edges). For each input a test is run ten times, tracking the execution time. In order to avoid noisy measurements, the median of this values is considered instead of the mean.

The figure 6a plots the execution time in function of the number of nodes and delay threshold. For small networks the algorithm runs in a few millisecond, and it manages to complete in about 10 seconds networks with 1000 or 2000 nodes. SLPA starts to struggle with networks with 5000 nodes, starting to show an exponential trend.

The figure 6b highlights the fact that the maximum community size does not change the performance of the algorithm, because the total execution time is dominated by SLPA. This means that Round Robin does not degrade performance, since its the only part of the algorithm affected by the community size, but just a constant cost regardless of network size.

7 Conclusion

Our work focused mostly on finding the best frameworks that would support PAPS functionalities. These serverless computing frameworks turned out to be excellent tools for the production and deployment of stateless functions, but they also had evident limits in managing the underlying infrastructure. Since we needed a much lower level of abstraction, we decided to use also Kubernetes and extend its functionalities, taking advantage of its modular architecture. We implemented the Partition module on Kubernetes, and we think that the remaining modules will be placed here as well. In this way, OpenFaas has the role of a frontend module, that allows the user to upload its function in the cluster without having to care about the infrastructure. We chose OpenFaas for its excellent support and simplicity, but it can be substituted with other frameworks with very little effort, since PAPS works at the Kubernetes level. PAPS is mostly focused on a workload that come from outside the MEC network, and so the trigger functionality, that links the execution of multiple functions, was very marginal. If a future work will focus on a micro-services architecture, OpenWhisk may be a better option, since it offers a cleaner and more powerful structure for this mechanism.

For what concerns the implementation of the Partition module, we decided to write ourselves the SLPA algorithm, and we used a round robin approach to break down too large communities. This part of the component could be substituted by using the GANXiS [8] software, a program that runs the same algorithm and has more complex and effective techniques to reduce the size of the communities inside the given boundaries. The next step in the development of PAPS will have to be the implementation of the Allocation and Placement components. Their functioning is completely decoupled from the Partition module, since they just need to know the structure of the communities, and any change in its logic will have no effect on how the container will be disposed inside the cluster. Kubernetes offers very powerful tools that allows to schedule and place pods on the nodes of the cluster, while taking care that every container is healty and running. Thanks to this functionality, the Allocation and Placement will just have to produce an allocation plan without having to care how to enforce it.

[8] GANXis: https://sites.google.com/site/communitydetectionslpa/

8 References

References

[1] Baresi, L., Mendonça, D., Quattrocchi, G.: *PAPS: A Framework for Decentralized Self-management at the Edge.* In: Service-Oriented Computing, pp.508-522, (2019).

[2] Xie, J., Szymanski, B K, Liu, X.: *SLPA: Uncovering Overlapping Communities in Social Networks via A Speaker-listener Interaction Dynamic Process.* In Proc: Data Mining Technologies for Computational Collective Intelligence Workshop at ICDM, Vancouver, CA pp. 344-349, (2011)

[3] Wang R, Yan J, Wu D,: *Edge Computing: Applications, State-of-the-Art and Challenges, Advances in Networks.* Vol. 7, No. 1, pp. 8-15. doi: 10.11648/j.net.20190701.12 (2019)

[4] Shufen Wang: *Knowledge-Centric Edge Computing Based on Virtualized D2D Communication Systems.* IEEE Communications Magazine, 56 (5): 32-38. (2018)

[5] Zhang Yanjun, Yang Xiaodong, Liu Yi, Zheng Dayuan, Bi Shujun: *Research on the Frame of Intelligent Inspection Platform Based on Spatio-temporal Data.* Computer & Digital Engineering, 47 (03): 616-619+637. (2019)

[6] Y. Wu, Y. Liu, S. H. Ahmed, J. Peng, A. A. A. El-Latif, *"Dominant dataset selection algorithms for electricity consumption time-series data analysis based on affine transformation* IEEE Internet of Things Journal, pp. 1–1. (2019)

[7] 1 Satyanarayanan M.: *The Emergence of Edge Computing.* Computer, 50 (1): 30-39. (2017)

[8] Tuyen Xuan Tran, Pompili D.: *Joint Task Offloading and Resource Allocation for Multi-Server Mobile-Edge Computing Networks* PP (99): 1-1. (2017)

[9] Feng W, Jie X, Xin W.: *Joint Offloading and Computing Optimization in Wireless Powered Mobile-Edge Computing Systems.* IEEE Transactions on Wireless Communications, PP (99): 1-1. (2017)

[10] Mao Y, Zhang J, Song S H.: *Stochastic Joint Radio and Computational Resource Management for Multi-User Mobile-Edge Computing Systems.* IEEE Transactions on Wireless Communications, 16 (9): 5994-6009. (2017)

[11] Mach, P., Becvar, Z.: *Mobile edge computing: A survey on architecture and computation offloading.* IEEE Comm. Surveys and Tutorials 19(3), 1628–1656 (2017)

[12] Several authors: *Mobile edge computing (mec); framework and reference architecture.* Tech. rep., ETSI GS MEC (01 2019), http://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/01.01.01_60/gs_MEC003v010101p.pdf

[13] Gadepalli P.K., Peach G., Cherkasova L., Aitken R., Parmer G.: *Challenges and Opportunities for EfficientServerless Computing at the Edge.* pp. 1. (2019)

[14] Pahl C., Brogi A., Soldani J., Jamshidi P.: *Cloud Container Technologies: a State-of-the-Art Review.* IEEE Transactions on Cloud Computing. PP. 1-1. (2017).