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

xx - May - 2020

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 [1], 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 [3], 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 [2] 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

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 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, Planning 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 Planning 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, 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*.

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. Each community will elect a *leader* which will be in charge of managing the community and the communication between the nodes.

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.

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 [5], a serverless platform able to run functions on top of an existing infrastructure. The idea was to integrate it on top of a Kubernetes 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 [6] and Kubeless [7]. 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 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 dependencies 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.

STA PARTE MI SA CHE E' DA SPOSTARE *This feature can be used in PAPS to perform the partition of the network in communities. In fact, Kubernetes doesn't hide the structure of the cluster, and allows to obtain information about the hosts running through the Node object. Each Node correspond to an actual host that is connected to the cluster, and therefore can be managed by*

Kubernetes. 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.

4 General Implementation Idea

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 [2] 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 understand why those structures are added, a brief introduction on SLPA is needed. It is composed of a variable number of iterations (stable solutions are obtained with 20) where nodes spread their labels around to their neighbors. In each iteration, all the nodes will become once *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. When it will become speaker it will select a label from its memory according to a probability distribution based of the number of occurrences of each label in the memory. An iteration will end when all the nodes have been listener once.
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 Conclusion

7 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] https://kubernetes.io/

[4] https://www.docker.com/

[5] https://openwhisk.apache.org/

[6] https://www.openfaas.com/

[7] https://kubeless.io/