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

PAPS is an edge computing framework which aims to improve network performances of some applications moving the computation closer to the final user instead of doing it on a centralized server. This will reduce the amount of traffic flowing into the network with the result of an improvement of the throughput allowing to better meet the SLAs of delay sensitive applications.

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 complex, 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 SLPA [2] 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.
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.

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

To start the process of developing PAPS, the first thing was to search for any useful tool that could speed up the process of creating what was designed. The starting idea was to use OpenWhisk [5], a serverless platform able to run functions on top of an 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 option were available: OpenFaas [6] and Kubeless [7]. For what concerns the functionalities the three frameworks are pretty similar one to another. OpenWhisk and OpenFaas work almost identically, and both are used in many applications nowadays. Kubeless has a built-in synergy with Kubernetes, since it's written on top of it, and for that reason it could be a good alternative. 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 a lot of 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 have the characteristic to hide to the final user the underlying infrastructure making it hard to integrate PAPS functionalities in it. 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 also offers the possibility to install a statistics collection module which can be really useful to monitor how the system is working in a real environment.

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, the Pods, that contain all the dependency 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.

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/