Network-Integrated Edge Computing Orchestrator for Application Placement

Vasileios Karagiannis, Apostolos Papageorgiou
NEC Laboratories Europe, Heidelberg, Germany
basilkaragiannis@gmail.com, apostolos.papageorgiou@neclab.eu

***Abstract*— In an effort to detach applications from centralized clouds with high latency responses, service providers turn their attention to edge computing solutions that offer low latency and improved user experience. Existing edge deployment strategies use network-related information as decision basis, but their design and their placement logic are biased by the assumption that the network cannot be controlled. In this paper, we design an orchestrator that operates within the telecom infrastructure and assumes cooperation with access and core network controllers. As a result, network adjustments can be requested, which leads to an orchestrator that participates in the provisioning of resources and solves an optimization problem that -contrary to the state of the art- performs sequential component placement and does not assume a known or fixed replication degree of the applications. Its function relies on heuristics, including one based on pre-computed shortest paths, which runs in polynomial time (i.e., much faster than an exhaustive search) and finds the optimal solution in approximately 99% of the tested scenarios.**

***Keywords*— Edge Computing, Application Placement**

I. INTRODUCTION

The main system design since the emergence of the Internet of Things is centered on cloud-hosted applications and their interactions with end-devices that integrate sensing and actuating features [8]. Even though cloud-hosted servers offer sufficient processing power and storage to support multiple users, the increasing traffic from a growing number of clients causes bandwidth bottlenecks on the underlying network [10]. Edge computing counters this effect by pushing multiple instances of the (server-side) application closer to the end-users, while keeping the programming model intact. Hence, the traffic from end-devices follows shorter paths, which reduces latency and hinders data accumulation.

Although application-hosting is monopolized by cloud providers (e.g., Amazon or Microsoft), telecom operators have been trying to enter this business. Edge computing poses a great opportunity for them because they already have access to edge infrastructure i.e., cellular base stations with service hosting capabilities, currently used as liaisons to server-side applications instead of hosting them. A telecom-driven application-hosting infrastructure requires an application placement controller, with architecture that resembles specifications as ETSI MEC [5] or OpenStack [9]. Accordingly, its placement logic can pursue goals such as: minimum latency [1] or execution delay [12]. However, all these studies miss an important observation that can make a difference in the current application-hosting landscape, i.e., the fact that telecom operators control the network and the entire infrastructure in a fine-grained way before, during, and after application placement operations. Taking this into account, we develop an Edge Computing Orchestrator which, contrary to related work, i) interacts with controllers of the access and core network, ii) places applications until their requirements are satisfied, considering the option to request network and infrastructure adjustments, and iii) reduces the placement problem to an Optimal Subset Selection [4], instead of the typical Bin Packing or Knapsack [2]. As a result, this orchestrator applies a placement logic that is tailored to dynamic networks which favor application deployment on the edge.

II. RELATED WORK

There are two fields of related work, namely application placement architecture and application placement algorithms.

Regarding **application placement architecture**, we mention the Nova scheduler of OpenStack [9] and the Mobile Edge Orchestrator of ETSI MEC [5]. Both components are part of architectures that deploy applications on the computing nodes that best fit the application requirements. Nova [9] maintains a view of all the nodes and determines the placement of virtual machines based on a filtering mechanism that focuses on compatibility features. However, it is explicitly disconnected from the networking components of OpenStack and does not consider requirements that originate from the application provider. The Mobile Edge Orchestrator [5], deploys applications on appropriate hosts based on requirements related to computation, storage, latency etc., but lacks an interface to the Network Level, thus being forced to ignore cases as: detecting violations in application requirements (e.g., required bandwidth or latency thresholds) or requesting additional resources. Finally, [3], [13] and [6] provide further information on related architectures and frameworks.

Regarding **application placement algorithms**, related work can be found in various domains such as content distribution or cloud server management. In [1], the authors develop algorithms for resource allocation in cloud environments that are geographically distributed on a wide area network. Required resources for a computational task are assumed to be known a priori and the problem is formulated as finding the nodes with available resources that minimize the maximum latency among all the latency values induced by the links that interconnect the selected nodes. They conclude by stating that an algorithm based on subgraph selection provides significant gains over alternative methods. In [7], the authors explore the placement problem in the context of Context Delivery Networks. They design heuristics that replicate and deploy objects on selected

nodes so that the cost, which is associated with the hop count, is minimized. The results point to the conclusion that a greedy approach can result in performance that is close to optimal. In [12], placement is applied to solve the problem of distributing workloads from mobile users to computing nodes. The authors organize cloud and edge nodes in a tree hierarchy and develop a placement logic that outsources excessive load to nodes higher in the hierarchy, with optimization goal to minimize the execution delay. Finally, [11] addresses failure scenarios and performs fault-tolerant placement by deploying redundant instances of the application.

Despite the variety of placement logic in the literature, no related work addresses the exact problem of deploying multiple tasks with requirements to each other on multiple nodes when task replication is supported but the optimal number of replicas is unknown. To solve this problem, we formulate task placement in a different manner, as explained in the next section.

III. Edge Computing Orchestrator

The Edge Computing Orchestrator (ECO) is an application placement controller that relies on controllers of other layers to help in meeting application requirements. Such solutions can lead to business benefits, especially for telecom operators, since they are the only ones who can natively implement this multi-layer cooperation, because of being able to control the whole network infrastructure in a fine-grained way.

A. Architecture

The high-level architecture of the ECO is depicted in Fig. 1 and includes the following components:

Controller Cooperation Coordinator (CCC): It implements an interface to send/receive requests to/from other network controllers. It is used for: i) initial submission of applications along with a Service Level Agreement (SLA), ii) dynamic network slice negotiation (refers to the capacities of nodes and links, that remain private to this application), iii) reception of requirement violations (in layers not visible to ECO) or reconfiguration requests (e.g. migrations and resource adjustments) initiated by other network controllers.

Network Topology Assembler (NTA): It converts the network slice (computed by the controller that provisions resources) into a graph representation that is compatible with the internal algorithms of the Deployment Plan Finder.

Deployment Plan Finder (DPF): It computes an optimized plan for replicating and deploying applications on computing nodes based on their SLA, the network slice and the location of client-side users (acquired from access network controllers).

Deployment Plan Executor (DPE): It receives a placement plan from DPE or independent requests from CCC and converts them into properly formatted commands that can be executed by an Infrastructure Manager (e.g., Nova scheduler [9]).

Note that dynamic slice negotiation and detection of violations based on metrics in the access and core network are only possible due to the introduced inter-controller interactions.

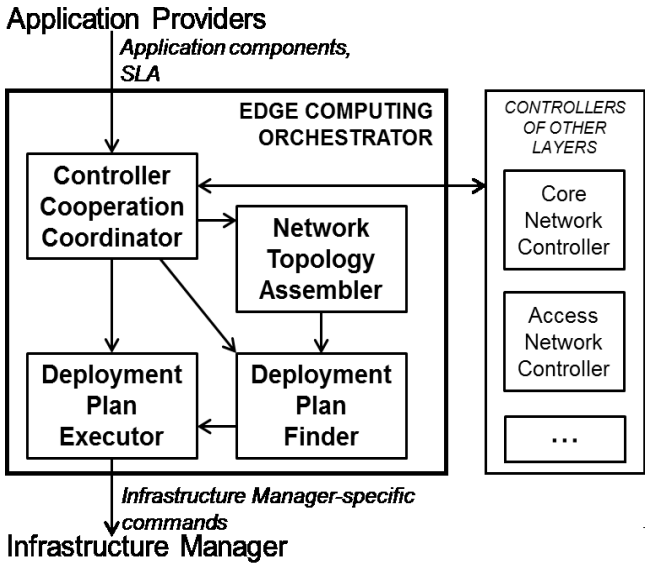

Fig. 1. High-level architecture of the ECO

B. Problem Formulation

The DPF handles task placement based on the optimization problem that is formulated in this section. The problem is firmly shaped by the following assumptions: i) applications consist of components, each one executing a specific task and being deployable separately from the others, but potentially interacting with them. Each component is paired with a unique priority value that reflects its importance. ii) There are quantitative requirements among the components (e.g. latency). Each requirement is associated with a component and is referring to another one of higher priority (for avoiding duplicate requirements). iii) Applications intend to serve final-users and thus, the highest priority component has requirements towards client-side applications. iv) Components can be replicated and deployed on each one of the computing nodes.

The network slice is a set C of I nodes: $C = \{c_1, c_2, \dots, c_I\}$. Latency between two nodes c_1 and c_2 is denoted as: Lat_{c_1,c_2}. An application is a set E of J deployable components: $E = \{e_1, e_2, \dots, e_J\}$ and each component has a set Q of X_e hardware requirements: $Q = \{q_1, q_2, \dots, q_{X_e}\}$ and a set R of M_e latency requirements: $R = \{r_1, r_2, \dots, r_{M_e}\}$. Requirements towards client-side applications are redirected to the nodes of the set C that are closest to users of the client side. This information is shared though the CCC by access network controllers. Each component is treated as a separate case of optimization and beginning from the highest priority component e, the goal is to find a set $T \subseteq C$ such that replicating and deploying e on the nodes of T results in the minimum sum of latencies of all the latency requirements of e. Additionally, it is required to have the minimum number of replications of e that achieves minimum latency (i.e., cardinality of the set T). Hence, minimum cardinality of T:

$$|T| \; with \; T \subseteq C \; and \; T = \{c_1, c_2, \dots, c_k\}$$

subject to the minimum value of the sum of the latencies towards each one of component e's latency requirements:

$$\sum_{r_1}^{r_M} Lat_{c_a,c_b}$$

with $c_a \in C$ being the host node of the component that the requirement r is referring to, and $c_b \in T$ the node that has min latency from c_a, under hardware constraints so that hardware requirements of components are subject to the respective node capacities (e.g., CPU, RAM, storage).

To prove complexity, we rephrase the problem to a case of the Optimal Subset Selection [4]. The search space is a set A that contains all possible subsets of C and is comprised by $2^{|C|}$ items. The optimal solution is the item of A that satisfies the requirements of the optimization goal i.e., min latency and replications, under hardware constraints. The Optimal Subset Selection is known to be NP-hard in its general form and thus, this problem has the same complexity. Identifying the optimal solution in NP-hard problems becomes impractical for large datasets so in the next section, we develop applicable heuristics.

C. Placement Heuristics

In this section, we develop heuristics that compute optimal and near-optimal solutions based on the optimization goal. The pseudocode follows the notation of the problem formulation and performs placement for one component only. The exact same logic is repeated sequentially (based on priorities) for the rest of the components of the application.

1) Optimal Placement based on Exhaustion (OPE)

This heuristic searches the whole search space to identify the optimal solution. It has exponential time complexity.

Algorithm 1: OPE

Input: C, e | **Output:** dp

```
int minSumLatency = INF; // variable to hold the min value found for the
                            cost function. Initially set to infinity.
int sumLatency; // variable to hold temporary values of the cost function,
                   i.e., for each examined possible solution.
List currSubset; // a list variable to store the nodes of each possible
                    deployment plan, i.e., each subset of C.
List selectedNodesList; // a list variable to store the nodes of currSubset
                           that have enough resources to host the component.

for ( i = 0; i < 2^|C|; i++ ){ // 2^|C| possible deployment plans…
  currSubset = C.getSubset(i); // returns the i-th subset of C (which is
                                  also, a list of computing nodes)
  selectedNodesList = {};

  for ( j = 0; j < |currSubset|; j++ ){
    if ( currSubset[j].fits(e) ){ //returns true if the j-th node in currSubset
                                     can host the component e, false otherwise
      selectedNodesList.add(currSubset[j]);
  } }

  sumLatency = e.sumLatencyIfDeployedOn(selectedNodesList); //
returns the value of the cost function in the case that e is deployed on each
one of the nodes of nodesList

  if (sumLatency < minSumLatency) {
    dp = selectedNodesList;
    minSumLatency = sumLatency;
} }

return dp; // list of nodes on which the component shall be deployed
```

2) Greedy Placement Unlimited (GPU)

This heuristic deploys each component on all the nodes that have enough resources to host it. Its time complexity is linear.

Algorithm 2: GPU

Input: C, e | **Output:** dp

```
List selectedNodesList; // a list variable to store the nodes that have
                           enough resources to host the component.

for ( i = 0; i < |C|; i++ ){
  if ( C[i].fits(e) ){
    selectedNodesList.add(C[i]);
} }

dp = selectedNodesList;
return dp; // list of nodes on which the component shall be deployed
```

3) Greedy Placement with Cost threshold (GPC)

In this heuristic, the node that results in minimum latency is selected for replication and placement until the value of the latency-related metric of the optimization goal drops below a threshold. The time complexity is polynomial.

Algorithm 3: GPC

Input: C, e, threshold | **Output:** dp

```
int minSumLatency = INF; // variable to hold the min value found for our
                            cost function. Initially set to infinity.
int sumLatency; // variable to hold temporary values of our cost function,
                   i.e., for each examined possible solution.
List selectedNodesList; // a list variable to store the nodes that lead to a
                           minimum cost in each iteration.
int bestNodeIndex; // a variable to keep the index of the node that leads to
                      the best value of the cost function in each iteration.

for ( i =0; i<|C|; i++ ){
  minSumLatency = INF;

  for ( i =0; i<|C|; i++ ){
    if ( C[i].fits(e) & C[i] NOT IN selectedNodesList){
      selectedNodesList.add(C[i]);
      sumLatency = e.sumLatencyIfDeployedOn(selectedNodesList); //
                   returns the value of the cost function in the case that e is
                   deployed on each one of the nodes of nodesList
      if (sumLatency < minSumLatency) {
        bestNodeIndex = i;
        minSumLatency = sumLatency;
      }
      selectedNodesList.remove(C[i]);
  } }

  selectedNodesList.add(C[bestNodeIndex]);
  if (minSumLatency≤threshold) {
    break;
} }

dp = selectedNodesList;
return dp; // list of nodes on which the component shall be deployed
```

4) Optimized Placement based on Shortest Path (PSP)

For every host node of a component a requirement is referring to, this heuristic selects the node of the shortest path

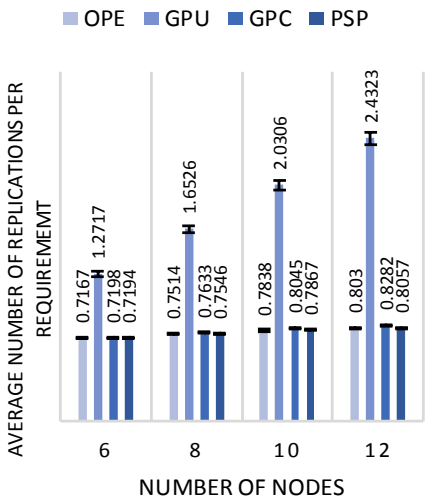

Fig. 3. Average number of replications per requirement each heuristic requires in order to find minimum latency.

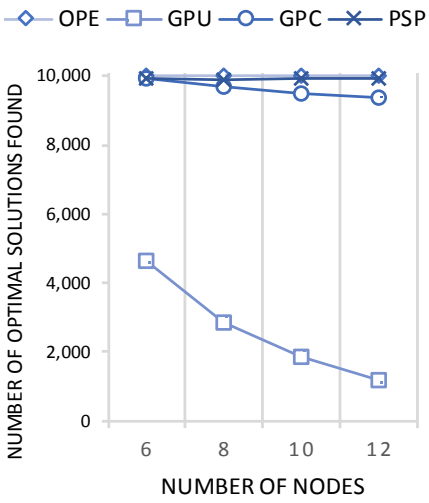

Fig. 4. Number of times each heuristic finds the optimal deployment plan (i.e., the same as OPE).

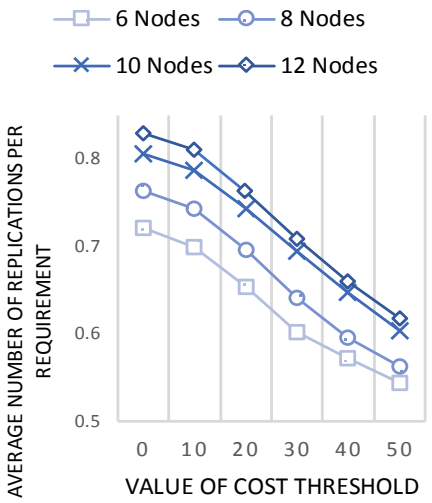

Fig. 2. Average number of replications per requirement needed by GPC in order to achieve the latency that is indicated by the cost threshold.

that has enough resources to host the component to be deployed. Note that this approach benefits from the inter-controller communication because ECO has access to core network information. Core network controllers maintain a static view of the topology and can calculate shortest paths only once upon initialization, but share these values with ECO when requested. Not having to calculate shortest paths reduces the time complexity of the heuristic, which is polynomial.

Algorithm 4 PSP

Input: C, e | **Output:** dp

```
List selectedNodesList; // a list variable to store the nodes that already
                           been selected for the optimal deployment plan.
List examinedNodes; // an initially empty list to store the nodes that have
                       already been examined for hosting a replica of the component.
List R = e.getRequirements(); // returns a list R of requirements (of
component e to already placed replications of other components), in which
R[i] is the i-th requirement, and R[i].host is the node on which the
replication that this requirement refers to is hosted.

for ( i = 0; i < |R|; i++ & R[i].host NOT IN examinedNodes){
  examinedNodes.add(R[i].host);
  // The "sorting command" below actually corresponds with a single action
     of retrieving the respective (already sorted) list
  Sort_the_elements_of_C_based_on_shortest_path_to_R[i].host;
  for ( j=0; j<|C| ; j++ ){
    if ( C[j].fits(e) & C[j] NOT IN selectedNodesList){ // returns true
          if the j-th node in C can host the component e and in not in
          selectedNodesList.
      selectedNodesList.add(C[j]);
      break;
} } }

dp = selectedNodesList;
return dp; // list of nodes on which the component shall be deployed
```

IV. Evaluation

Using a Java-based implementation of the ECO and by simulating the rest of the infrastructure, we perform an evaluation that focuses on the optimality (or near-optimality) of the developed heuristics. We use four topologies (with 6, 8, 10 and 12 nodes) and simulate 10000 placement scenarios for each one of them. Each scenario is created by assigning random values to variables related to the infrastructure such as: link latency (1-20), node CPU cores (0-10), node RAM (0-2000), node storage (0-4000) and related to the application component such as: number of requirements to other components (1-4), CPU cores (1-5), RAM (100-1000), storage (200-2000).

Fig. 3 shows the average number of replications per requirement required by each heuristic to find minimum latency. Note that, as indicated by the overlapping confidence intervals (99%), OPE and PSP are very close to each other.

Fig. 4 shows the number of times each heuristic succeeds in computing the optimal solution. Note that PSP is extremely close to OPE, despite the increasing number of nodes, whereas the success rate of GPC and GPU decreases.

Fig. 2 is based on the performance of GPC and shows how increasing the threshold gradually, results in fewer replications and thus, reduces the number of utilized resources within the network slice. The cost threshold represents the sum of latencies towards the component's requirements as expressed in the optimization goal and thus, the same unit applies. Hence, GPC also depicts the tradeoff between the number of replications and effective latency delay and provides a mechanism to adjust this tradeoff based on the application provider's requirements.

V. Conclusion

We propose the architecture and the internal algorithms of an edge computing orchestrator that performs replication and placement of application components in a telecom-driven application-hosting infrastructure. This orchestrator implements an interface for collaborating with access and core network controllers. This leads to a placement logic that is different to the state of the art (e.g., dynamic network slice negotiation, sequential component placement) and thus involves a different placement optimization problem and respective heuristics. Among the developed heuristics we highlight the introduced Placement based on Shortest Path (PSP) as being the only solution that comes very close to optimality in polynomial time.

REFERENCES

[1] M. Alicherry and T. V. Lakshman. “Network aware resource allocation in distributed clouds.” 31st Annual IEEE International Conference on Computer Communications (INFOCOM 2012), pp. 963-971, IEEE 2012.

[2] S. R. M. Amarante, F. M. Roberto, A. R. Cardoso, and J. Celestino. “Using the multiple knapsack problem to model the problem of virtual machine allocation in cloud computing.” 16th IEEE International Conference on Computational Science and Engineering (CSE ‘13), pp. 476-483. IEEE, 2013.

[3] S. A. Baset. “Open source cloud technologies.” 3rd ACM Symposium on Cloud Computing (SOCC ‘12) , pp. 28-29. ACM, 2012.

[4] M. Binshtok, R. I. Brafman, S. E. Shimony, A. Martin, and C. Boutilier. “Computing Optimal Subsets." Association for the advancement of artificial intelligence (AAAI) press, pp. 1231-1236. AAAI, 2007.

[5] ETSI (European Telecommunications Standards Institute). “Mobile edge computing (MEC) framework and reference architecture (GS MEC 003 V1.1.1)”, March 2016, online (last visited June 2017): http://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/01.01.01_60/gs_MEC003v010101p.pdf

[6] J.M. Kang, H. Bannazadeh, and A. Leon-Garcia. “Savi testbed: control and management of converged virtual ict resources.” In International Symposium on Integrated Network Management (IM 2013), pp. 664-667, IFIP/IEEE, 2013.

[7] J. Kangasharju, J. Roberts, and K. W. Ross. “Object replication strategies in content distribution networks.” Computer Communications, vol. 25, no. 4, pp. 376-383, Elsevier, 2002.

[8] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate. "A survey on application layer protocols for the internet of things." Transaction on IoT and Cloud Computing vol. 3, no. 1, pp. 11-17, 2015.

[9] OpenStack, documentaion on Nova Scheduler, online (last visited June 2017): https://docs.openstack.org/developer/nova/filter_scheduler.html

[10] M. Satyanarayanan. “The emergence of edge computing.” Computer, vol. 50, no. 1, pp. 30-39, IEEE, 2017.

[11] B. Spinnewyn, B. Braem and S. Latré. “Fault-tolerant application placement in heterogeneous cloud environments.” 11th International Conference on Network and Service Management (CNSM), pp. 192-200, IEEE, 2015.

[12] L. Tong, Y. Li, and W. Gao. “A hierarchical edge cloud architecture for mobile computing.” 35th Annual IEEE International Conference on Computer Communications (INFOCOM 2016), pp. 1-9, IEEE, 2016.

[13] M. Yannuzzi, F. van Lingen, A. Jain, O. L. Parellada, M. M. Flores, D. Carrera, J. L. Pérez, D. Montero, P. Chacin, A. Corsaro, and A. Olive. “A new era for cities with fog computing.” Internet Computing vol. 21, no. 2, pp. 54-67, IEEE, 2017.