Docker Layer Placement for On-Demand Provisioning of Services on Edge Clouds

Piet Smet, Bart Dhoedt, and Pieter Simoens

***Abstract*—Driven by the increasing popularity of the microservice architecture, we see an increase in services with unknown demand pattern located in the edge network. Pre-deployed instances of such services would be idle most of the time, which is economically infeasible. Also, the finite storage capacity limits the amount of deployed instances we can offer. Instead, we present an on-demand deployment scheme using the Docker platform. In Docker, service images consist of layers, each layer adding specific functionality. This allows different services to reuse layers, avoiding cluttering the storages with redundant replicas. We propose a layer placement method which allows users to connect to a server, retrieve all necessary layers -possibly from multiple locations- and deploy an instance of the requested service within the desired response time. We search for the best layer placement which maximizes the satisfied demand given the storage and delay constraints. We developed an iterative optimization heuristic which is less exhaustive by dividing the global problem in smaller subproblems. Our simulation results show that our heuristic is able to solve the problem with less system resources. Last, we present interesting use-cases to use this approach in real-life scenarios.**

***Index Terms*—Service-centric, on-demand, placement algorithm, long-tail, services, docker.**

I. Introduction

OVER THE LAST few years the amount of services on the Internet has increased rapidly while users have increasing performance demands. Cloud Computing was developed to facilitate resource scaling, resilience and security while distributed clouds bring services to the edge network, closer to the users [1]. The monolithic nature of most services prevents different components of a service to be scaled separately, making high service availability difficult to accomplish under increasing demand. Instead of interweaving all functionality into one service, we see more and more services designed as separate, loosely coupled services which communicate with each other to accomplish full functionality. This approach is commonly referred to as microservice architecture [2], [3]. Microservice architectures allow efficient code reusability and more fine-grained scaling.

Another reason for the popularity of microservices is the emergence of container virtualization. Cloud Computing uses virtualization techniques to deploy, relocate or scale Virtual Machines (VM) dynamically to meet the changing service requirements. Due to the high resource demand of VMs on the host machine there was a need for a more lightweight approach aimed at hosting a specific application, which led to the development of software containers [4]. Unlike VMs, software containers share the same operating system kernel with the host machine to reduce the on-demand provisioning overhead and provide more efficient resource usage. This design goes hand in hand with microservice architectures as each lightweight service component can be provisioned on-demand in a container and scaled for its specific requirements.

Containerized services are increasingly deployed on edge cloud infrastructure, e.g., for low-latency processing of IoT sensor data or to protect sensitive data by keeping the analysis close to the source of the data [5].

These software containers must be deployed in a way that efficiently uses the limited resources available. Although there is already a large body of work on service placement for popular services based on the availability of user demand patterns, our focus is on the growing amount of long-tail services coming from the microservice architecture. We define a long-tail service as a service with infrequent demand, for which no statistically reliable demand pattern is available to find a suitable pre-deployment scheme. The predicate 'long-tail' stems from the business marketing domain, and refers to the shape of the curve when ranking sold items in decreasing popularity [6]. While there are a few very popular products, the long tail of a very large majority of infrequently sold items (niche markets) represents an economically equality attractive business opportunity. With the advent of cloud computing, the Software-as-a-Service model has made it much easier to offer a delivered service to the end-user which resulted in a very long tail of services that are used relatively infrequently [7], [8]. Recently, the shift from monolithic application designs to micro-service oriented system architectures further increases the number of services that must be orchestrated. Note that a novel service might start as a long-tail service, but might lose this status as it becomes more popular and more data on demand patterns become available.

A key element in the long-tail paradigm is that the economic valorization of the items in the long tail should not be neglected. Hence, the long tail of software services should not be overlooked by operators of distributed cloud infrastructure.

When it comes to finding an optimal deployment scheme, pre-deploying service instances across the edge network is economically infeasible because they are infrequently used.

Manuscript received October 25, 2017; revised March 17, 2018; accepted May 27, 2018. Date of publication June 5, 2018; date of current version September 7, 2018. This work was supported by the UGent BOF-GOA Project "Autonomic Networked Multimedia Systems." The associate editor coordinating the review of this paper and approving it for publication was M. Steinder. *(Corresponding author: Piet Smet.)*

The authors are with the IDLab, Department of Information Technology, Ghent University—imec, 9052 Ghent, Belgium (e-mail: piet.smet@ugent.be).

Digital Object Identifier 10.1109/TNSM.2018.2844187

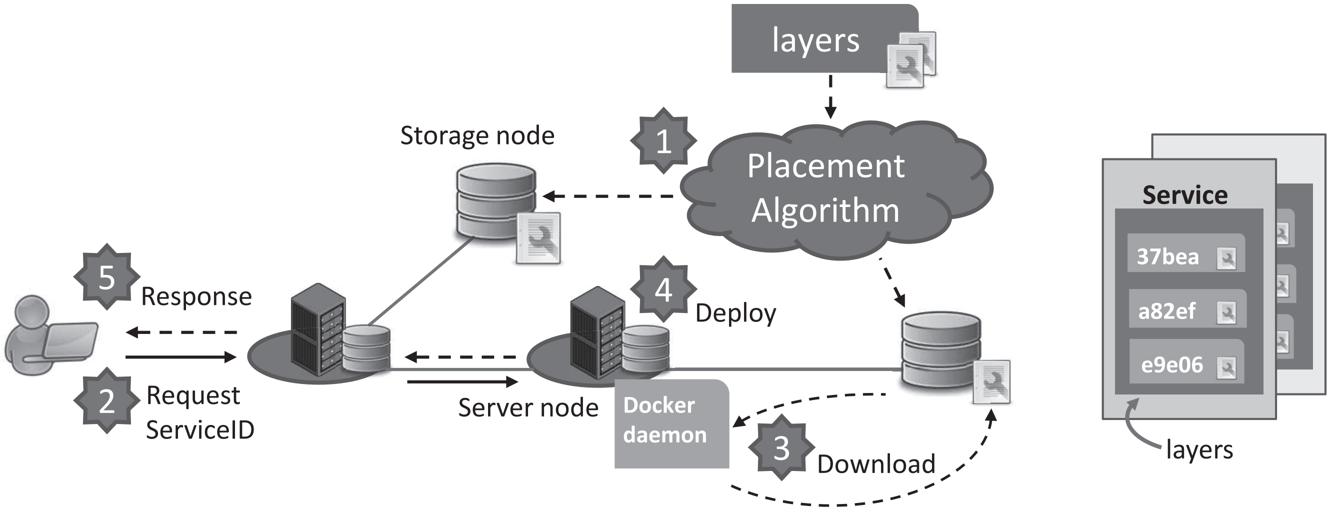


Fig. 1. We place service layers on storage nodes so that, upon service request, Docker can download the required service layers, possibly from different locations, and deploy a service instance within the desired response time. The server nodes, located in the edge network, have limited storage capacity.

Additionally, pre-deploying instances limits the number of services we can offer due to limited resources on edge nodes which is an issue for the high amount of services considered. In this case on-demand provisioning is a better solution than pre-deploying instances across the edge network. To facilitate on-demand provisioning we can use the popular framework Docker [9], an open-source project which utilizes operating-system-level virtualization to facilitate the deployment of applications inside software containers. Container images are broken down into layered filesystems (*layers*) which can be considered the service building blocks, each adding a piece of required functionality. Layers can be shared by multiple services and retrieved from remote storage nodes. This allows more efficient storage usage and adds new optimization possibilities to the service placement problem which have not been explored yet.

In this paper we propose a service placement method which maximizes the amount of clients we can serve on-demand within a desired service response time, focusing on services without known demand patterns. Following the Docker design, services are broken down into layers which can be shared by multiple services. Assuming that a client may connect from any given client location, we place the layers so that users can connect to a server, retrieve the layers from their perspective storage nodes and deploy a service instance on that server within a desired response time. Docker handles the layer retrieval and instantiation so we only concern ourselves with finding a suitable location for the layers (Fig. 1).

The research presented in this paper extends our previously published work from [10]. We added an additional approach to support parallel downloads as seen in the Docker v2 API. Additionally, we introduce an iterative search method to solve larger scenarios using less system resources. New benchmarks were added to compare our approach with existing solutions. Last, simulation results were obtained by modeling real-life networks and latency distributions.

In the remainder of this paper we present our service placement models and discuss our experiment results. In the next section we describe related work on service placement, multi-stage selection algorithms and relevant research on Docker, all important aspects of our research. In Section III we explain the Docker software in more detail as it is an important part of our research. In Section IV we formulate our problem statement for both a serial and parallel download model. Next, we introduce an iterative search method in Section V. Our simulations are driven by realistic datasets so that our results match real-life scenarios, we describe this process and our results in Section VII. Last, we discuss future work which we will expand on in Section VIII.

II. Related Work

A. Service Placement

Due to dynamic pricing schemes on Clouds, service placement algorithms often aim to minimize the total hosting cost for a set of services while trying to maximize a performance metric [11]. Zhang *et al.* [12] minimize the hosting cost while ensuring a minimum response time. This is also a dynamic algorithm which considers change in user demand over time and adjusts the amount of service instances deployed at each datacenter. The researchers assume that services are instantiated so users can connect instantly. In contrast, we are dealing with services without known demand patterns (e.g., long-tail services or newly deployed services) so we cannot assume any distribution of user demand (in most cases the service runs idle), making it economically infeasible to pre-deploy such instances. Moreover, the storage capacity constraints limit the amount of services we can deploy.

Similar to our model, other approaches aim to maximize the satisfied demand given a set of constraints (budget, storage capacity...). Famaey *et al.* [13] assume that services are instantiated before a request arrives. The selection algorithm in this research only needs to connect clients to servers as it assumes the services is already fully pre-deployed. As this is economically infeasible for services with limited edge cloud resources, our approach places service layers on strategic locations so users can deploy services on-demand within a desired response time.

Other research on service placement in so called micro clouds has similar goals to ours; to make services easily accessible in the user's proximity. Selimi *et al.* [14] find the optimal placement for services by considering bandwidth, network delay and node availability. This research was later optimized [15] with a focus on availability and bandwidth. Although bandwidth is only a secondary objective in our research, it affects the download time of our service layers which is part of our primary user satisfaction constraint and therefore also relevant.

Trneberg *et al.* [16] present a model that combines cost efficiency, bandwidth, computational resources and client location. Finding a solution for this model also requires an exhaustive search which is not suitable for large-scale scenarios. Similar to our research, an iterative approach is employed to get near-optimal results. However, this existing research also assumes a clear mapping from clients to service instances while we also consider the provisioning time to pull service layers from storage to server nodes before we can deploy an instance on-demand.

B. Multi-Stage Selection

To validate a placement, our solution must find the best possible selection to validate that users can be served on-demand with this placement. A practical example is shown in [17] where Yeoh and Chua aim to minimize construction costs by selecting a suitable crane for each supply demand and finding the best location for each crane at the same time. This is similar to the Facility Location problem which aims to find the optimal location, amount of facilities and client-facility selection to serve as many customers as possible within a certain response time. This is very useful to find the right location to place hospitals to cover as many potential locations within the desired time.

However, our solution not only assigns servers to clients but also needs to find the best storage location to download layers from to ensure that the instance can be deployed in the desired time. This is similar to the multi-stage supply chain problem where consumers are not only assigned to a distribution center but distribution centers are also linked to production plants. An example of the multi-stage supply chain is presented in [18] where Bahrampour *et al.* attempts to find a distribution center for each client, as well as which factories deliver to which distribution center. Most related research on the multi-stage supply chain problem [19], [20] all face the same difference from our research; existing research typically assumes knowledge of the user demand patterns which are harder to predict for long-tail services and newly registered services. By solving the on-demand scenario in our research, users can come online from any location and deploy a service instance on-demand within the desired response time, without needing any knowledge of the user demand.

A recent study on the two-level uncapacitated facility location problem [21] also tackles this case without assuming specific demand patterns. Although our challenge is also a two-level selection, our research is more challenging than selecting a server and storage for each client, as we must consider that all layers of a service can be located on different storages. This means that a (client, server, storage) triplet in existing research now turns into multiple (client, server, storage, layer) quadruples. Additionally, our research does consider storage capacity and makes use of layer reusability to maximize satisfied demand given this limited capacity.

C. Docker in Fog Computing

Fog computing, or also called edge computing, uses Internet of Things (IoT) devices to leverage resources in the edge network. Similar to how our research finds the optimal server and storage nodes to deploy Docker layers on, Fog computing attempts to make optimal use of nearby (storage, networking or computational) resources. Docker containers are becoming an increasingly popular choice for Fog computing development. Some examples of research relying on Docker virtualization in a Fog computing and IoT environments can be found in [22] and [23]. In the research presented in [24], a real-world Fog landscape was made using Docker containers to support the research.

IFogStor [25] introduces a data placement strategy focused on fog infrastructure. Similar to how our research attempts to quickly retrieve Docker layers in the right location, iFogStor attempts to reduce the data retrieval and service latency in a Fog infrastructure. Using integer programming and a search heuristic, this placement strategy managed to reduce the average latency by 60% compared to existing fog approaches.

III. DOCKER

Docker provides a lightweight alternative to full virtualization using Linux container (LXC) based operating-system-level virtualization. It provides portable deployment of software containers across platforms, shared and reusable layers and versioning amongst others. In comparison to VMs, Docker containers require less system resources, deploy in subseconds and focus on running applications rather than emulating hardware. Due to its modular design and ability to easily deploy and re-use services across platforms, Docker makes it easier to ensure the same deployment environment is used amongst researchers, thereby facilitating reproducible results.

Docker services are built from a series of layers, allowing us to store replicas on different locations and reuse these layers across different services. Our placement method considers these characteristics when finding a location to place layers so that a service instance can be deployed almost instantly upon request arrival. In our model each service consists of a set of reusable layers and a service can only be deployed when each layer is located on the same host.

In Docker, each service consists of a base layer and every consecutive layer adds a piece of functionality. With one pull command, Docker is able to retrieve closeby replicas of each layer and compile the service on the local host. The retrieval process differs depending on the version of Docker we use. In the first version, a service is defined by its top layer which contains a reference to its parent layer, the layer situated directly below it in the Docker image. By following the parent layer references, we eventually end up at the base layer, which means all layers of that service are retrieved and we can compile the service. Although each layer can be retrieved from a different storage location, this version assumes a serial download model; we need to retrieve each layer at once to find all layers of that service.

Since each layer was not only defined by its functionality but also by its parent reference, multiple layers with the same function were not recognized as the same layer by Docker, essentially defeating the purpose of reusability.

In the second version of the Docker API, each service is identified by its functionality by using a hash of the layer as an identifier. This allowed Docker to reuse services with the same functionality across multiple services with different base layers. Moreover, the layered structures of popular services which are officially supported by Docker are known by the Docker daemon, allowing it to retrieve each layer in parallel as it no longer needs to discover the parent layer references. This substantially speeds up the layer retrieval process.

In our research we examine both the serial and parallel download model, described in the following section.

IV. Service Placement

Our goal is to maximize the amount of users we can serve within a desired response time (*satisfied clients*), focusing on services with unknown demand pattern (e.g., long-tail services and newly registered services). The response time includes the client-server latency, the time to pull all layers onto the server node and the time required to compile the layers into a service image and deploy a service instance. We focus on the edge cloud case where small processing servers with limited resources are located in the edge network, closer to the user, while large storage nodes are located deeper into the backbone network. Pre-deploying idle service instances is economically infeasible and limits the amount of services we can offer due to the limited server resources. Instead, we aim to maximize the users we can serve on-demand by placing Docker layers at strategic locations so users can connect to a server node, pull the necessary layers and deploy a service instance on that server node within the desired response time. Our solution focuses on finding the optimal layer placement and we use Docker to facilitate service deployment. If there is no layer placement that satisfies all clients then we find the solution which maximizes the amount of satisfied clients.

We formulate our problem for both a serial and parallel download model. The serial model assumes that all layers of a service must be downloaded one at a time, similar to Docker v1 where each layer contained a field that points to the next required layer in the service. The parallel model follows a Docker v2 approach where the service structure is known beforehand and all layers of a service can be downloaded simultaneously. In both models we aim to solve our objective, to maximize the satisfied users, while using constraints to ensure that our solution is feasible.

To reduce the amount of required system resources, we divide our larger datasets into smaller ones which include only one service of the original problem. We then solve each smaller dataset consecutively and remember our previous decisions to make use of shared layers between multiple services. We compare the performance of both methods and evaluate their scalability.

A. Problem Statement: Serial Download

1) Variables: Consider a set of clients *I*, servers *J* and storage nodes *K*. We define the collection of network nodes $N = I \cup J \cup K$ as the union of these three collections. The collection of services is S and each service $s \in S$ consists of a set of layers L_s required to deploy an instance of *s*. The collection of layers to be placed on storage nodes is *L* but a layer $l \in L$ may be shared by multiple services and belong to more than one L_s.

The decision variables can take the values one or zero. $P_{i,j,s,k,l} = 1$ if client *i* connects to server node *j* to pull layer $l \in L_s$ from storage node *k*, otherwise it is 0. If we place layer *l* on storage node *k* then $Y_{k,l} = 1$, otherwise 0.

2) Constraints: The amount of used storage capacity cannot exceed the available storage capacity

$$\sum_{l\,\in\,L} Y_{k,l} * M_l \leq C_k \ \forall k \in K \tag{1}$$

where C_k is the storage capacity of storage node $k \in K$ and the storage capacity required by layer *l* is M_l. Each server node also has limited storage capacity to instantiate services or pre-deploy layers on, which is modeled as an additional storage node with latency zero to that server node.

A client *i* can only pull layer *l* from storage node *k* to server node *j* if layer *l* is placed on storage node *k*

$$P_{i,j,s,k,l} \leq Y_{k,l} \ \forall i, j, s, k, l \tag{2}$$

For each request for service *s* from client *i*, our selection result *P* must assign exactly one server node *j* where the client must connect to and for each layer *l* in service *s* there must be exactly one storage node *k* where server node *j* pulls layer *l* from. If there were multiple options, we could select multiple duplicates from different locations, which is undesirable.

$$\sum_{j\in J}\sum_{k\in K} P_{i,j,s,k,l} \leq 1 \ \forall i, s, l \in L_s \tag{3}$$

Any selection we make needs to be valid; if the selection variable $P_{i,j,s,k,l}$ does not select a storage location for every layer *l* from service *s* for client *i*, then that client can never be satisfied for that service. Therefore, when our selection result $P_{i,j,s,k,l}$ assigns a client *i* to a server node *j* for service *s*, then it must also assign a storage node *k* for each layer $l \in L_s$ where node *j* can download that layer from. We define the decision variable $Q_{i,j,s}$ to mark a selection as valid ($Q_{i,j,s} = 0$) when all the layers of a service are selected in $P_{i,j,s,k,l}$, or invalid ($Q_{i,j,s} = 1$) if not all layers of a service are assigned in $P_{i,j,s,k,l}$. This can be written as

$$Q_{i,j,s} \geq \frac{L_s - \sum_{k\in K}\sum_{l\in L_s} P_{i,j,s,k,l}}{L_s} \ \forall i, j, s \tag{4}$$

Last, the selection must allow a client *i* to connect to a server *j*, pull layers $l \in L_s$ from storage node *k* and deploy them on server node *j* within the desired response time D_s for service *s*

$$\sum_{j\in J}\sum_{k\in K}\sum_{l\in L_s} P_{i,j,s,k,l} * \left(\frac{d_{i,j}}{size(L_s)} + D_{j,k,l} + T_l\right) \leq D_S \ \forall i,s \tag{5}$$

We define the latency between two nodes $i, j \in N$ as $d_{i,j}$ and the time to download layer *l* from storage node *k* to server *j* as $D_{j,k,l}$. T_l is the required time to install a layer on the server after downloading it. To avoid double summation of the client-server latency in the nested sum, we divide the latency by the amount of layers in L_s as the client only connects to the server once for each service request.

3) Objective Function: We aim to find the placement values $Y_{k,l}$ and selection values $P_{i,j,s,k,l}$ which maximize the amount of satisfied users. Any selection where $Q_{i,j,s}$ is not zero will result in unsatisfied clients. When we find a solution where $Q_{i,j,s}$ is zero and the solution is within the boundaries of all our other constraints, then a user is satisfied. Therefore, by searching for the solutions where $Q_{i,j,s}$ is zero, within the boundaries of our constraints, we find the placement and selection with the maximum amount of satisfied clients. In other words, we need to minimize $Q_{i,j,s}$:

$$minimize \sum_{i\in I}\sum_{j\in J}\sum_{s\in S} Q_{i,j,s} \tag{6}$$

Note that Eq. 3 guarantees that only one server node *j* will be selected for each client-service pair.

If multiple solutions can satisfy the same amount of users, we search for the solution that minimizes the used capacity. The resulting objective function is:

$$minimize \ \frac{\sum_{k\in K}\sum_{l\in L} Y_{k,l}}{|K|.|L|} + \sum_{i\in I}\sum_{j\in J}\sum_{s\in S} Q_{i,j,s} \tag{7}$$

where |K| is the amount of elements in K and |L| the amount of elements in L. When multiple solutions result in the same satisfied demand, the first term will be a tiebreaker that selects the solution with the least capacity used. We will always prioritize the satisfied demand as the minimum unit of the second term is one, while the first term is always smaller than one because we divided by |K|.|L|.

We use integer linear programming (ILP) [26] to solve the above problem statement using CPLEX. As we place layers for all services in this method and assume a serial download model, we call this approach the *All Services Serial ILP* (*ILP-AS-S*). In the next section we describe how to apply this approach to a parallel download model.

B. Parallel Model

The previous problem statement assumes that all layers are downloaded sequentially which is the case in the Docker v1 API where each layer referenced the next layer to be downloaded (*parent layer*). However, Docker v2 introduced parallel layer downloads so all layers can now be fetched simultaneously. This limits the influence of storage node locations and puts more importance on network latency between clients and servers. We modify our problem statement from the previous section to reflect this change.

To adjust the serial download model to a parallel model we must replace the sum of all layer download times $D_{j,k,l}$ in Eq. 5 with the maximum download time of all layers in the service we are looking to deploy. To linearize this maximization function, we extend our model in two ways.

First, we introduce a supporting decision variable $W_{i,s}$ as the maximum download time of all layers of service *s* when downloaded from the selected server and storage nodes for client *i* as determined by $P_{i,j,s,k,l}$.

Second, in Eq. 5 we change the sum of layer download times with the one-time parallel download time $W_{i,s}$, as defined above

$$W_{i,s} + \sum_{j\in J}\sum_{k\in K}\sum_{l\in L_s} P_{i,j,s,k,l} * \left(\frac{d_{i,j}}{size(L_s)} + T_l\right) \leq D_S \ \forall i,s \tag{8}$$

Then we ensure that $W_{i,s}$ is equal or larger than the actual maximum download time of all layers in the requested service *s* for client *i*

$$W_{i,s} \geq P_{i,j,s,k,l} * D_{j,k,l} \ \forall i,j,k,s,l \in L_s \tag{9}$$

Since the solver is trying to maximize satisfied demand, it will assign small values to $W_{i,s}$ so that the total download time is smaller than the allowed delay (see Eq. 8). However, by definition, $W_{i,s}$ will still be larger than the longest download time for any layer from the selected server. As a result, $W_{i,s}$ is a good representation for the maximum download time of all the layers in that service. Therefore, using $W_{i,s}$ in Eq. 8 allows us to model a parallel download time rather than the serial download time from Eq. 5.

This is the parallel model of *ILP-AS* (*ILP-AS-P*), while Section IV-A assumed a serial download model (*ILP-AS-S*).

V. Iterative Search

Whether we use the serial or parallel model, scalability remains a challenge for ILP solvers. Typically, these solvers require large amounts of system resources to solve the problem statement and take a very long time to run. We can reduce the dataset by combining smaller layers in a service to one larger layer or by combining similar services. However, our experiments [10] showed that these methods are not impactful enough to allow *ILP-AS* (see Section IV-A) to solve very large problems.

The next step in our research was to find search heuristics to find a layer placement without requiring large amounts of system resources like *ILP-AS*. A first attempt came from a *top down* approach; the search heuristic places each layer on every storage node and then removes one layer at a time until the available storage capacity is sufficient for the remaining layers. Each iteration we removed the layer which least affected the satisfied demand in an attempt to keep the satisfied demand maximized. Our second heuristic was a *bottom up* approach were each layer is placed, one at a time, each time selecting the layer and storage node which adds most to the satisfied demand. However, both approaches are greedy and experiments showed that they performed considerably worse than *ILP-AS*, even on smaller scale scenarios. Additionally, each step of these approaches required an exhaustive search, making it less feasible for large scale scenarios due to the increasing computation time.

Rather than utilizing a greedy heuristic, we attempted to solve the layer placement problem with the Simulated Annealing search heuristic. Starting with a randomized solution, we make small changes to the current solution and compare both placements. To avoid getting stuck in a local optimum, the heuristic sometimes continues with a worse solution than the current best to try and explore other areas of the search space. In the end, we utilize the layer placement which resulted in the highest satisfied demand. Although more promising than our greedy top down or bottom up approaches, this algorithm also failed to be a convincing competitor for *ILP-AS*.

Instead, we solve the problem by placing layers for one service at a time. We employ a similar model as described in Section IV-A and run it iteratively to deploy one service at a time, where each iteration deploys the next service and considers the placement decisions from the previous iterations. We repeat this until we notice that additional iterations do not increase the satisfied demand level. We call this method **Sequential Per Service** (*SPS*), as we now place one service per run and we repeat this process for each service. This method can be applied to both the serial (*SPS-S*) and parallel (*SPS-P*) ILP models described in the previous sections.

Consider the following example; We need to serve 2 clients (U1, U2) for service A with 2 layers L1, L2 and B with layers L2, L3 on a network with two server nodes (J1, J2) and two storage nodes (K1, K2). Using *ILP-AS*, the solver evaluates every acceptable combination and comes to the conclusion that, e.g., (L1, K1), (L2, K2) and (L3,K1) is the best placement and that both users U1 and U2 should connect to server node J1. Using *SPS*, the solver first attempts to place service A and finds (L1, K1) and (L2,K2) as best solution to maximize satisfied demand for service A. In the next iteration, the solver places service B and detects that layer L2 has already been placed on storage K2. The solver evaluates the best position for L3 and finds (L3,K1) as the optimal solution for service B if both clients U1 and U2 connect to server J1.

The final placement matches this of *ILP-AS* but each iteration of the *SPS* solver had less combinations to explore and reduced the search space by detecting previous placements. The disadvantage of *SPS* is that each iteration does not have any information about the next service to be placed, making it a greedy approach in worst case. In our example, it was possible for the first iteration of *SPS* to place L2 on storage K1, preventing the second iteration from ever finding the optimal placement from *ILP-AS*.

In Section VII-C we evaluate the importance of the service order when placing services one at a time.

We start by introducing a new input variable called $Z_{k,l}$ which contains the result of the decision variable $Y_{k,l}$ of all previous services combined. During the first run $Z_{k,l}$ is an array of zeros and the solver attempts to place all layers of the first service so that the amount of satisfied demand for that service is maximized. In the second run the solver attempts to solve the problem statement with only the second service as input but this time with $Z_{k,l}$ containing the $Y_{k,l}$ values of the first run. The available capacity given as input for the second run is the original storage capacity minus the capacity used during the previous runs of our solver.

If $Z_{k,l} = 1$ we already placed layer l on storage node k during one of the previous runs and there is no use of predeploying a second replica on that storage node. We ensure that this does not happen by forcing $Y_{k,l}$ to be zero if $Z_{k,l} = 1$

$$Y_{k,l} \leq \left(1 - Z_{k,l}\right) \forall k, l \tag{10}$$

This ensures that we make use of shared layers amongst multiple services.

We adjust Eq. 2 so $P_{i,j,s,k,l}$ can select a layer placed by $Y_{k,l} == 1$ or a layer from one of the previous placements if $Z_{k,l} == 1$

$$P_{i,j,s,k,l} \leq Y_{k,l} + Z_{k,l} \ \forall i, j, s, k, l \tag{11}$$

When finding the optimal placement for all services at the same time, our solver uses the available resources to maximize the satisfied demand for all services. However, when solving the problem one service at a time, our solver attempts to maximize the satisfied demand for each service in a greedy manner. The tiebreaker in our objective function (Eq. 7) minimizes the amount of layers placed, which prevents one iteration from using all the available storage locations and starving the services in upcoming iterations.

However, our objective function alone is not enough as one service is still allowed to place many layer replicas in order to achieve 100% satisfied demand for that service, while from a global point of view this means we are still starving the available resources for all other services. Therefore, we add an additional constraint limiting our problem to just one layer replica for each service we are deploying. If we have two services with one shared layer, each layer will be placed once besides the shared layer which is allowed to be placed twice. Instead of one replica per run we can also evaluate two, three ... replicas, which results in the following constraint

$$\sum_{k \in K} Y_{k,l} \leq x \ \forall l \in L \tag{12}$$

where *x* denotes the amount of replicas for layer $l \in L_s$ we are allowed to place each time we deploy a service instance of *s*.

The objective function remains the same as in Eq. 7.

Using this adjusted model we place services one at a time and keep repeating this until we notice that placing additional services no longer increases the satisfied demand. In the next section we present our simulation results as we evaluate if this sequential approach makes our solution more scalable.

VI. Benchmarks

Both of the suggested approaches in our research use an ILP solver (CPLEX) to solve the selection and placement problem. To verify our results, we compare with two existing approaches and implemented the following two benchmarks.

A. Random Allocation (RA)

We attempt to place each layer $l \in L$ on a randomly chosen storage location. We repeat this until all storage capacity is fully occupied and no additional layers can be placed. For each client and each service, we then select the server and storage locations which result in the fastest on-demand deployment time for that service and that client. This guarantees that each client selected the ideal locations for its specific request and no quicker access to that service is possible given the randomized placement. Last, we remove all layers which were not selected by any clients in the previous step to free up storage capacity for future use.

B. Greedy Allocation (GA)

This benchmark is an adapted implementation of the 'Greedy Heuristic' presented in [27]. In this research, the algorithm starts by sorting a list of services by demand. As we are assuming that there is no knowledge of demand patterns, we start by randomly sorting the list of services instead. The remaining steps are:

1) Select the next service s in the sorted list of services.
2) For each client $i \in I$, create a list of possible candidate storages to host an instance of service s for it. Only include storages that have sufficient storage capacity available to host service s and a low enough delay.
3) Sort the storages from most to least possible clients (the storage which occurs on the most candidate lists comes first). Also include clients of previously placed services in this count. Sort the clients from least to most candidate servers (this will also give clients with only few possible candidates a chance).
4) Select the next storage k in the sorted list.
5) For each client i in the sorted list for which no candidate has been selected yet. Check if k has enough remaining storage capacity and if there is a server j which connects client c and storage k with low enough delay (take into account resources used by previously placed clients for service s now). If any clients remain without candidate and not all servers have been checked yet, return to step 4.
6) If any services remain, goto step 1, otherwise finish.

The main drawback of this greedy approach is that this model does not consider separate layers and only speaks of services. As a result, in our implementation this means that all layers of one service will be placed on the same storage node. This is a general issue with existing research which is not modeled to Docker where each service consists of layered building blocks.

VII. SIMULATION RESULTS

In this section we present our simulation results on the performance of our model described in Section IV-A. We compare *ILP-AS* with our improved scalability efforts of *SPS*. Rather than solving the problem as a whole, we divide it into smaller pieces and solve each part iteratively, considering the decisions made during previous steps. For smaller problems our simulation results show that *SPS* achieves the same result as *ILP-AS*. This makes *SPS* a viable approach to tackle larger scale problems which *ILP-AS* cannot solve without more system resources and execution time.

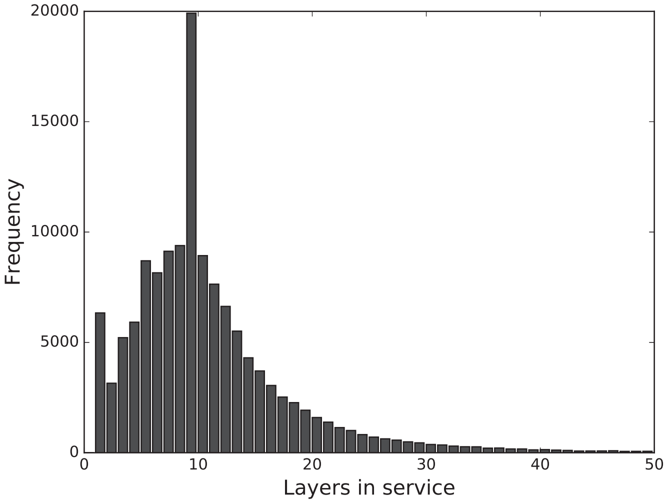


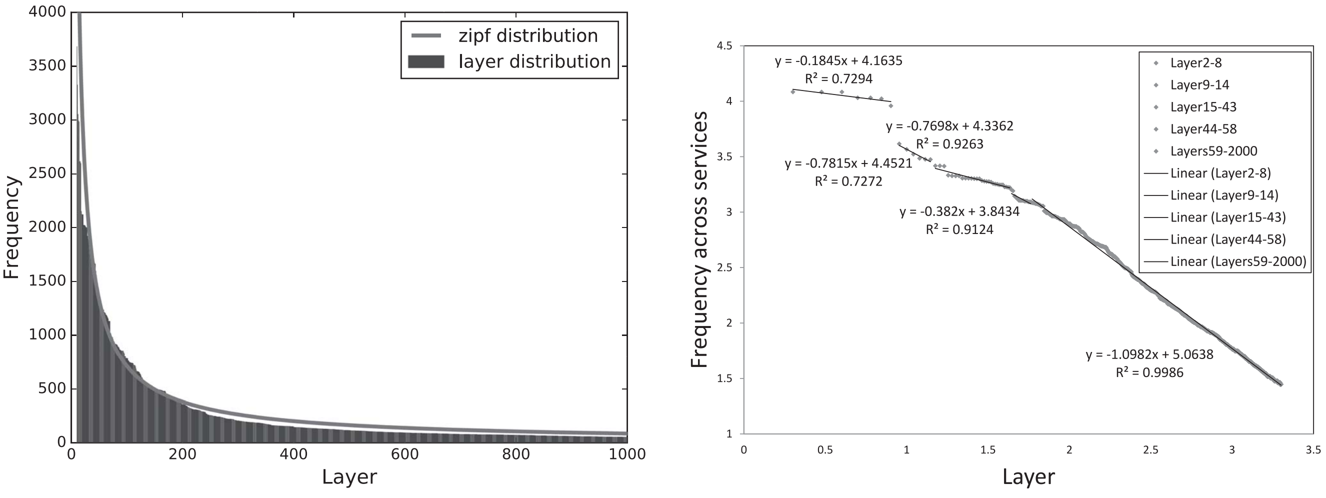
Fig. 2. A histogram visualizing the average layer distribution in Docker services. The X-axis represents the amount of layers and the Y-axis indicates the frequency of services with this amount of layers. We observe that each service has 10 layers on average.

We also investigate the influence of the execution time on the quality of our solution. More specifically, how much performance do we gain from running the *ILP-AS* solver for a longer duration. A deployment solution found by the solver will remain a suitable solution as long as the amount of services and available resources stays the same. However, as services and user demand have a dynamic nature, it is in our best interest to keep the execution time as low as possible so we can react to changing dynamics. Although the latter is part of future work, we still touch on the performance gained from longer execution times.

The service characteristics were calculated from real-life Docker repositories as explained in Section VII-A while our network topology is based on real-life latency distributions as described in Section VII-B. All results are obtained by running each simulation on five different sample networks, each one generated based on these real-life parameters, and averaging the results. All simulations are performed on the iLab.t Virtual Wall [28] using a server with a Hexacore Intel E5645 (2.4GHz) CPU, 24GB RAM, 1x 250GB hard disk and 1-5 gigabit network interface cards.

A. Docker Data Crawling

In order for our simulation results to match real-life applications, we crawled the Docker Hub [29] to determine the characteristics of Docker images. Inspecting the thousands of services in the Docker Hub showed us that services consist of few larger base layers and several smaller databases, including layers which only contain meta-data. On average, a service has 10 layers of which 10-20% are shared with other services. In Fig. 2 we illustrated our findings with a histogram, showing the amount of layers on the X-axis and the frequency of services with this amount of layers on the Y-axis.



(a)

(b)

Fig. 3. Service layer reusability modeled as a Zipf distribution (a) and a sum of Zipf distributions in different ranges (b). These results show how often a specific layer is used in the different Docker services we crawled from the Docker repository. We can approximate this reusability behavior with a Zipf distribution or more accurately using a sum of Zipf distributions.

We plotted the layer distribution for all services in the Docker repository (Fig. 3 a). The x-axis shows the 1000 most frequently used layers in descending order. The y-axis indicates how many services used this layer. At first sight this resembles a Zipf distribution but a closer fit on a LOG-LOG plot reveals that we are dealing with a sum of Zipf distribution in different ranges, as shown in Fig. 3 b.

To ensure that our data model matches a realistic Docker model, we use this data to model the average layers per service, as well as the layer sizes.

B. Latency Distributions

In order to compare our results with real-life applications, we need realistic latency distributions for our simulations. In this section we describe the latency models used in our experiments to describe a real-life network.

As discussed in Section IV, our research focuses on the edge cloud case where services run in the edge network close to the users, while our data is stored on larger storage nodes located in the backbone network. Our simulation expects client-server latencies and the estimated time required to download a layer from a storage to a given server node. Considering that servers can make use of layers deployed on other servers, we observe three latencies to model:

1) *The client-to-server latency:* Each client must connect to an edge cloud where the service will run. The main factor in this delay is the time to reach the access point, usually done through the wireless network in modern scenarios. Research [30] shows that this latency is located between 15-60ms, while studies [31] on the latency to the provider of a 4G wireless network in the UK is close to 50ms. Therefore we use a uniform distribution between 30-70ms to model the client-server latency.

2) *Server-to-server latency:* Edge clouds can download layers from other nearby clouds in the edge network so we do not have to fetch all necessary data from far away storage nodes in the backbone network. Research [32] on the distance between two servers shows that each 1000km adds 20 ms delay with a minimum threshold of 20ms RTT. As our edge clouds are all located nearby in the edge network, we model this latency as a uniform distribution between 10 and 30ms.

3) *Server-to-storage latency:* When there is no copy of a desired layer available nearby, we must retrieve that layer from storage nodes located in the backbone network. These storage nodes are often spread out across the network and can even span continents. Sources [33] show that the average latency between storage nodes in the same continent is between 50-100ms, which is the distribution we chose to model the server-to-storage latency.

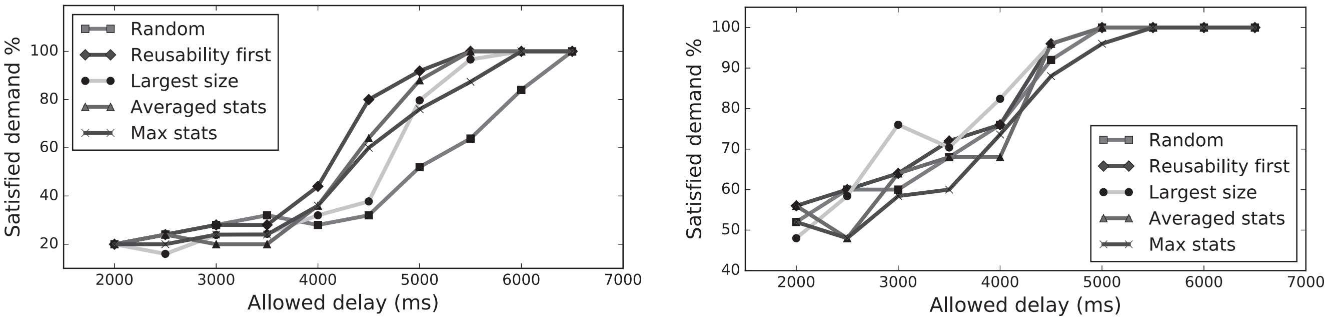
Using these link latencies we can calculate the estimated download time for each layer ($D_{j,k,l}$) as layer size (in bits) divided by the link download speed (Gbps).

This latency distribution together with the Docker distribution is used as input to drive our simulations.

C. Heuristic Optimization

In this section we investigate the importance of the service order in our iterative search method *SPS*. As there is a limited storage capacity on each node, not all services can be placed on the same node. The sorting order will dictate which services are prioritized and placed first.

Our search heuristic *SPS* reduces the required system resources by solving the problem one service at a time while utilizing previous decisions. We investigated the impact of the service sorting order on the performance of this sequential approach. As we focus on long-tail and newly registered services, we cannot predict the demand patterns for each service. From our crawling results (see Section VII-A) we know each layer in a service and their file sizes. We call the amount of services reusing a specific layer the reusability of that layer, assuming a layer can only occur once in each service. Using



(a)

(b)

Fig. 4. The performance difference between different sorting methods in our heuristic approach for (a) 25 client nodes, 25 services, 2 server and 5 storage nodes and (b) 5 client nodes, 25 services, 5 server and 5 storage nodes. In both setups 20% of the service layers are reused by multiple services. We observe that the optimal sorting order depends on the experiment setup.

this information, we investigate the following four sorting methods:

1. *Random:* we traverse all services to be placed in a randomized order. This rather naive approach is used as a benchmark for our other methods.

2. *Reusability first:* in this method we prioritize layers which are reused by most services. By processing these layers first, they can be placed in the most centralized locations, allowing the maximum number of services to reuse that replica and thus saving storage capacity elsewhere. Therefore, we summarize the reusability frequency of all layers in a service and assign that as score to that service. We then solve *SPS* in order of decreasing service score, allowing services with the most frequently reused layers to be processed first.

3. *Averaged stats:* in the previous method it is possible that the sorted queue is dominated by tiny service layers which are reused frequently. As the server nodes in the edge network typically have less storage capacity than the large storage nodes in the backbone network, it may be beneficial for the heuristic to prioritize larger layers to take up the centralized locations. As we still want to consider the reusability of each layer, we now take a weighted average of both the reusability frequency and the layer sizes.

Layer score = (reusability / average reusability) + (layer size / average size)

Service score = sum(layer scores in service)

4. *Max stats:* A slight variation of the previous method where we normalize the factors with the maximum instead of the average. Our layer score now becomes:

Layer score = (reusability / max reusability) + (layer size / max size)

5. *Largest size first:* we attempt to prioritize larger base layers by sorting the layers by decreasing size. This will put base layers near the users in smaller storage nodes while the smaller remaining layers will be placed in larger but more distant storage nodes.

Fig. 4 shows the performance of our solution for each sorting method discussed above. The X-axis represents the allowed response time to satisfy a client and the Y-axis represents the percentage of satisfied clients for the desired response time. In the first setup (Fig. 4 a) we have 25 client nodes, 25 services, 2 server and 5 storage nodes. Our second setup (Fig. 4 b) contains only 5 clients but 5 server nodes. In Fig. 4 a, reusability stands out as the best performing sorting method while random sorting is the least efficient method, taking the longest time to find a solution with 100% satisfied demand. However, in Fig. 4 b the different approaches have more server locations to choose from and we observe that our Max stats curve now performs worse than the other sorting methods.

After repeating the same experiment for different setups we concluded that the most suitable sorting order depends on the experiment setup. However, on average the sorting order does not have a big impact on the percentage of satisfied users.

In the next sections we use the reusability first method to sort services for *SPS* as this method was consistently amongst the best methods for different setups.

D. Execution Time

Given infinite system resources and time, *ILP-AS* will find an optimal solution to our placement problem. In this section we evaluate how much performance is lost by introducing a limited execution time, which is a more practical scenario. Additionally, we perform the same steps with *SPS* and compare the performance to *ILP-AS*. Our goal is to evaluate if the divide and conquer approach of *SPS* is beneficial given a limited execution time.

Our research focuses on solving a placement and selection problem using an ILP solver (CPLEX). We differentiate two techniques; the first method solves the entire problem in one run (*ILP-AS*). The second method attempts to solve our problem for one service at a time (*SPS*), keeping in mind placement decisions from previous runs. Both approaches must evaluate possible placements by finding the optimal selection and calculating the satisfied demand.

The *ILP-AS* solver has a global view of our problem, including all layers to be placed, the possible storage locations and servers where clients can connect from. As a result, the solver will always find the optimal solution for any given scenario, assuming the system we run it on has sufficient memory. *SPS* solves the same problem but separately for each service, without considering the services that still need to be placed and which layers they have in common with other services. Therefore, the solution of *SPS* will be the same as *ILP-AS* at best.

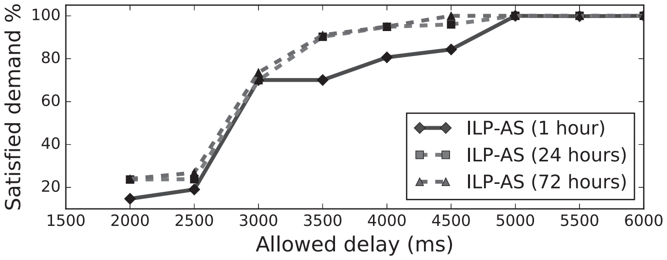


Fig. 5. The current best solution when pausing our *ILP-AS* solver after 1 hour, 24 hours and 72 hours. We evaluate how much our solution after 1 hour differs from the optimal solution, represented by a 72 hour evaluation. We observe a maximum performance increase of 20% for a 720% execution time increase. In this scenario, there are 25 client nodes, 25 services, 2 server and 2 storage nodes in this experiment. 20% of the service layers are reused by multiple services.

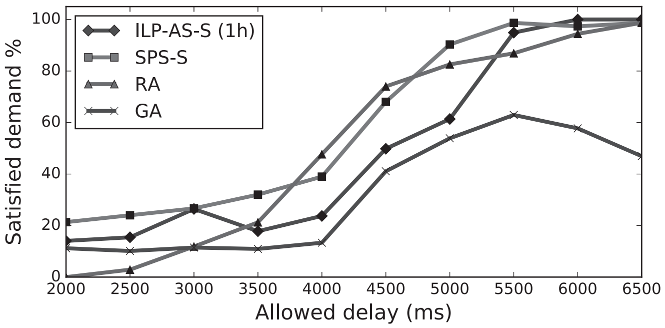


Fig. 6. The satisfied demand for the current best solutions of *ILP-AS (1 hour)*, *SPS*, *RA* and *GA*. There are 45 client nodes, 25 services, 2 server and 8 storage nodes in this experiment. 20% of the service layers are reused by multiple services. *SPS* performs best in most scenarios while *GA* satisfies the least demand.

Our experiments showed that, for small scenarios, *SPS* is close to the optimal solution found by *ILP-AS* but it does not always guarantee that this solution can be found, as explained in the example in Section V. For smaller scenarios where *ILP-AS* is able to solve the problem in a feasible time span, *ILP-AS* is always the recommended approach as it guarantees an optimal solution.

However, due to the complexity of the *ILP-AS* model, this approach scales poorly when the size of our problem increases. The system will lack the memory required to solve the problem or require a much longer execution time to find a solution. *SPS* only considers one service per run, reducing the amount of memory required in each run of our solver. As a result, this approach is able to find better solutions than *ILP-AS* given a limited execution time or system memory.

Our goal in this paper is to demonstrate that *SPS* can achieve better results than *ILP-AS* given a limited execution time, due to less system memory consumption in each run. As *ILP-AS* is not able to solve larger scenarios in a feasible time, we forcefully interrupt the *ILP-AS* solver after a certain execution time and retrieve the current best solution. We argue that, in the same time span, *SPS* will find a current best solution which satisfies more users due to solving much faster.

Before we proceed to compare *ILP-AS* and *SPS* in a certain time span, we evaluate the influence of the timing where we interrupt the *ILP-AS* solver. In Fig. 5 we observe the performance of *ILP-AS* when stopped after 1 hour, after 24 hours and 72 hours. The X-axis represents the allowed response time to satisfy a client. The Y-axis represents the percentage of satisfied clients in the current best solution for a desired response time. There are 25 client nodes, 25 services, 2 server and 2 storage nodes in this experiment. There is always enough storage capacity available to host each service at least once, meaning that we can reach 100% satisfied demand if the response time tolerance is high enough (increasing X-axis values).

Note that *ILP-AS* is not able to find a suitable solution for this problem in a feasible execution time without large amounts of memory. However, we observe that there is little satisfied demand gained by running the solver for 72 hours compared to only running it for 24 hours. We can assume that running the solver longer will see the same trend. Therefore, we approximate our optimal solution by our 72 hour curve. As such, the satisfied demand in Fig. 5 is not the final solution but only a current best after 1, 24 and 72 hours.

Considering the difference in allowed execution time of the *ILP-AS* solver, we observe a maximum satisfied demand increase of 20% for a 720% execution time increase. For a static environment this may be acceptable but when the demand is rapidly changing, waiting 72 hours may not be feasible. In these environments, using the solution obtained after 1 hour may be more suitable.

Therefore, in the remainder of our simulations we will assume that our solver is interrupted after 1 hour for each run (*ILP-AS (1 hour)*) and the presented satisfied demand is only a current best solution.

E. Reduced Resource Consumption

In this section we compare *SPS* to our benchmarks, including *ILP-AS* when the solver is interrupted after 1 hour (*ILP-AS (1 hour)*). Although *ILP-AS* would find an optimal solution if given infinite system memory and execution time, by interrupting the solver after 1 hour we can no longer guarantee an optimal solution. Our goal is to show that other methods such as *SPS* will find a better solution in the same time span due to improved resource consumption. All results were obtained by running a simulation on five different sample topologies with similar characteristics (see Table II) and averaging the results of each run.

Consider Fig. 6, showing the performance of both the *ILP-AS (1 hour)* and *SPS* approaches, as well as our two other benchmarks *RA* and *GA*. Once again, the X-axis represents the allowed response time to satisfy a client and the Y-axis represents the percentage of satisfied clients in our current best solution for the desired response time. There are 45 client nodes, 25 services, 2 server and 8 storage nodes in this experiment. The satisfied demand values represent the current best solutions found by interrupting the CPLEX solvers after 1 hour of execution time.

GA generally satisfies the least users. This is because this algorithm was not made for layered Docker services and expects all layers of a service to be placed on the same storage node. As a result, the services which are placed first will

TABLE I
PROBLEM STATEMENT VARIABLES

Parameter	**Description**
N	The union of all client, server and storage nodes.
I	The client node collection.
J	The server node collection.
K	The storage node collection.
S	The service collection.
L	The collection of layers.
L_s	The collection of layers *l* in service *s*.
$d_{i,j}$	Minimal latency between node *i* and *j*
$H_{i,j}$	Minimal hop count between node *i* and *j*
$D_{j,k,l}$	Download time between node *j* and node *k* for layer *l*.
C_k	Storage capacity on storage node *k*.
M_l	Storage capacity required by layer *l*.
D_s	Maximum tolerable delay for service *s* so clients will be satisfied.
T_l	Deployment time for layer *l*.
$Y_{k,l}$	DECISION VARIABLE: $Y_{k,l}$=1 if layer *l* is downloadable from storage *k*, else zero.
$P_{i,j,s,k,l}$	DECISION VARIABLE: $P_{i,j,s,k,l}$=1 if client *i* connects to server *j* to run service *s* and downloads layer *l* from storage *k*.
$Q_{i,j,s}$	DECISION VARIABLE: $Q_{i,j,s}$=0 (a logical solution) if the selection variable P assigned all layers of a service *s* to a client and that client only connects to one server *j* for all layers of that service. Else $Q_{i,j,s}$=1 denotes an infeasible solution.
$W_{i,s}$	DECISION VARIABLE: Parallel download time of server s for client i. This is the max download time of each layer in s through the selected server and storages. (cfr. section IV-B)
$Z_{k,l}$	Cumulative placement result from previous runs in *SPS* (cfr. section V)

TABLE II
SIMULATION VARIABLES

Description	**Value**
The average size of a service*	200MB
Layers per service*	15
Layers shared by multiple services*	20%
Available storage node capacity	2GB
Available server node capacity	600MB
Client-server latency*	Uniform [30,70] ms
Server-to-server latency*	Uniform [10,30] ms
Server-to-storage latency*	Uniform [50,100] ms

All values marked with * are obtained through measurements of real-life systems. For other (artificial) variables only their relative values are important.

occupy the capacity on nodes in their preferred location, forcing the next services to be placed in suboptimal locations where they cannot serve client requests in time. This is an example of how existing approaches cannot be applied to a Docker approach out of the box, and additional research is required.

Note that in Fig. 6 *GA* satisfies more clients for 5.5 seconds allowed delay than for 6.5 seconds. This is because *GA* will choose different storage locations for a service for different amounts of allowed delay (x-axis), always in a greedy manner. In this case, this greedy behavior happened to work out better for the total satisfied demand for 5.5 seconds allowed delay compared to 6 or more seconds. Additional experiments verified that for other simulation parameters (see Table II and Table I), the satisfied demand increased in function of the increasing allowed delay, as expected.

When comparing *ILP-AS (1 hour)* and *SPS*, we observe how *SPS* is generally able to find a solution which satisfies the most users. This is because our dataset in each *SPS* run is reduced and the solver requires less memory, allowing it to solve the problem faster than *ILP-AS*. As a result, *SPS* tends to find better solutions given a limited execution time (e.g., rapidly changing environments). Note that, if the solvers were not interrupted, that eventually *ILP-AS* would have found the optimal solution.

RA performs very well in the middle of the curve and has the additional benefit that it is easy to implement and executes very quickly. However, when looking at time-sensitive applications with very strict delay requirements (left side X-axis), *RA* performs the worst. Additionally, when we're looking for the method which can satisfy 100% of the users with the least delay (right side of X-axis), both *ILP-AS (1 hour)* and *SPS* would be better choices than *RA*. For example, in this scenario *SPS* can satisfy any client request within 5.5 seconds, while *RA* can only satisfy 85% of the requests with that delay requirement.

Although none of the approaches aim to minimize the used capacity, it is still interesting to evaluate how each method behaves. Providers could use this evaluation to decide which method is most cost efficient, comparing the required storage capacity with the percentage of satisfied users. Fig. 7 visualizes the used capacity for the same experiments as presented in Fig. 6. We observe how *RA* uses the same capacity regardless of the allowed delay or satisfied users. This is because *RA* does not consider the users during the randomized placement steps. As a result, *RA* is the least cost efficient method when considering storage costs.

GA shows a direct correlation between the satisfied users and the required capacity. This method only uses the capacity that it needs to satisfy users but fails to satisfy a large amount of users compared to the other benchmarks, as previously explained. By grouping all layers of one service on the same storage node, this method fails to make use of the reusability of layers.

ILP-AS (1 hour) and *SPS* show a slight correlation between the used capacity and satisfied demand but not nearly as strong as *GA*. Generally, an increase in satisfied demand will also see an increase in used capacity. This is because in these two approaches, the solver may deploy additional layer replicas to satisfy more clients. However, there are cases where the satisfied demand increases while the used capacity remains the same. This is because the increased allowed delay (X-axis) allows the solver to place layers on more distant storage nodes

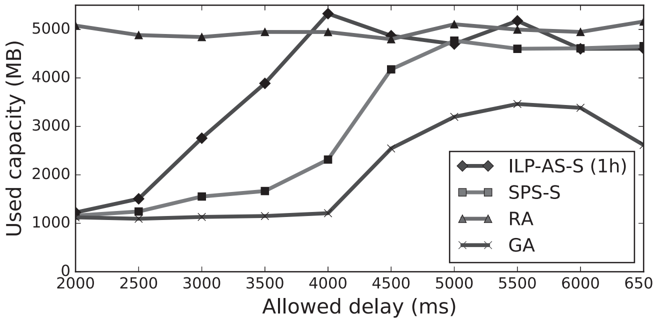


Fig. 7. The corresponding capacity used for the solutions presented in Fig. 6. *RA* consistently uses the most storage capacity while *GA* shows a direct correlation between the satisfied users (Fig. 6) and the required storage capacity.

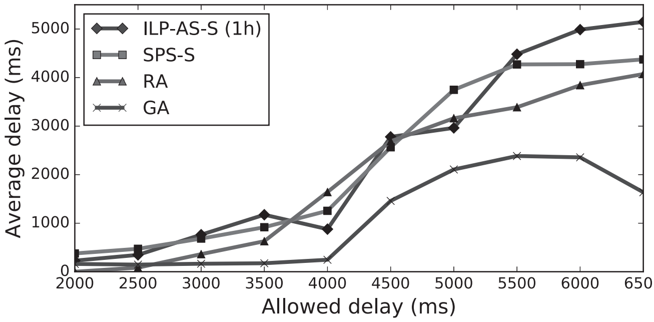


Fig. 8. The average delay for a satisfied client for the solutions presented in Fig. 6. *GA* shows a direct correlation between the satisfied users (Fig. 6) and the average delay. The remaining methods result in roughly the same delay.

while still satisfying clients, requiring fewer replicas to be deployed. In the end, both solutions will attempt to maximize the satisfied demand, regardless of the required capacity.

Fig. 8 visualizes the average delay for all satisfied clients in the same experiments as Fig. 6 and Fig. 7. We only display the average delay for the clients who can be served within the allowed delay (X-axis). The less strict our delay requirements become, the higher the average delay is allowed to be while still satisfying the clients. As a result, all curves with the exception of *GA* follow an increasing average delay trend. *GA* fails to satisfy clients because it places all layers of a service on the same storage node. This flaw is independent of the allowed delay, which is why we do not see an increase in satisfied demand or average delay when the allowed delay increases.

These simulations show that in rapidly changing environments (short execution times), *SPS* is able to find a solution that, in most scenarios, satisfies the most demand. Additionally, *SPS* is the second best performing method when it comes to capacity usage, only falling short to *GA* which in turn also results in the lowest amount of satisfied demand.

In the next section we compare these results with our parallel download model as introduced by Docker v2.0.

F. Serial Versus Parallel

Docker v2.0 came with new features which included parallel downloads of all service layers at once, greatly reducing the required time to download a service. If the download time decreases, the importance of the client-server delay increases. In this section we want to evaluate how the parallel download model introduced in Docker v2.0 influences the trends we noticed in the previous section.

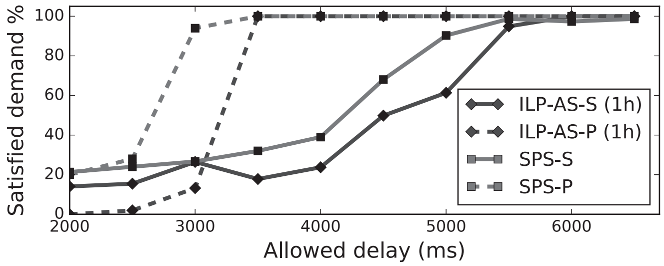


Fig. 9. The satisfied demand in function of the allowed delay for the same experiments as Fig. 6, now comparing *ILP-AS-S (1 hour)* with *ILP-AS-P (1 hour)* and *SPS-S* with *SPS-P* respectively. We observe that a parallel download model improves the satisfied demand since layers are retrieved quicker.

Consider Fig. 9 where the solid curves (*ILP-AS-S*and *SPS-S*) are the same curves as in Fig. 6, representing the satisfied demand assuming a serial download model. The dashed curves (*ILP-AS-P* and *SPS-P*) represent the corresponding satisfied demand for the same experiment when using the parallel problem statement from Section IV-B. In a parallel download model we require less time to pull all the layers of a service, so we expect to satisfy more clients than the serial counterpart.

Compare the dashed *SPS-P* (*ILP-AS-P*) curve with the solid *SPS-S* (*ILP-AS-S*) curve. We observe how the satisfied demand is generally higher for the parallel models, confirming that services in Docker v2 are downloaded faster in parallel and therefore more clients can be satisfied.

An important observation is that for *ILP-AS-P* the satisfied demand is lower than *ILP-AS-S* for an allowed delay smaller than three seconds. This is because our solver has more layer placements to consider within the allowed delay when assuming a parallel download model, requiring more execution time to find a solution. Since we interrupt the *ILP-AS-P* solver after 1 hour, the solver was not able to calculate a good current best solution yet. Note that if we allowed *ILP-AS-P* to run for more than 1 hour, it would definitely find a solution which satisfies the same or more users than *ILP-AS-S*, due to the relaxed layer download conditions.

As expected, these results confirm that a parallel download model improves the satisfied demand since layers are retrieved quicker. However, the relaxed delay constraint now results in increased complexity, as more storage nodes can be considered for a suitable placement.

After comparing the performance of both approaches, we now discuss an interesting use case how this solution can be used to find the minimal required resources for a desired satisfaction level, and thus the minimal cost.

G. Minimum Resource Requirement

After evaluating the effect of the execution time for both a serial and parallel download model, we now look at practical usages of our solution. A key objective for any provider is to

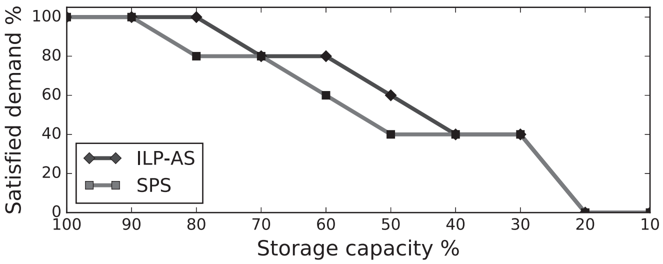


Fig. 10. The importance of the capacity on storage nodes (X-axis) for the satisfied demand (Y-axis). This experiment can be used to determine the minimal resources required to achieve a desired satisfaction level. There are 45 client nodes, 5 services, 2 server and 2 storage nodes in this experiment. 20% of the service layers are reused by multiple services.

minimize the cost, often offset against user satisfaction. Our goal is to demonstrate how our research can be used to solve this use case.

Consider Fig. 10, visualizing the satisfied demand (Y-axis) in function of the decreasing storage capacity (X-axis). There are 45 client nodes, 5 services, 2 server and 2 storage nodes in this experiment. Using this graph, we are able to determine the minimal resources required to achieve a desired satisfaction level. For example, assuming we provision 100% storage capacity, we can reduce the capacity to 90% and still maintain 100% user satisfaction. When using the *ILP-AS* approach we can even scale down to 80% capacity and maintain complete satisfaction. This experiment can be repeated and used to avoid overprovisioning.

Regardless of the scale of the problem and the desired method, this use case validates that providers are able to use this approach to find minimal resources required for a desired satisfaction level.

VIII. Conclusion

In this paper we presented a service placement method using the image layers introduced by Docker. By placing service building blocks called layers in strategic positions, we allow users to connect to a server, retrieve the necessary layers and deploy a service instance. We presented our layer placement problem statement for both a serial and parallel download approach. To reduce the required system resources, we introduce a search heuristic solving the problem one service at a time, considering previous decisions in each step. We evaluate the influence of the sorting order for our search heuristic and provide four different sorting methods. Simulation results show that the iterative search method finds a better solution in the same time span due to improved resource consumption. We explain how we obtained our simulation results which are based on real-life data obtained from crawling the Docker repositories and modeling real network distributions. Last, we introduce an interesting use-case to find the minimal storage resources necessary for a desired satisfaction level, avoiding any overprovisioning.

As conclusion we can say that our model is able to find a suitable layer placement to maximize the satisfied demand, using an on-demand deployment scheme. By reusing previously deployed layers, we avoid cluttering the network with layer replicas.

In future work we look at dynamic placement algorithms to adjust to the dynamic service demand. With our current method we are able to find the best placement for initial demand but changes are necessary when the available session slots on servers are occupied. We will develop placement and selection algorithms using frequent reports from servers and storages as input. This allows us to place data where it is most needed by analyzing demand patterns. We can also use this data to auto-scale server and storage resources based on the changing demand, optimizing the amount of clients we can serve without overprovisioning. Additionally, we will build larger evaluation scenarios and use emulation techniques [34] to evaluate our solutions. Recently, a number of frameworks have been presented in [35] for container orchestration on a cluster of low-cost devices such as Raspberry Pi. These platforms provide an excellent tool to evaluate our solutions in more realistic conditions.

Acknowledgment

This project was partly funded by the UGent BOF-GOA project “Autonomic Networked Multimedia Systems”.

References

[1] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for VM-based cloudlets in mobile computing,” *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct./Dec. 2009.

[2] J. Thönes, “Microservices,” *IEEE Softw.*, vol. 32, no. 1, p. 116, Jan./Feb. 2015.

[3] N. Dragoni *et al.*, “Microservices: Yesterday, today, and tomorrow,” in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017, pp. 195–216.

[4] C. Costache, O. Machidon, A. Mladin, F. Sandu, and R. Bocu, “Software-defined networking of Linux containers,” in *Proc. RoEduNet Conf. 13th Ed. Netw. Educ. Res. Joint Event RENAM 8th Conf.*, Sep. 2014, pp. 1–4.

[5] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, “Consolidate IoT edge computing with lightweight virtualization,” *IEEE Netw.*, vol. 32, no. 1, pp. 102–111, Jan./Feb. 2018.

[6] C. Anderson, “The long tail,” *Wired Mag.*, vol. 12, no. 10, pp. 170–177, 2004.

[7] A. Ojala and P. Tyrvainen, “Developing cloud business models: A case study on cloud gaming,” *IEEE Softw.*, vol. 28, no. 4, pp. 42–47, Jul./Aug. 2011.

[8] C. Weinhardt *et al.*, “Cloud computing—A classification, business models, and research directions,” *Business Inf. Syst. Eng.*, vol. 1, no. 5, pp. 391–399, Oct. 2009, doi: 10.1007/s12599-009-0071-2.

[9] C. Anderson, “Docker [software engineering],” *IEEE Softw.*, vol. 32, no. 3, pp. 102–c3, May/Jun. 2015.

[10] P. Smet, B. Dhoedt, and P. Simoens, “On-demand provisioning of long-tail services in distributed clouds,” in *Proc. IEEE/IFIP Netw. Oper. Manag. Symp. (NOMS)*, Istanbul, Turkey, Apr. 2016, pp. 1320–1323.

[11] A. Amiri, “Application placement and backup service in computer clustering in software as a service (SaaS) networks,” *Comput. Oper. Res.*, vol. 69, pp. 48–55, May 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S030505481500283X

[12] Q. Zhang, Q. Zhu, M. F. Zhani, R. Boutaba, and J. L. Hellerstein, “Dynamic service placement in geographically distributed clouds,” *IEEE J. Sel. Areas Commun.*, vol. 31, no. 12, pp. 762–772, Dec. 2013.

[13] J. Famaey, T. Wauters, F. De Turck, B. Dhoedt, and P. Demeester, “Network-aware service placement and selection algorithms on large-scale overlay networks,” *Comput. Commun.*, vol. 34, no. 15, pp. 1777–1787, 2011.

[14] M. Selimi, D. Vega, F. Freitag, and L. Veiga, “Towards network-aware service placement in community network micro-clouds,” in *Proc. Eur. Conf. Parallel Process.*, Springer, 2016, pp. 376–388.

[15] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag, and L. Veiga, "Practical service placement approach for microservices architecture," in *Proc. 17th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput. (CCGrid)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 401–410, doi: 10.1109/CCGRID.2017.28.

[16] W. Tärneberg *et al.*, "Dynamic application placement in the mobile cloud network," *Future Gener. Comput. Syst.*, vol. 70, pp. 163–177, May 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X16302060

[17] J. K. W. Yeoh and D. K. H. Chua, "Optimizing crane selection and location for multistage construction using a four-dimensional set cover approach," *J. Construct. Eng. Manag.*, vol. 143, no. 8, 2017, Art. no. 04017029.

[18] P. Bahrampour, M. Safari, and M. B. Taraghdari, "Modeling multi-product multi-stage supply chain network design," *Procedia Econ. Finance*, vol. 36, pp. 70–80, 2016.

[19] T. Wu, F. Chu, Z. Yang, and Z. Zhou, "A Lagrangean relaxation approach for a two-stage capacitated facility location problem with choice of depot size," in *Proc. IEEE 12th Int. Conf. Netw. Sens. Control (ICNSC)*, Apr. 2015, pp. 39–44.

[20] J. Li, F. Chu, C. Prins, and Z. Zhu, "Lower and upper bounds for a two-stage capacitated facility location problem with handling costs," *Eur. J. Oper. Res.*, vol. 236, no. 3, pp. 957–967, 2014.

[21] B. Gendron, P.-V. Khuong, and F. Semet, "Comparison of formulations for the two-level uncapacitated facility location problem with single assignment constraints," *Comput. Oper. Res.*, vol. 86, pp. 86–93, Oct. 2017. [Online]. Available: http://www.sciencedirect.com/science/articl/pii/S0305054817300540

[22] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over RaspberryPi," in *Proc. 18th Int. Conf. Distrib. Comput. Netw. (ICDCN)*, Hyderabad, India: ACM, 2017, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/3007748.3007777

[23] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in IoT context: Horizontal and vertical Linux container migration," in *Proc. Glob. Internet Things Summit (GIoTS)*, Geneva, Switzerland, Jun. 2017, pp. 1–4.

[24] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards QoS-aware fog service placement," in *Proc. IEEE 1st Int. Conf. Fog Edge Comput. (ICFEC)*, Madrid, Spain, May 2017, pp. 89–96.

[25] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand, "iFogStor: An IoT data placement strategy for fog infrastructure," in *Proc. IEEE 1st Int. Conf. Fog Edge Comput. (ICFEC)*, Madrid, Spain, May 2017, pp. 97–104.

[26] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*, vol. 4. New York, NY, USA: Wiley, 1972.

[27] J. Famaey, T. Wauters, F. De Turck, B. Dhoedt, and P. Demeester, "Towards efficient service placement and server selection for large-scale deployments," in *Proc. 4th Adv. Int. Conf. Telecommun. (AICT)*, Athens, Greece: IEEE, 2008, pp. 13–18.

[28] (2013). *ILab.T Virtual Wall | Internet Based Communication Networks and Services*. [Online]. Available: http://www.ibcn.intec.ugent.be/content/ilabt-virtual-wall

[29] (2017). *Docker Hub. Docker Inc*. [Online]. Available: https://hub.docker.com/

[30] F. L. Vilela *et al.*, "Design of network architecture for femto-cloud computing," document D22, TROPIC, Surrey, U.K., 2013.

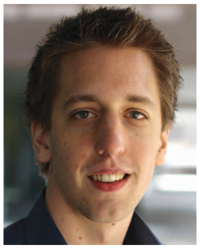
[31] *How Fast Is 4G? 4G Speeds and U.K. Network Performance*. Accessed: Mar. 4, 2018. [Online]. Available: http://www.4g.co.uk/how-fast-is-4g/

[32] A. Jain and J. Pasquale, "Internet distance prediction using node-pair geography," in *Proc. 11th IEEE Int. Symp. Netw. Comput. Appl. (NCA)*, Cambridge, MA, USA: IEEE, 2012, pp. 71–78.

[33] *Softlayer IP Backbone: Looking Glass*, SoftLayer Technol. Inc., Dallas, TX, USA. (2001). [Online]. Available: http://lg.softlayer.com/

[34] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft, "PiCasso: A lightweight edge computing platform," in *Proc. IEEE 6th Int. Conf. Cloud Netw. (CloudNet)*, Sep. 2017, pp. 1–7.

[35] L. Miori, J. Sanin, and S. Helmer, "A platform for edge computing based on Raspberry Pi clusters," in *Data Analytics*, A. Calì, P. Wood, N. Martin, and A. Poulovassilis, Eds. Cham, Switzerland: Springer, 2017, pp. 153–159.



Piet Smet received the M.Sc. degree in informatics from Hogeschool Ghent, Belgium, in 2012 with a thesis on the development of a large-data social media game. He is currently pursuing the Ph.D. degree under the supervision of B. Dhoedt and P. Simoenson focusing on service-centric networking. He was also active as a Researcher on the European Research Project H2020 FUSION.

Bart Dhoedt received the master's degree in electro-technical engineering and the Ph.D. degree focusing on the use of micro-optics to realize parallel free space optical interconnects from Ghent University in 1990 and 1995, respectively. He was a Post-Doctoral Fellow in opto-electronics with Ghent University for two years, where he became a Professor with the Department of Information Technology. He has authored or co-authored over 300 publications in international journals or conference proceedings.

Pieter Simoens received the M.Sc. degree in electronic engineering and the Ph.D. degree from Ghent University, Belgium, in 2005 and 2011, respectively. He was funded by the Fund for Scientific Research Flanders for his Ph.D. research. In 2012, he was a Visiting Researcher with the School of Computer Science, Carnegie Mellon University, USA. He is currently an Assistant Professor with the Department of Information Technology, Ghent University and iMinds. His main research interests include mobile cloud offloading, service-oriented networking, edge/fog computing paradigms, and service engineering for advanced mobile applications. He has authored and co-authored over 70 papers published in international journals or in the proceedings of international conferences in the above areas. He has also been involved in several national and European research projects (FP6 MUSE, FP7 MobiThin, and H2020 FUSION).