Scalable Linux Container Provisioning in Fog and Edge Computing Platforms

Michele Gazzetti[1(✉)], Andrea Reale[1], Kostas Katrinis[1], and Antonio Corradi[2]

[1] IBM Research, Dublin, Ireland
michele.gazzetti1@ibm.com
[2] Department of Computer Science and Engineering,
University of Bologna, Bologna, Italy

Abstract. The tremendous increase in the number of mobile devices and the proliferation of all kinds of new types of sensors is creating new value opportunities by analyzing, developing insights from, and actuating upon large volumes of data streams generated at the edge of the network. While general purpose processing required to unleash this value is abundant in Cloud datacenters, bringing raw IoT data streams to the Cloud poses critical challenges, including: (i) regulatory constraints related to data sensitivity, (ii) significant bandwidth costs and (iii) latency barriers inhibiting near-real-time applications. Edge Computing aspires to extend the traditional cloud model to the "edge of the network", to deliver low-latency, bandwidth-efficiencies and controlled privacy. For all the commonalities between the two models, transitioning the provisioning and orchestration of a distributed analytics platform from Cloud to Edge is not trivial. The two models present totally different cost structures such as price of bandwidth, data communication latency, power density and availability. In this paper, we address the challenge associated with transitioning scalable provisioning from Cloud to distributed Edge platforms. We identify current scalability challenges in Linux container provisioning at the Edge; we propose a novel peer-to-peer model taking on them; we present a prototype of this model designed for and tested on real Edge testbeds, and we report a scalability evaluation on a scale-out virtualized platform. Our results demonstrate significant savings in terms of provisioning latency and bandwidth utilization.

1 Introduction

The number of devices connected to the Internet has registered a steady increment, and the 6 billion things connected today define an ecosystem of objects called "Internet of Things" (IoT). An increasing number of industries is betting on IoT as a way to boost efficiency and explore new business models through better real-time insights on their processes. While various new paradigms are emerging to support and make this vision a reality, Edge Computing, which enables the placement of services directly at the edge of the network, is a very promising one. Edge Computing augments the traditional cloud model, by allowing to create new latency/privacy sensitive services and, at the same time, lowers

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 304–315, 2018.
https://doi.org/10.1007/978-3-319-75178-8_25

operational costs by reducing communication between devices and remote backends. Following well consolidated industrial practices adopted in Cloud computing, virtualization approaches based, for example, on hypervisor-governed virtual machines [11] and Linux containers [12], have been proposed as the execution environment of choice for Edge computing answering the common requirements of resource isolation and dependency management.

Still, the new paradigm does not come without new challenges, including scalable distribution and update of applications across large Edge/IoT deployments. In this paper, we attempt to address this challenge in an Edge computing environment employing Linux containers as application distribution and execution unit. In Sect. 2, we introduce a baseline (best-practice to date) method to deploy Linux containers on the Edge and discuss its scalability challenges. To overcome them, we present in this paper a distributed streamed deployment approach. Our method leverages the inherent layered structure of container images and filesystems to develop a peer-to-peer provisioning protocol that improves latency at scale, while conserving on Edge-Cloud bandwidth costs.

We have implemented our approach on target edge devices (NVidia Tegra, ARMv7 Raspberry Pi) as a proof of viability. To showcase the promise of the approach at scale, we have used the device-based results to calibrate Virtual Machines (VMs) executing Edge device operating system and applications. We obtained results on up to 21 Edge nodes running on this virtual environment, showing an up to 3 times improvement in container provisioning time within each locality and up to 10 times reduction of Edge-Cloud bandwidth utilization, when compared to provisioning a locality from a single centralized container image registry.

2 Motivation: Naive Edge Container Provisioning

The Edge computing model we assume in this paper is the following. As shown in Fig. 1, Edge devices are grouped in "localities". Nodes within a locality are horizontally interconnected at lower latency and cost of bandwidth, when compared to the network link used for Edge-Cloud communication. We assume Edge nodes to be executing on embedded/microserver devices, running a general purpose operating system. Without loss of approach applicability to other operating systems and virtualization approaches - as long as a layered structuring of deployment images can be inferred - we focus our presented work to devices using Linux and Docker containers as the runtime for Edge applications.

Docker [1] defines a container as a runtime instance of a Docker image. An image is an ordered collection of changes compared to the initial filesystem representing the base of the image. We can think of an image as a set of layers stacked on top of each other to form the container filesystem. To facilitate image sharing and streamlining of container provisioning, Docker provides a dedicated image repository and server called Registry. The Registry is a stateless, highly scalable server side application that stores Docker images and responds to requests for deployment from remote nodes. This component of the architecture is usually

running within a (Cloud) datacenter, receives pull requests and responds with the requested image. As part of this deployment procedure, the layers comprising an image are combined into a single archive binary file and then transmitted to the host where the container image is instantiated.

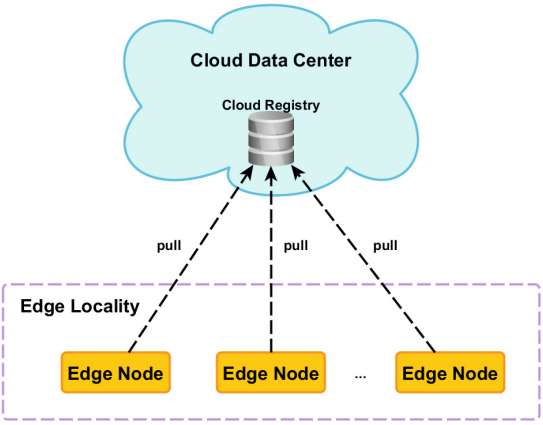

Fig. 1. Edge computing model with nodes in each locality pulling container images from a centralized Docker Registry

Applying the above baseline best practice approach ("naïve") for container provisioning at the Edge leads to multiple pull requests coming from different nodes of the same locality. This is inefficient in terms of cost of bandwidth, as it entails avoidable exchange of redundant data for the same location. Moreover, this approach can lead to increased load at the Registry and thus decreased service quality, to the extent that the Registry itself could rapidly become a bottleneck. The latter effect can specifically occur when a large number of edge nodes concurrently issue download requests for an image, for example, when updated image versions become available. One solution that can limit the load on the registry is clustering or geo-distribution, which though again clearly raises the cost of the service.

The graph presented in Fig. 2 provides a visual representation of the problem, showing the time to deploy a container image of 500 MB on a varying number of concurrently provisioned edge nodes. The concurrent scenario represents the worst-case in terms of provisioning overhead, for reasons outlined above. The data represented in the graph is derived assumming 200 Mb/s bandwidth between the Cloud Registry and the Edge locality and Edge nodes with 100 Mb/s network interfaces. Figure 2 shows that the time taken to deploy a new container image in a locality increases linearly: while a single node requires only 40 s to pull the image, provisioning 10 nodes saturates the available Edge-Cloud bandwidth. In case of 1000 nodes, provisioning takes more than 5 h.

Mirroring the remote Registry within each locality would help improving the scalability of the solution and reduce provisioning latency; however, such an

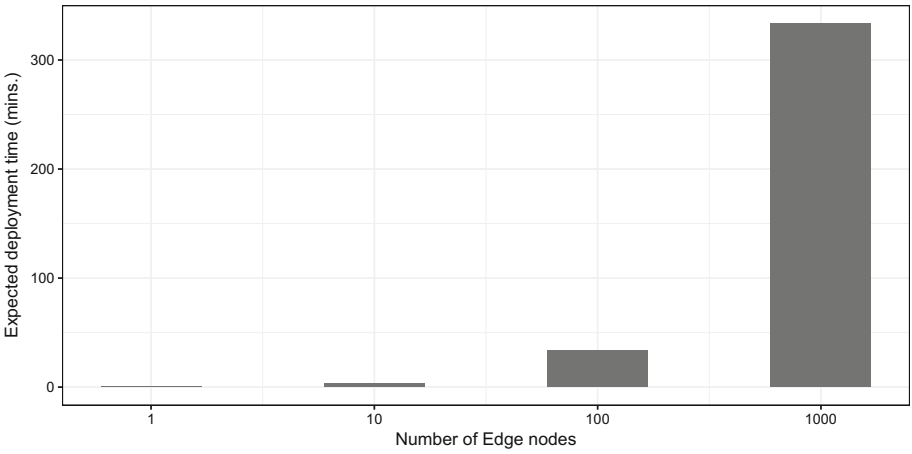

Fig. 2. Estimated time to fully provision a locality with a container image of 500 MB as the number of Edge nodes in the locality increases.

approach would suffer from following shortcomings: (a) cost of bandwidth would still be very high, due to continuous syncing between edge and cloud registries, especially if there is a large set of discrete images deployed in a locality and (b) the typically limited processing capability and storage capacity of Edge devices would be an important limiting factor for them to act as local Registries, given the average size of container images and the potential size in terms of devices of a locality. Beyond the technical challenges, there are also business barriers in following such an approach, since, in some deployments, the Edge premise may not be under the control of the Registry/Cloud providers.

3 Streamed Container Deployment

To overcome the complexity of distributing the image within a locality without creating bottlenecks, we propose a paradigm shift toward a novel peer-to-peer provisioning approach where nodes in a locality co-operate to accelerate provisioning. We call this approach "Streamed Deployment" (SD). Figure 3 shows the block diagram of the components involved in our streamed deployment approach. To provision an image within a locality, one of the Edge nodes is elected as entry point to the Edge-Cloud network. This node (termed "Gateway") interacts with the remote Cloud infrastructure, pulls the image on behalf of the entire locality and provides information regarding the status of the nodes within it. Within the Gateway, the Gateway Manager (GM) dynamically manages the formation of a peer-to-peer distribution graph within the locality. This includes the dynamic repair of the distribution topology in the case of node timeouts (due to, e.g., failures or in case of mobile nodes exiting a locality). The coordination between the various nodes within the locality occurs via a Message Broker (MB, in our case

realized through a stock MQTT broker), a PUB/SUB broker capable of decoupling sender and receiver through asynchronous messaging. Last, each edge node carries an implementation of a Stream Manager (SM), an agent that implements the real-time container image streaming protocol on top of the formed peer-to-peer distribution graph. Also, the SM interacts with the Docker daemon on each node to import received image layers to the local image store that each Docker Edge instance maintains.

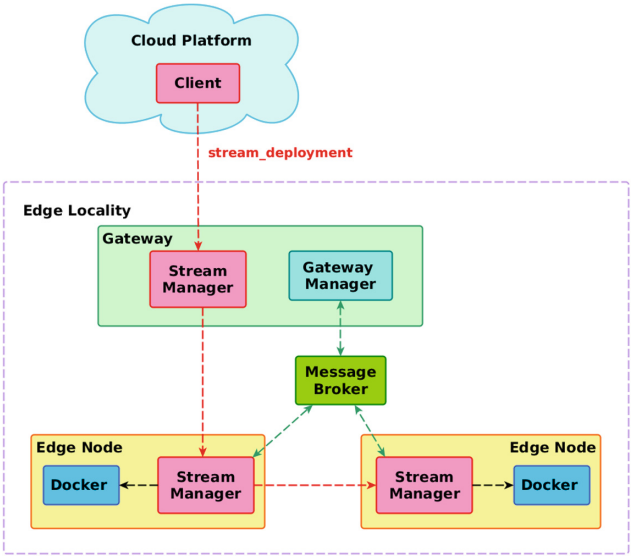

Fig. 3. Architecture of our Streamed Deployment implementation.

Figure 4 provides a more comprehensive representation of the interactions performed during the deployment. There are three actors involved in the depicted workflow: the formerly described Gateway Manager of the locality, the Stream Managers of each Edge node being provisioned (only one instance of the SM is shown for brevity), and the client initiating the deployment. Typically, the client would be situated in a remote location relative to Edge localities, e.g., within an Edge orchestration entity running in the Cloud as part of an integrated IoT platform solution. In this case, the client contacts the edge Gateway to request the deployment of a new container image including ancillary information of the image that needs to be deployed (image identifier and composing layers). The SM on the Gateway responds with an ID that uniquely identifies the deployment procedure and the list of layers to be pulled. If a subset of the requested layers is already present within the locality, the SM requests to pull only the differences between the received list of deployment layers and the ones that are already stored in the locality. After this handshaking phase succeeds, the GM establishes the peer-to-peer distribution topology within the locality describing the

communication chains among Edge nodes. Depending on different optimization objectives, different algorithms can be used to build this tree. As this aspect is not central to our method, and for reasons of space, we do not discuss it further in this paper, but we refer the interested reader to the many existing solutions described in the literature [7,8,10]. Finally, the differential container image is streamed through the distribution topology to the edge nodes within the locality, thus getting all nodes eventually provisioned.

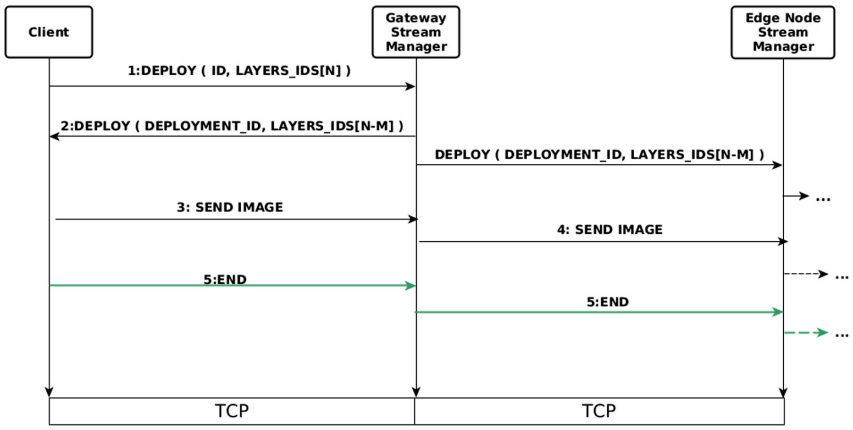

Fig. 4. Workflow of a Streamed Deployment within an edge locality

We now elaborate on the streamed forwarding procedure that implements our approach. The procedure is implemented within the Stream Manager, implemented as an application server running on each Edge node and serving deployment requests coming from peers. The algorithm executed by the SM is summarized in the pseudocode listing of Algorithm 1. The algorithm is executed by the Gateway in response to client deployment requests and, symmetrically, by Edge nodes in response to subsequent requests by peer nodes.

Upon its submission, the deployment request is first handled by the Gateway. After constructing a distribution topology (line 2), it extracts information about the layered image to deploy from the request (line 3). This information is used to compute the difference between the set of layers in the image to deploy and the layers already available within the locality (lines 4–6); the result is sent back to the client which will start streaming only missing layers.

The rest of the algorithm is executed simultaneously and identically by the Gateway and all the other Edge peers in the locality. Following the distribution topology computed (Gateway) or received (Edge device), each peer receives from its "parent" in the topology the missing layers and concurrently streams them to its "children" peers (lines 11–15). Our implementation is based on an in-memory Pipe data structure to which image data can be streamed in and out concurrently. The Pipe abstraction takes care of establishing and keeping connectivity with

one peer's children (we used TCP connections for that) and of asynchronously saving the received image on secondary storage. Once the full content of the image is received, the SM instruct the local Docker daemon to import the image from disk (line 15).

Algorithm 1. Forwarding Mechanism

```
   input: parent, deploymentRequest
 1 if isNodeGateway() then
 2 |   distrTopology ←computeTopology ();
 3 |   layers ←getLayers (deploymentRequest);
 4 |   installedLayers ←getInstalledLayers ();
 5 |   commonLayers ←intersection (layers, installedLayers);
 6 |   layersToDeploy ←difference (layers, commonLayers);
 7 |   send (parent, layersToDeploy);
 8 else
 9 |   distrTopology ←getTopology (deploymentRequest);
10 end
11 children ←getReceivers (distrTopology);
12 pipe ← createPipe ();
13 startReader (parent, pipe);
14 startWriter (children, pipe);
15 importImage ();
```

4 Evaluation Results

To prove the feasibility of our approach, we developed a prototype implementation of the Streamed Deployment architecture and implementing logic, as shown in Fig. 3 and outlined in the previous section. We deployed the prototype on representative Edge/microserver boards, specifically on an NVidia Tegra TK1 development board (acting as locality Gateway) and a Raspberry-Pi 2 board (acting as Edge node), and successfully tested the prototype, demonstrating correct and efficient chained deployment from a centralized Docker registry.

To evaluate the proposed solution at higher, more realistic, scale, we created a virtual Edge locality leveraging a set of virtual machines (VMs) running on a fully dedicated Openstack [4] private Cloud hosted at IBM Research. Our deployment consists of one VM acting as Gateway and a variable number of VMs acting as Edge nodes. The resulting testbed features a cluster of 11 bare-metal servers and up to 22 VMs running on these servers. We provisioned each VM with 2 virtual CPUs, and 2 GiBytes of DRAM. We also limited the network interface throughput of each VM to 100 Mbit/s, so as to emulate the nominal bandwidth available on typical edge nodes. In order to make it easier to reason about the collected the results, our Streamed Deployment experiments assume a linear distribution tree, where each node has one parent and one child only.

To demonstrate the improvement in terms of deployment latency within a locality, we executed a set of experiments where all nodes within the locality are

concurrently deploying a specified container image. For that, we chose to deploy a popular media server image, namely the Plex Media Server [5] - the back-end media server component of Plex. This image was chosen because representative (especially in size) of a large class of multimedia applications that might be running on an Edge locality. In each experiment execution, we vary the number of nodes in the locality from 1 up to 21 nodes. With this configuration, we made two groups of experiments, using either the Streamed Deployment (SD) approach or the baseline approach outlined in Sect. 2.

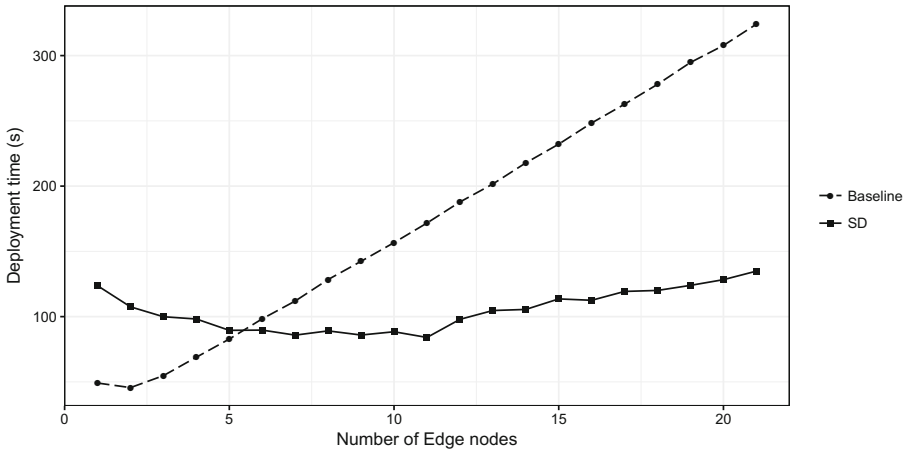

Fig. 5. Time taken to deploy a Plex Docker container within a locality with an increasing number of edge nodes

Figure 5 reports the results of the time taken to deploy a Docker container within a locality using the two approaches, versus an increasing number of Edge nodes. We observe that in a locality with less than five nodes, the baseline approach (centralized Docker Registry) provides faster deployments compared to the proposed solution, because of the co-ordination overhead in the Streamed Deployment. However, as the locality size increases, our approach yields faster deployments compared to the baseline. At the largest scale tested (21 nodes), our approach is 3 times faster compared to the baseline. While continuing the evaluation to larger locality sizes is part of our on-going work, we don't have a reason to expect that the shown trends will change: while the naive registry baseline has a steep linear scaling pattern, our approach exhibits a much more gradual linear increase pattern, fit for much larger scale localities.

In addition to provisioning latency, another important factor in Edge Computing deployments is bandwidth utilization. Figure 6 reports the network throughput time-series on the downstream direction of the Edge-Cloud link. The Edge Gateway pulls the image from the registry only once for the entire locality. Therefore, the amount of data that needs to be sent to the locality does not

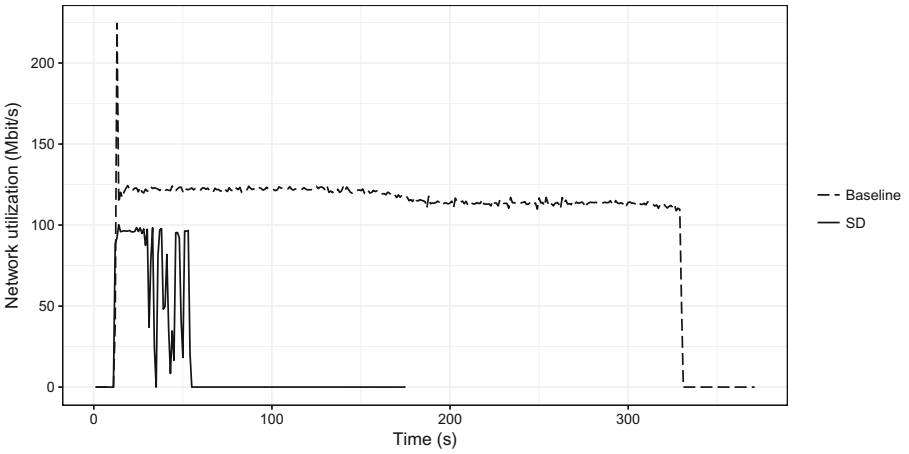

Fig. 6. Bandwidth utilization during container deployment on the Edge-Cloud link.

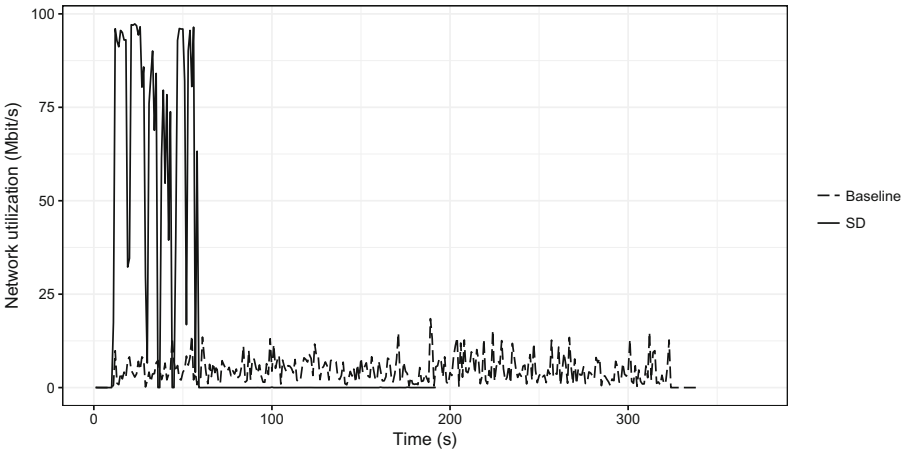

Fig. 7. Bandwidth utilization on one edge node within the locality during container deployment.

increase with the number of nodes, resulting in reduced bandwidth utilization in both size and time. The same cannot be said for the baseline approach: as the Docker Registry is the only image provider for the 21 nodes, there is a lengthy and steady utilization of the Edge-Cloud link, until the separate deployment on each locality node completes.

Although Edge-Cloud bandwidth is, in general, the most expensive network resource, making efficient use of local bandwidth might also be critical, especially in scenarios where local connectivity is also being used for application traffic (e.g., real time data-communication among Edge nodes). We evaluated how the Streamed Deployment uses this resource and we show the results in Fig. 7, which

depicts the network utilization of an Edge node while receiving an image as a time-series. The graph clearly highlights how, in the baseline approach, the local link is underutilized during image deployment, due to the bottleneck effect of the Edge-Cloud link being shared among all the local nodes (21 in our experiment). On the contrary, Streamed Deployment makes full use of the 100 MBit/s local link, leading to faster deployment and, in general, to better network utilization.

5 Related Work

While commercial and enterprise deployment of Internet of Things is a reality, to date the vast majority of roll-outs has either employed very thin general purpose computing on the edge (e.g. filtering/aggregation/sampling of sensor time-series) or highly specialized - both software- and hardware-wise - processing that is monolithically designed and usually tied to a specific solution (e.g. signal processing for speech recognition [9]). There are several standardization efforts aspiring to develop consensus on an edge/fog computing reference architecture [3] and its constituent layers [2]; also, ample research efforts have experimented with various challenges, among others node roles and node architecture [6,14,15], end-user value exploration [13] and customized storage/data models [16].

As the field of general purpose distributed, potentially multi-tenant, computing and analytics at the edge of the Internet of Things is only nascent, the vast amount of prior art has focused on architectural exploration, with no special focus on addressing provisioning and infrastructure/platfrom management challenges, more so from the perspective of massive scalability. Pahl and Lee [12] have discussed the fitness of Linux containers as the execution unit in edge deployments versus hypervisor-based virtualization; major advantages of containers are typically their lightweight footprint, performance and native support for microservices. Early results presented by the Superfluid Cloud [11] indicated that customization of virtual machines resp. hypervisors (Xen) can yield provisioning latency results at large scale that are comparable to those of LXC containers. It must be noted though that these results have been obtained on a mid-range datacenter-grade server (64-core x86-64 with 128 GB DRAM) and it remains to be seen how the two technologies compare against each other in terms of provisioning/footprint, when tested on low-power microservers and embedded devices. The latter are typically much more highly candidate to host edge/fog computing nodes in large-scale, distributed deployments. For the same reason, it is impractical to put the findings of [11] in perspective to our findings, as this paper has focused its value on addressing Xen virtualization optimizations for edge computing purposes. Instead, we focus on provisioning techniques of stock Linux containers, whereby we evaluate our approach in a full-fledged distributed setting, incorporating edge-cloud bandwidth and latency, compute/memory/storage capabilities that are representative of an edge gateway/node (microserver) and a remote centralized image repository.

6 Conclusions

Container-based virtualization techniques are being commonly accepted as a solution to support packaging, deployment and execution of applications on Edge/Fog computing deployments. In this paper, we have discussed the challenges in provisioning containerized applications to large numbers of Edge nodes, especially in terms of scalability of deployment latency and bandwidth utilization. We have shown that baseline/standard methods for container provisioning directly derived from Cloud best-practices are not suitable to be used unmodified in Edge scenarios, where bandwidth can be limited and more expensive (e.g., if based on cellular connectivity).

We have therefore presented the design and prototype implementation of a novel approach that addresses these problems, called Streamed Deployment. Based on a simple peer-to-peer data distribution model, our approach distributes the cost of container image provisioning across all the interested nodes within an Edge locality. Our evaluation on a scale-out testbed shows that Streamed Deployment provides up to a threefold deployment speed-up and a tenfold reduction on the utilization of the expensive Edge-to-Cloud network link.

While our solution improves provisioning speed and cost, it also creates new complexities and challenges if considering aspects like security and high-availability. Future work will investigate solutions to guarantee secure authentication and data exchange between all the involved actors, and protocols to guarantee deployment success despite dynamic topology reconfigurations and failures. Furthermore, we are extending our evaluation results to more realistic environments where edge devices are distributed across multiple locations. We also plan to evaluate the impact of different levels of Registry replication on system performance and reliability. These additional experimental results will provide a better understanding of the overall performance of the two approaches, especially for scenarios featuring a large number of devices.

References

1. Docker. https://www.docker.com/
2. Open edge computing. http://openedgecomputing.org
3. Open fog consortium. https://www.openfogconsortium.org
4. Openstack open source cloud computing software. http://www.openstack.org
5. Plex media server. https://github.com/greensheep/plex-server-docker-rpi
6. Chandra, A., Weissman, J., Heintz, B.: Decentralized edge clouds. IEEE Internet Comput. **17**(5), 70–73 (2013)
7. Chen, K., Nahrstedt, K.: Effective location-guided tree construction algorithms for small group multicast in MANET. In: Proceedings of Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 3, pp. 1180–1189 (2002)
8. Hosseini, M., Ahmed, D.T., Shirmohammadi, S., Georganas, N.D.: A survey of application-layer multicast protocols. IEEE Commun. Surv. Tutor. **9**(3), 58–74 (2007)

9. Jones, A., Benton, M.: Amazon Echo: A Simple User Guide to Amazon Echo and Essential Hacking Guide, vol. 6. CreateSpace Independent Publishing Platform, USA (2016)
10. Kim, M.S., Lam, S.S., Lee, D.Y.: Optimal distribution tree for internet streaming media. In: Proceedings of 23rd International Conference on Distributed Computing Systems, pp. 116–125, May 2003
11. Manco, F., Martins, J., Yasukata, K., Mendes, J., Kuenzer, S., Huici, F.: The case for the superfluid cloud. In: Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing, HotCloud 2015, p. 7. USENIX Association, Berkeley (2015)
12. Pahl, C., Lee, B.: Containers and clusters for edge cloud architectures - a technology review. In: 3rd International Conference on Future Internet of Things and Cloud, pp. 379–386, August 2015
13. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The case for VM-based cloudlets in mobile computing. IEEE Pervasive Comput. **8**(4), 14–23 (2009)
14. Tong, L., Li, Y., Gao, W.: A hierarchical edge cloud architecture for mobile computing. In: IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications, pp. 1–9. IEEE, April 2016
15. Zachariah, T., Klugman, N., Campbell, B., Adkins, J., Jackson, N., Dutta, P.: The internet of things has a gateway problem. In: Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications - HotMobile 2015, pp. 27–32. ACM Press, New York (2015)
16. Zhang, B., Mor, N., Kolb, J., Chan, D.S., Lutz, K., Allman, E., Wawrzynek, J., Lee, E., Kubiatowicz, J.: The Cloud is Not Enough: Saving IoT From The Cloud (2015)