Re-designing Cloud Platforms for Massive Scale using a P2P Architecture

João Soares*, Fetahi Wuhib*, Vinay Yadhav*, Xin Han†* and Robin Joseph‡*
*Ericsson Research, Stockholm, Sweden
Email: {joao.monteiro.soares, fetahi.wuhib, vinay.yadhav, robin.r.joseph}@ericsson.com
†Chalmers University of Technology, Gothenburg, Sweden
Email: hxin@student.chalmers.se
‡Blekinge Institute of Technology, Karlskrona, Sweden

***Abstract*—Cloud platforms need to scale with the number of resources and users they manage, while maintaining the needed performance levels with respect to service parameters such as application deployment time, service availability and response time. With the increase in capacity of todays data centers and distributed cloud deployment scenarios like edge computing, the scalability requirements of a cloud management software has become paramount. Most commercially available cloud management software are centralized from a software architecture point of view, and this often poses a limit on the number of resources that the software can manage while maintaining an acceptable performance level. In this paper, we present a generic design based on peer-to-peer (P2P) architecture for scaling cloud management software. We demonstrate how this design can be realized by applying it to the OpenStack cloud management platform. Our implementation alters neither the core functionality of OpenStack nor the way users interact with OpenStack. Performance evaluation results of the implementation show that, when used in a large system under high load, the solution greatly improves VM boot up times and VM startup success rates.**

I. Introduction

Cloud computing today is a ubiquitous technology enabling services to be offered over the internet in a cost effective manner. Cloud computing has commoditized computing and service delivery making it possible for enterprises to run their processing and service delivery tasks inside a data center in a scalable manner and just pay for what they use. Data centers essentially are composed of two major components, the hardware (servers, networking etc.) and the cloud management software. Cloud management software is responsible for managing the hardware and deployment of services inside a data center, this includes the servers, networking, storage, access control, resource scheduling, monitoring, and interfacing with user among others.

Cloud management software needs to scale with the number of resources and users it manages while maintaining the needed performance metrics such as service deployment time, availability and response time. With the increase in capacity of data centers and distributed cloud deployment scenarios like edge computing, the scalability requirements of a cloud management software become essential.

Most commercially available cloud management software are centralized from a software architecture point of view, and this often poses a limit on the number of resources that the software can manage while maintaining reasonable performance. For example, OpenStack, an open source cloud management software that has seen wide adoption has been shown to have scaling issues in some of its core components like scheduling and service APIs when the deployment reaches around 1000 server [1]. Another open source software for automated deployment and management of containerized applications, Kubernetes, does not claim scalability beyond 5000 nodes under a single kubernetes cluster [2]. It is clear that trying to manage large number of resources in a cloud deployment with a single instance of cloud management software does not scale well.

In this paper, we present a peer-to-peer (P2P) model of cloud deployment for scaling of cloud management software. The solution proposes the management of small pools of resources under a cloud management software instance which peer with other similar pools of resources to collectively in a P2P manner manage the entire cloud deployment. The proposed solution abstracts multiple cloud management software instances as a single instance. The proposed solution also suits very well for distributed cloud deployment scenarios such as edge cloud [3] where the cloud deployment consists of large number of geographically distributed datacenters.

The solution presented in this paper can be used to scale any cloud management software. In our work we present an implementation of our solution by applying it to scale OpenStack. Our implementation does not alter the core functionality of OpenStack and also does not alter the way users interact with OpenStack. We also present a performance evaluation of our solution illustrating the improvement of OpenStack scalability.

The rest of this paper is organized as follows. Section II presents various methods used to scale software sytems with particular focus on cloud management software. Section III provides an overview of OpenStack's architecture. The solution proposed in this paper is described in section IV, and performance evaluation results are presented in section V. Finally, section VI provides a set of concluding remarks and future work items.

II. Background

This section provides an overview of different solutions used to scale software systems. Moreover, it particularly discusses existing solutions available, or under development, that allow cloud deployments to scale.

A. Scaling Software Systems

Software systems (e.g., cloud platforms) are often designed following the centralized paradigm whereby management functions are implemented to run one node (often called the controller node) in the system. The centralized paradigm has the advantage in that it is often the simplest to implement. However, centralized systems have well-known problems that relate to scalability (i.e., become a performance bottleneck) and robustness (i.e., become a single point of failure). When a function of some software system needs to scale beyond a single node, there are three common approaches to do it. All of these approaches introduce complexities of varying degree to the implementation of the management function.

1) Cluster-based Solutions: The simplest place to start is to re-implement the management function in a clustered-way whereby an instance of the management function is run on several nodes that are placed behind a load balancer. This solution works very well for those scenarios where the management function is stateless (i.e., each request can be handled by any of instances). However, in situations where the functions are stateful or in scenarios where the instances need to share some resource, the solution becomes more complex and the load balancer or the shared resource become centralized again. Most OpenStack services (e.g, nova-api, keystone, glance-api, neutron-server) are implemented following this approach.

2) Hierarchical Solutions: When the need to scale beyond a single cluster arises for large-scale software systems, one approach is to organize the management nodes in some form of tree/hierarchy and distribute the computation associated with the management function across the nodes of the tree. Solutions for the situation where the tree is configured statically and different roles are manually assigned to the nodes (e.g., see OpenStack cells below) often are simpler to realize and implement. On the other hand solutions where the tree is configured dynamically and nodes have identical roles (e.g., ARTEMIS [4], GAP [5]) tend to be more complex. A key drawback of hierarchical/tree-based solutions is that they tend to be brittle (i.e., a failure of a node makes an entire subtree unavailable) and the root may still end up being a bottleneck. (Some of the above drawbacks may be mitigated through some form of redundancy, which comes at the cost of more complexity in design.)

3) Peer-to-Peer Solutions: An alternative approach to implement scalable management functions is the use of peer-to-peer (P2P) techniques. In P2P all nodes are treated equal and any node is able to receive and process the request passed on to it. P2P systems are boradly divided in to structured systems (e.g., DHTs [6]) and unstructured systems (e.g., gossip-based systems [7], [8]. Though P2P systems are often the most complex to design and analize, they also are highly scalable and very robust to failures.

B. Scaling Cloud Deployments

Most cloud platforms in the public domain follow the centralized management paradigm where control functions are centralized to a few 'controller' nodes in the system (e.g. OpenStack, OpenNebula [9], Kubernetes [10]). OpenStack, as the IaaS de facto cloud standard platform, has been pushed to support cloud deployments at large scale both from a centralized cloud deployment perspective and from a distributed cloud perspective. Typical deployment alternatives rely on the OpenStack concepts of Regions and Availability Zones. A Region has its own full Openstack deployment, including API endpoints, networks and compute resources. Regions are used for multiple site deployments, where resources are scheduled to a particular site. However, there is no coordenation between regions. Availability Zones allow the logical seperation of compute nodes to address cases like physical isolation and redundancy (e.g. power, network).

Another deployment alternative is OpenStack Cells [11]. Cells allows scaling through the division of multiple compute node installations into 'cells'. Cells are configured in a tree-fashion way, where the top-level cell hosts a compute service API and a scheduler to forward requests to the cells below (child cells). Each child cell can be seen as a normal OpenStack compute deployment in that each cell has its own host scheduler, database server and message queue broker. By attenuating the pressure on queues/database at scale, cells allows OpenStack to scale beyond its standard deployment. However, besides the experimental nature of Cells, there is a lack of awareness of cells in other OpenStack services (e.g. networking, storage) which drastically limits functionalities. Moreover, Cells breaks other OpenStack compute service principles like Security Groups and Host Aggregates.

Other OpenStack alternatives are being explored. [12] proposed a cascading solution, where a parent OpenStack orchestrates a set of child OpenStack instances. However, this solution scales in a hierarchical structure, making it complex and expensive to implement and manage when the whole cloud system is growing larger. Another drawback of this approach lies on the fact that although the child OpenStack cloud instances are still accessible even if the parent OpenStack cloud instance is down, the consistency will be lost between the parent instance and child instances once the parent instance is up again. More recently, the concepts behind [12] have been split into two distinct OpenStack projects [13] [14].

More recently, an OpenStack working group was created to debate and investigate how OpenStack can address Fog/Edge Computing use-cases through the supervision and use of a large number of remote data centers through a single distributed OpenStack system [15]. So far this group has focused on evaluating alternatives for communication and database solutions in OpenStack [1].

There is multiple work, not limited to OpenStack, in the literature that looks at distributed cloud deployments (e.g. [16]

[17] [18] [19] [20] [21]). However, most of these solution fall short by either following a hierarchical/centralized approach (e.g. [16] [19]) or by having single point of failures (e.g. [17] [18] [21]). Moreover, no effective evaluation of the scalability of these solutions is available.

In summary, there is a lack of solutions that easily scale horizontally. Solutions that do not have a hierarchical relation between cloud instances and where the system architecture is flat, by which the solution enables cloud instances to easily join or leave without affecting proper functioning.

III. Architecture of OpenStack

OpenStack consists of a set of interrelated services that control hardware pools of computing, storage and networking resources throughout a data center. Figure 1 depics the logical architecture of the most common integrated services within OpenStack and how these interact with each other. All services authenticate through a common identity service, and individual services interact with each other through public APIs, except where privileged administrator commands are necessary. Below is a short description of the depicted services.

- Identity service (Keystone): provides API client authentication, service discovery, and distributed multi-tenant authorization by implementing OpenStacks Identity API.
- Image service (Glance): provides a set of services for discovering, registering and retrieving VM images and VM image metadata. Images made available can be stored in different locations (e.g. Swift, Cinder).
- Compute service (Nova): provides on-demand access to compute resources, including bare metal, VM, and containers.
- Networking (Neutron): provides Networking-as-a-Service (NaaS) in virtual compute environments through SDN technologies, among others.

OpenStack scalability had long been a topic of high interest. There have been studies from the OpenStack community to find and quantify the bottle necks in the OpenStack architecture. As per the studies *nova-scheduler* becomes a performance bottle neck. *neutron* starts to present performance degradation as the number of nodes exceeds 400. When the VMs in an OpenStack instance reach above to 24000, issues are seen with RabbitMQ and Ceph which in turn result in massive failure of control plane services and agents.

IV. Solution Approach

As mentioned in section II, OpenStack services use clustering-based solutions to scale most of the control functions. This solutions have allowed OpenStack to scale up to a thousand nodes. However, this is not good enough to scale up to the size required for a large datacenter or distributed cloud scenarios where one could easily expect up to several tens of thousands of servers. Therefore in order to allow OpenStack to scale to these sizes, one needs to look to hierarchical or P2P solutions. Hierarchical solutions are already considered in the literature and their drawbacks are highlighted in section II. Therefore, this work considers P2P-based approaches to achieve the required level of scalability and robustness.

A. Designs Principles

Our solution is built upon three key design principles. The first principle is to avoid centralized components as much possible. This is primarily to allow for the required scalability and robustness properties of the proposed solution. The second principle is to minimize or if possible avoid any changes to the existing software. This allows an easy adoption of the solution and also enables the solution to be applicable to also other cloud platforms than the ones in consideration. The last principle is to *externalize* problems to an outside system whenever those problems can be efficiently and effectively solved by external systems. This simplifies the solution and allows us to reuse existing solutions as much as possible.

B. Architecture

The straightforward approach to making any software system scalable involves redesigning the software subsystems in order to make the entire system scalable. However this approach is against our second principle and therefore, our architecture is based around an unmodified OpenStack cloud platform whereby the required level of scale is achieved by running several of these unmodified OpenStack clouds which we call *cloudlets* [23]. The key challenge in such a setting is thus the question of how to make these independent cloudlets appear as one big cloud platform for the end users. We achieve this by a setup that looks like the one depicted in Figure 2. As can be seen from the figure, an agent is associated with each cloudlet whereby the request of a user is received and processed. The agent may choose to forward the request to the OpenStack instance it is associated to, or may decide to forwarded it to another agent at a remote OpenStack instance.

The agents implement three key functions. First, an agent implements a *service proxy* functionality for each supported (OpenStack) service. For instance, the Glance service proxy should expose an API interface that is identical to that of an OpenStack Glance service, allowing unmodified OpenStack glance clients to connect to the service and issue normal Glance commands. The request forwarding and translation logic for each service is also implemented in each service proxy. Second, each agent implements *overlay management* function. This function maintains several overlay networks that allow agents to discover each other and identify an agent responsible for a given request. Finally, the agent also implements a *state management* function that is responsible for keeping information about users and the cloud resources associated with them. There is also an Agent-to-Agent (A2A) interface whereby an agent communicates with another agent. A summary of the functions of an agent is shown in figure 3

The design assumes that a user is mapped to an agent and all the user's (and all other users in the user's tenant) requests are sent to this agent. In other words, an agent will be responsible for handling *all* requests that come from all users of a given tenant. The mapping can be statically created at the time the

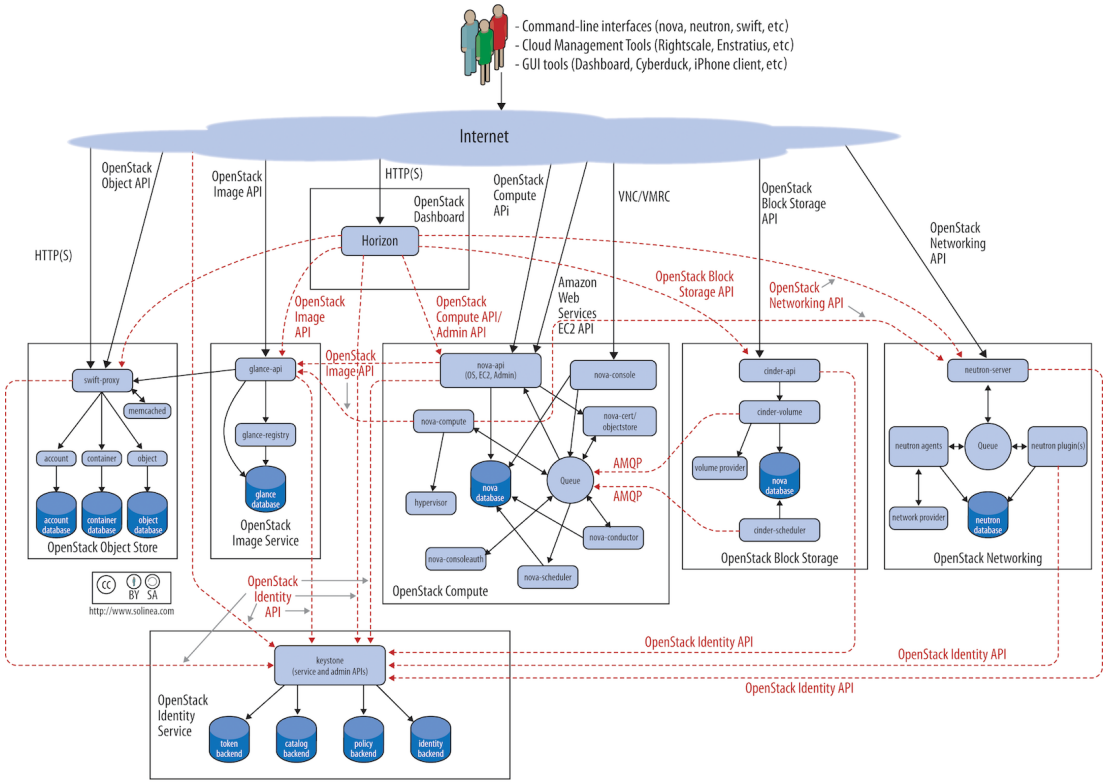

Fig. 1. OpenStack Logical Architecture [22]

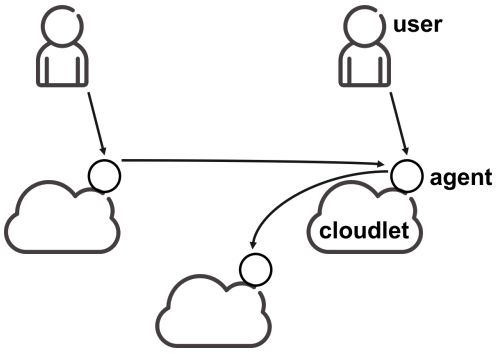

Fig. 2. Proposed architecture

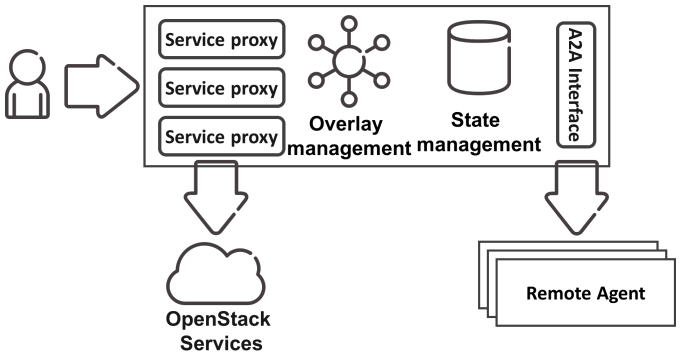

Fig. 3. Key functions implemented by an agent

user's tenant is created, or it could be determined dynamically, for instance through some form of DHT whereby the hash of a user's tenant ID determines the agent that processes the request. Our implementation follows the static approach.

In the following sections we outline how the various components as well as individual service proxies can be implemented and when relevant, we detail how our implementation realizes that specific function.

C. Overlay Management

The backbone of any P2P system is its group membership management system. Such a system typically maintains, at each node of the system, a list of other 'neighbor' nodes whereby the transitive closure of the graph representing the neighbor relationships returns all nodes in the system. On top of the group membership management system, other overlay graphs can be implemented for specific functions, such as DHTs (for e.g., user to agent mapping) or random peer sampling service (for e.g., scheduling, see section IV-G). For implementation we use the Cyclon protocol to implement group membership management [7].

D. State Management

As each tenant is mapped to an agent, each agent needs to keep track of the resources allocated to that tenant. This is particularly important since the resources of a tenant may be provisioned at several different cloudlets and the agent is the only point in the system where this information is available.

Beyond collecting resource information, the state management also manages the mapping of resource IDs. Specifically, the agent uses internal IDs to resources that are created through it, and the mapping keeps track of the IDs assigned to the resources at the cloudlet(s) where the resources are actually provisioned. The state does not contain detailed information regarding the resource, rather it contains information regarding at which cloudlet the resource is provisioned and what the ID of that resource is at that cloudlet. The details of the resource will be retrieved from the actual cloudlet. The state information may be replicated using paxos-based systems (such as etcd [24]) on several agents, improving system robustness as well as system performance for scenarios where tenants are large in size. In our implementation the state management is implemented as a simple database table.

E. Keystone

As mentioned above, Keystone in OpenStack implements three main functions: authentication, authorization and service discovery. With regards to service discovery, as a response to a service catalog request, the agent returns the addresses as well as the API versions of the implemented service proxies. This way, the solution ensures that all requests to the cloud go through the agent and not directly to cloudlet instances. With regards to authentication and authorization, the solution uses *federated identity* whereby the problem user authentication is externalized to an identity provider. In federated identity, the user authentication is performed at some external *identity provider* (IDP) which is typically different from *service providers* (SP) that are responsible for authorizing the users. In our solution, the Keystone proxy receives authentication request from users, performs the authentication at the configured IDP and returns the authentication token to the user. A user normally uses this token in all subsequent calls, which in turn is used by the agent at local and remote Keystone instances (that are configured as SPs) to obtain tokens valid for that OpenStack instance. Keystone currently supports OpenID Connect and SAML federated identity protocols. Our implementation uses Keystone-to-Keystone federation based on the SAML2 protocol.

F. Glance

There are several ways to realize the functionality of Glance in our architecture. We first describe a solution that would work on most deployments of Glance. When an image is first created, the agent will create/upload it to an OpenStack instance. This instance maybe the instance the agent is attached to (simple solution), a randomly selected instance (load balancing, see section IV-G), or selected through some other parameter (e.g., instance where the user will likely start VMs on). The agent will store the ID of the image in Glance and the address of the agent responsible for that instance in the state management system. When the agent needs to boot a VM from this image, it will first look up whether the image is uploaded to the Glance of instance where the VM is supposed to start. If not, the agent will download the image locally and then upload the image to the said instance, and then boot the VM. Each time the image is uploaded to a new OpenStack instance the agent will store this information in order to make use of it in the future. This solution has the advantage that it is portable, but has the drawback that booting VMs on OpenStack instances that do not already have the image will be slow (depending on the size of the image). A better solution is to externalize the image storage problem (e.g., use some form of object storage server such as Swift or Ceph) and instead of uploading the actual image to the Glance service of the OpenStack instance, only upload the image URL. Our implementation uploads the image to the first OpenStack instance and exposes the image through a web server. The URL of the image is then used for all subsequent uploads to other instances.

G. Nova

At its core, the Nova service is responsible for VM provisioning. The mechanics of receiving a request for a VM, translating the IDs to ones that are relevant to instance where the VM is to be started and issuing the command to start the VM is rather straight forward, as long as the appropriate ID mappings are available in the state of the agent. A key challenge here is the selection of the instance where the VM is to be started (i.e., VM scheduling). For instance, one may also choose to start VMs from large images at those instances where the image is already uploaded. One may also prefer for all communicating VMs to be started in the same OpenStack instance (to ensure maximum network performance, but this may cause the instance to be overloaded). There are possible scenarios where the above two conflict with each other.

The specific implementation of the scheduling function is dependent on the operational policy of the specific cloud operator and as such we will not attempt to cover all possible policies. However, we will choose a specific policy and describe how it is realized in our implementation. The specific policy we adopted in our implementation is that of balancing the memory load on all servers. This policy is also the default scheduling policy in OpenStack. The scheduling policy is based on the classic 'power of two choses' approach [25]. To achieve this, the agent first selects two instances at random (using the overlay management function) and identifies from the two instances the instance that has a server with the smallest amount of memory load. The agent will then send the VM request to this instance.

H. Neutron

Neutron is the networking subsystem that provides both layer-2 and layer-3 connectivity for the VMs deployed. Providing a unified view of network constructs across multiple instances of OpenStack brings forth new challenges. Extending a layer-2 network between multiple OpenStack instances requires the L2 broadcasts domain (required for ARP and DHCP) of a virtual network to be spread across the OpenStack instances, which can have implication on the performance of the network. The MAC addresses assigned to VMs on a L2

network need to be consistent across OpenStack instances to make sure that there are no MAC address collisions between VMs on the same L2 networks deployed on the different OpenStack instances. The same goes with IP address allocation, as these need synchronization cross OpenStack instances. An L3 network that needs to be distributed has to ensure that only one OpenStack instance hosts the gateways for the L3 network and design for routing the packets needs to be taken care of to ensure tnetwork traffic does not trombone.

A simple solution to distribute an L2 network across OpenStack instances is to run the neutron in VLAN mode and coordinate between the neutron instances to ensure that the same VLAN ID is used across OpenStack instances for the same network. Packets can be tunneled with their VLAN tags between the OpenStack instances, this solution is simple but is also limiting, there can only be up to 4096 VLAN across the entire distributed OpenStack deployment. Another possible solution is to develop a distributed neutron plugin that takes care of stitching the different segments of the same L2 and L3 networks on different OpenStack instances together.

Spanning a virtual network across multiple data centers or cloud instances has been and continues to be a topic of research interest. Part of the SAIL project [16] focused on interconnecting data center networks over WAN provider networks in an automated manner, proposing solutions for the federation of both L2 and L3 networks. A current effort in OpenStack, project Tricircle [13], aims to provide automatic networking spanning across multiple OpenStack deployments. In essence, although our implementation does not currently support network federation, we can leverage on these existing efforts to achieve the objective.

V. Evaluation

We evaluate the functionality and performance of the P2P approach under different scenarios whereby we vary the system size and the load on the system and collect different metrics. Performance results from an earlier version of the system are reported in [26]. In this work, we report on evaluation results from the latest version of the system where we measure the system performance while varying both the system size and the load on the system at the same time.

A. System Setup

We use an emulated environment (similar to the one used in [1]) to test our solution whereby we deploy the various OpenStack services in (resource constrained) LXC containers that emulate a physical OpenStack node. We ran all the OpenStack controller processes and the agent service in one container while the compute processes (processes needed to start and run a VM) are run in another container. We severely restrict the resources available to the containers (2 CPU cores and 4GB of RAM for controller, 2 CPU cores and 2GB of RAM for compute) so that we get an understanding of the system performance at its limit. The agents implement all the functionalities outlined in section IV, including the load balancing objective for VM placement, except for those in section IV-H. The Keystone IDP (see section IV-E) is run in its own container.

OpenStack's testing framework (OpenStack Rally) was used as a load generator to send requests to the system. The framework emulates the action of users where each user uploads a VM image and attempts to boot a configurable number of VM instances concurrently. The framework keeps track of the time it takes to boot each VM and whether or not each VM was booted successfully.

We also compare our system with the performance of an equivalent OpenStack deployed in the centralized way. Similar to the P2P setup, both the controller and compute services are run in resource constrained LXC container (4 CPU cores and 8GB of RAM for controller, 2 CPU cores and 2GB of RAM for compute). We say a P2P and standard OpenStack setups are equivalent when they have the same number of compute nodes. In the case of the standard setup, the load generator will send all user requests to the controller.

B. Performance Results

We run an experiment where we try to understand how the system performs when it scales, and how its performance compares to an equivalent standard OpenStack deployment. In this experiment we start with a deployment of OpenStack with one instance and configure the load generator with one user. The load generator uploads a VM image and attempts to start 4 VMs concurrently and measures how long it takes, on average, to boot up a VM and also what fraction of the attempts to boot a VM failed. This is repeated 15 times. We then double the system size (and the number of users) and redo the experiment. This is continued until the system has 32 instances with 32 users generating load concurrently. The entire set of experiments are repeated with the equivalent standard OpenStack setup whereby the number of compute nodes is increased with the number of users generating the load. The results are presented in Figure 4.

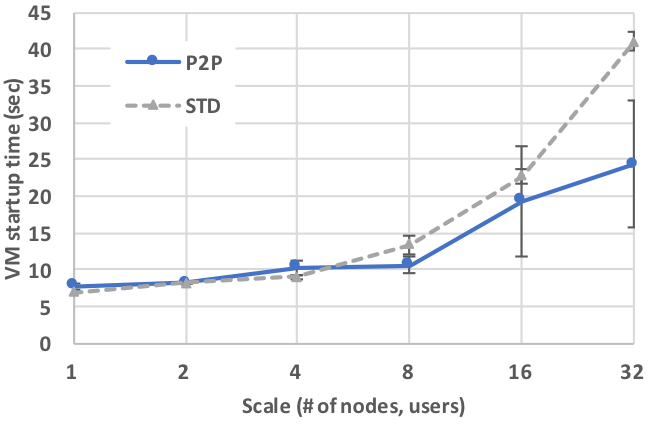

Fig. 4. VM startup time of the standard and P2P systems with increasing system size and load

The figure shows how the average VM startup time. As can be seen, both the P2P and standard OpenStack deployments

have comparable performance for 'small' systems (system size of up to 8) where VMs have similar startup times and no failures are reported. The P2P system outperforms the standard system, with the performance difference increasing with increasing system size. At system scale of 32, we see that the startup time for the standard system is 70% more than that of the P2P system.

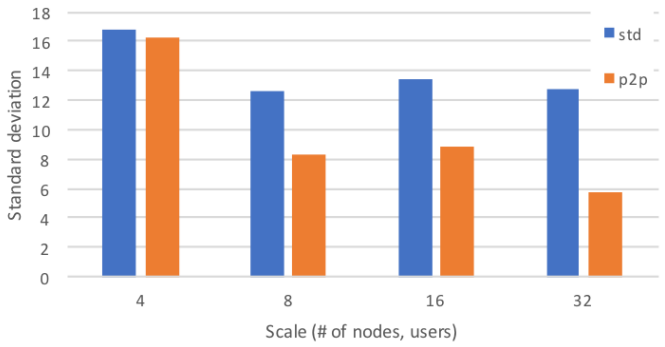

Fig. 5. Standard deviation of the number of VMs across the instances in the standard and P2P systems with increasing system size and load

We have also measured how well the objective of load balancing is achieved in both the standard and P2P systems. Our measurements, depicted in Figure 5, show that for the system size of 32, the standard deviation of the number of VMs across the instances in the P2P system is about 5.8 while that across the compute nodes of the standard system is about 12.7. The standard system performs bad here because it is unable to enforce the objective of load balancing under the high concurrency of the requests while for the P2P system the load is already balanced by design.

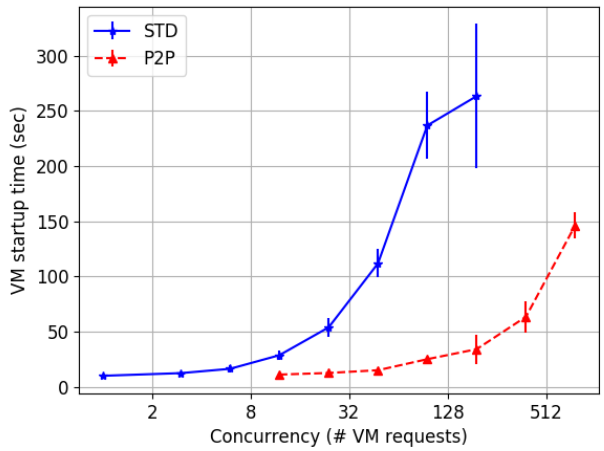

Fig. 6. VM startup time of the standard and P2P systems (size 64) with increasing concurrent VM requests

Moreover, we performed tests for a system size of 64 nodes while increasing the number of concurrent VM requests (1, 3, 6, 12, 24, 48, 96, 192, 384 and 768). With this we intended to further understand the system performance while incresing the demand on the system.

Figure 6 shows the VM startup time, where it is possible to see an increasy gap between the times registered for the P2P and standard systems as the VM concurrency increases. Note that for the P2P system, the load generator simultaneously stresses 12 cloudlet agents out of the 64. For that reason, the values for the P2P system were only register starting from a concurrency of 12. Comparing the two system, for example, for a concurrency of 12, the VM startup time standard system is 159% more than that of the P2P system, while for a concurrency of 192 is 674% more. Concurrent requests above 192 for the standard system are not depicted as the system does not manage to complete any request for such values (i.e. VM failure rate is 100%).

Figure 7 shows the failure rate of both systems in such conditions. The standard system starts to present failures for a concurrency of 192, with an average failure rate of 68%. For higher concurrency values the failure rate is of 100%. On the other hand, the P2P system only starts to show failures for concurrency 768, with an average failure rate of 22%.

In summary, results show that the P2P system can boot 4 times more VMs (i.e. 768 compared to 192 for the standard) with a much faster average startup time (approximately 80% faster than the stardard with concurrency 192) while maintaining an error rate that is less than one third of the standard (22% for concurrency rate 768, compared to 68% for concurrency 192 in the standard system).

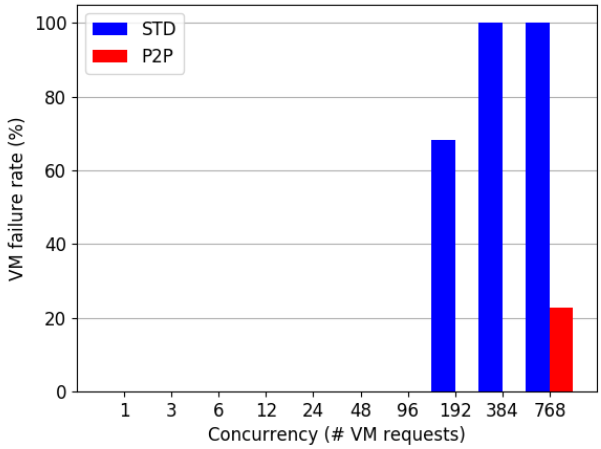

Fig. 7. VM failure rate of the standard and P2P systems (size 64) with increasing concurrent VM requests

VI. Conclusion and Future Work

A. Discussion and Conclusion

In this paper, we consider the problem of scaling cloud platforms. We specifically look at those cloud platforms in the public domain, where most of these platforms follow the

centralized management paradigm where control functions are centralized to a few 'controller' nodes in the system.

We argue that this centralized approach has inherent scalability and robustness issues and therefore propose a novel architecture for massively scaling cloud platforms based on a P2P design. Our architecture is based on three design principles that revolve around avoiding centralized components, and minimizing the work needed to implement, adopt and use the solution. We describe the architecture and detail how its core functions can be implemented. We exemplify how specific functions of a cloud platform can be realized by showing how the basic services of an OpenStack cloud (i.e., those included in the Compute Starter Kit [27]) can be implemented in this new architecture. The results from a comparative performance evaluation of our solution with that of a standard centralized system show that while the performance of our solution is comparable to that of a standard system for small system sizes, our solution outperforms the standard system at larger sizes with regards to how fast VMs are booted, how few VMs fail to boot and how well the objective of balancing is achieved.

B. Future Work

The work reported in this paper points out to a new approach for massively scaling cloud platforms using a P2P architecture. Though we have sufficiently demonstrated its feasibility by implementing key services of OpenStack, what we present here is a solution that is far from complete and there remain a number of interesting research challenge. One important challenge is that of scheduling of resources (e.g., VMs). For instance, beyond the challenges outlined in section IV-G, it is not also clear how our specific approach can be extended to settings where the cloudlets are not identical and can provide different types of services (e.g., have different hardware capabilities). Adopting other management objectives than load balancing is also an open problem that needs further investigation. There are also several problems related to implementing other OpenStack services such as block storage (e.g., how to access volume images on remote instances) or telemetry service (e.g., how to aggregate data at user level and datacenter level). Finally, though this approach can easily be used in both centralized as well as distributed/edge cloud deployments, considerable work is needed in order for this approach to fully exploit the benefits (that specifically relate to location and network resources available between instances) afforded by the edge/distributed deployment model for cloud services.

REFERENCES

[1] Chasing 1000 nodes scale, https://www.openstack.org/assets/presentation-media/Chasing-1000-nodes-scale.pdf, accessed: 2017-07-19.

[2] Scalability updates in Kubernetes 1.6: 5,000 node and 150,000 pod clusters, http://blog.kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html, accessed: 2017-07-19.

[3] H. Chang, A. Hari, S. Mukherjee, and T. V. Lakshman, "Bringing the cloud to the edge," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2014, pp. 346–351.

[4] ARTEMIS a parallel and distributed cloud simulation suite, https://www.ericsson.com/research-blog/artemis-parallel-distributed-cloud-simulation-suite/, accessed: 2017-07-19.

[5] M. Dam and R. Stadler, "A generic protocol for network state aggregation," in *In Proc. Radiovetenskap och Kommunikation (RVK*, 2005, pp. 14–16.

[6] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in p2p systems," *Commun. ACM*, vol. 46, no. 2, pp. 43–48, Feb. 2003. [Online]. Available: http://doi.acm.org/10.1145/606272.606299

[7] S. Voulgaris, D. Gavidia, and M. van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, Jun 2005. [Online]. Available: https://doi.org/10.1007/s10922-005-4441-x

[8] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, pp. 219–252, Aug. 2005. [Online]. Available: http://doi.acm.org/10.1145/1082469.1082470

[9] OpenNebula, https://opennebula.org/, accessed: 2017-10-27.

[10] Kubernetes, https://kubernetes.io/, accessed: 2017-10-27.

[11] Openstack Cells, https://docs.openstack.org/ocata/config-reference/compute/cells.html, accessed: 2017-07-19.

[12] Openstack Cascading Project, https://wiki.openstack.org/wiki/OpenStack_cascading_solution, accessed: 2017-07-19.

[13] Openstack Tricircle Project, https://wiki.openstack.org/wiki/Tricircle, accessed: 2017-07-19.

[14] Openstack Trio2o Project, https://wiki.openstack.org/wiki/Trio2o, accessed: 2017-07-19.

[15] Openstack Fog Edge Massively Distributed Clouds Working Group, https://wiki.openstack.org/wiki/Fog_Edge_Massively_Distributed_Clouds, accessed: 2017-07-19.

[16] H. Puthalath, J. Soares, A. Sefidcon, B. Melander, J. Carapinha, and M. Melo, "Negotiating on-demand connectivity between clouds and wide area networks," in *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, Nov 2012, pp. 124–130.

[17] F. Brasileiro, G. Silva, F. Arajo, M. Nbrega, I. Silva, and G. Rocha, "Fogbow: A middleware for the federation of iaas clouds," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 531–534.

[18] R. Buyya, R. Ranjan, and R. N. Calheiros, *InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 13–31. [Online]. Available: https://doi.org/10.1007/978-3-642-13119-6_2

[19] D. Villegas, N. Bobroff, I. Rodero, J. Delgado, Y. Liu, A. Devarakonda, L. Fong, S. Masoud Sadjadi, and M. Parashar, "Cloud federation in a layered service model," *J. Comput. Syst. Sci.*, vol. 78, no. 5, pp. 1330–1344, Sep. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jcss.2011.12.017

[20] A. Celesti, F. Tusa, M. Villari, and A. Puliafito, "How to enhance cloud architectures to enable cross-federation," in *2010 IEEE 3rd International Conference on Cloud Computing*, July 2010, pp. 337–345.

[21] E. Carlini, M. Coppola, P. Dazzi, L. Ricci, and G. Righetti, *Cloud Federations in Contrail*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 159–168. [Online]. Available: https://doi.org/10.1007/978-3-642-29737-3_19

[22] OpenStack Open Source Cloud Computing Software, https://www.openstack.org/, accessed: 2017-07-19.

[23] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct 2009.

[24] etcd - A distributed, reliable key-value store for the most critical data of a distributed system, https://coreos.com/etcd, accessed: 2017-07-19.

[25] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, Oct 2001.

[26] X. Han, "Scaling OpenStack Clouds Using Peer-to-peer Technologies," Master's thesis, Chalmers University of Technology, SE-412 96 Gothenburg, Sweden, 2017.

[27] Openstack Compute Starter Kit, https://governance.openstack.org/tc/reference/tags/starter-kit_compute.html, accessed: 2017-07-19.