Microservices Architecture for Tenant-oriented Cloud-based Control Software Services: POC and Proposals

Roman SOROKIN\textsuperscript{1,2,*} and Vinay SINGLA\textsuperscript{2}

\textsuperscript{1}Bauman Moscow State Technical University (RUSSIAN FEDERATION)
\textsuperscript{2}Cisco Systems Inc. (UNITED STATES OF AMERICA)

*Corresponding author

Keywords: Microservices architecture, Linux-container, Docker, Control software, Cloud software, Management system, Orchestration, Tenant-oriented architecture.

Abstract. Microservices architecture, where application is composed of a set of independently deployable services is a popular modern approach for building enterprise-level applications, meanwhile most of the cloud solutions for control software (including IoT) are designed in so called monolithic way. LXC-based (docker) microservices allow developers to seamlessly migrate current software solutions from VMs to containers. Nowadays, such container-based applications are used primarily for simple semi-automated deployments, when containers are deployed only at the launch time of the whole system and modified when needed. In this paper we analyze application of docker-microservices as functional blocks for control software, propose on-demand tenant-oriented microservices approach (where entire independent topologies of microservices are being deployed on per-tenant basis) for building clouds of control software, review requirements for management system of such cloud and investigate contemporary container-management systems. Constructed test cluster highlighted some disadvantages of Kubernetes regarding multi-tenant container administration. Advanced Container Management System (AMS) is designed as one more level of orchestration on top of Kubernetes. Conducted demo showed practical value and flexibility of the introduced tenant-oriented microservices method. Described design will be helpful for cloud providers of control software who want to organize elastic on-demand service provisioning.

Introduction

Fundamentally, microservices architecture is not something brand-new, but today this approach of software organization and distribution is widely used. Each application in microservices architecture is built as a set of relatively small services, each of them is executed in its own separate process. Inter-microservice communications are usually organized by means of lightweight standard mechanisms like HTTP [3]. On the Fig. 1 we depict differences between Monolithic and Microservice Architectures.

![Figure 1. Monolithic vs. Microservices Architectures.](image-url)
In monolithic architecture:
- entire set of functions is placed into a logical component (for example, VM);
- operational logic is hard-coded, lack of flexibility;
- scaling is performed by means of replication of the core (logical component) across nodes.

In microservices architecture:
- each functional element is placed into its own separate component (microservice);
- operational logic is flexible, microservices can be interchangeable;
- scaling is performed by distribution of particular microservice across different nodes, with replication if needed.

Mainly, microservices are used for enterprise-level applications, where each microservice represents particular piece of the whole system. For example, one microservice is responsible for portal, another one for db, the third one can play the role of email notification outbound component. Replication of microservices is performed upon needs of the load.

The interest of LXC-based microservices approach has dramatically increased over the past 2 years, especially huge impact is being brought by Docker (LXC-wrapper) [11]. Docker containers have advantages, including, but not limited to, small computation overhead, sharing the same kernel of the host VM, flexible management due to the CLI [12, 14]. The Docker efficiency is comparable to the KVM efficiency, and sometimes Docker is even better than KVM in this regard. All these features allow LXC to become an efficient mean of microservices representation [13]. Also, LXC extremely simplifies microservices development [9] and existing instruments of automation like Ansible and SaltStack make docker even more powerful [12].

Let’s boil down existing possible application designs with docker containers to three key approaches [15] (Fig. 2).

![Figure 2. Docker container scenarios.](image)

Hereby, usage scenarios are:
- Each container is a set of fixed software components. Such container is used as a standalone software (similarly to existing VM-based approach).
- Each container has a limited number of software installed. Thus it is possible to split software behaviour into logically separated containers.
- Container plays the role of datastore.

It is obvious that last two scenarios are the most interesting from microservices architecture perspective.

**Requirements for Control Software Systems with examples for IoT**

There is a whole set of use-cases in cloud architecture where cloud control software should be constructed as a set of resources, independently deployable for each tenant. Such resources could vary by functionality and underlying composition. A good example would be a cloud network virtualization where software by itself is a set of independently developed VM-oriented virtual appliances, deployed on the per-user basis. Per-tenant cloud organization and orchestration gives deployment flexibility and significantly simplifies development process of new appliances.

Nowadays, basically all control software is built using monolithic architecture. In IoT field the most functionally developed and significant IoT platform is IBM’s BlueMix proprietary software [18]. Even though it has internal docker containers and VMs as additional building blocks for custom architecture, nevertheless its architecture is monolithic and there are some disadvantages regarding automation of provisioning new services.
Here under control software we assume any software that manages some real devices. The good analogy to control software would be IoT, where cloud software is a core part of the whole management system. We’re not talking about sensor-level management of IoT. In fact, we study use-cases where some kind of the high level device preprocesses data and communicates with the cloud using high-level data representation (for instance, HTTP).

So, our idea is to introduce microservices architecture on LXC containers as a design for control software deployable on the per-tenant basis. In fact, last two container scenarios (see Introduction) could play the role of building blocks of the cloud. Logical and physical separation as well as provisioning automation of independent executable sets of blocks (container topologies) would make it possible to customize tenant's deployments and give an opportunity to dynamically change the cloud topology on per-user basis. Providers of cloud control software could use such approach to manage their cloud solutions and charge tenants only for those resources that were actually configured and provisioned.

Let’s start from giving some requirements for such microservice-oriented system of control software:

- Heterogeneity (of data, connections, topologies, functions) [5];
- Horizontal scalability (instead of vertical) [6];
- Simplicity of introduction of new functions / easy integration of new functions into existing architecture [7];
- Independence and ability to reuse separate cloud resources [8];
- Simple yet powerful programing administration – REST API [9];
- Platform-agnostic and hardware-agnostic cloud [10];
- Dynamic resource addition/deletion;
- Ability to deploy complex and diverse cloud control services;
- Independent namespaces for topologies;
- Proactive and root-cause capabilities on the per-deployment basis;
- Effortless migration of microservices to the new container provider;

Hence, let’s analyze existing container management systems which could satisfy our requirements.

**Cloud Systems for Container Management**

In the cloud hierarchy, the CAAS (Container-as-a-Service) level is determined as a subset of *IaaS* (*Infrastructure-as-a-Service*). In this case under compute resource a container is meant [16]. CAAS is implemented by special container management systems. The main component of such system is Cloud Management Software (Fig. 3), which is responsible for several functions:

- Container Scheduling and Provisioning;
- Multi-container Management;
- Service Discovery and etc.

![Figure 3. General CAAS Cloud Structure.](image)

All existisitng Container Management Systems can be classified into two major groups:

- Systems oriented for *Multi-server Deployment, High Availability and Service Discovery* (example: Google Kubernetes);
Systems oriented for local usage (example: docker-compose).

We do not take into consideration systems for local usage due to their limitations and inconsistency with our research goals. Among Multi-server container management systems, we would like to highlight Google Kubernetes and Amazon ECS. ECS is a proprietary software and it is available only as part of Amazon paid services (with free trial) [17]. Comparing Kubernetes and ECS, we have discovered that they are almost similar by functionality, hence further research will be held against Kubernetes (open source). Brief characterization of the most beneficial Kubernetes features are:

- container life-cycle management;
- container health checking (with auto restart);
- concept of Kubernetes Services (for container access consistency);
- simple db with information about created containers;
- CLI and REST API for remote access and administration [19].

Well, the Kubernetes is able to spin up simple container topologies and maintain their lifecycle. Thereby, we want investigate if Kubernetes can satisfy our requirements for management system of control software.

**Deploying Test Kubernetes-based Container Cluster**

Test cluster consist of 6 VMs (according to recommended replication ratio) in OpenStack on Cisco UCS hardware with two Open Stack Neutron subnets: one internal and one external. Creation of VM images for OpenStack (qcow2) we performed using Packer tool [20], thus all VMs have the same basic software installed. In order to eliminate configuration errors, we automated process of deploying Kubernetes with set of Ansible scripts, including, but not limited to, Neutron configuration, VM launching, Docker installation, configuration of Private Docker Registry on one of VMs, installation and configuration of Apache Mesos, installation of Master and Slave Kubernetes instances, configuration of inter-node overlay communications with flannel. All VMs are identical from resources standpoint. Some basic parameters of the test cluster are summarized in Table 1.

Table 1. Test cluster parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM, Gb</td>
<td>16</td>
</tr>
<tr>
<td>Amount of virtual kvm core, pcs.</td>
<td>8</td>
</tr>
<tr>
<td>Main partition, Gb</td>
<td>50</td>
</tr>
<tr>
<td>Amount of ports (Master-server)</td>
<td>1</td>
</tr>
<tr>
<td>Amount of ports (Slave-server)</td>
<td>5</td>
</tr>
<tr>
<td>Amount of Master-servers</td>
<td>3</td>
</tr>
<tr>
<td>Amount of Slave-servers</td>
<td>3</td>
</tr>
<tr>
<td>Amount of virtual ip-addresses, assigned by Flannel for each Slave-server</td>
<td>255</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu Server 14.04 (64 bit)</td>
</tr>
</tbody>
</table>

In such a way, we got the cluster, schematically depicted in Fig. 4.
Let's briefly emphasize main groups of components and their role in the whole built system:

- components, responsible for Kubernetes functioning: Kubernetes Master (including API server, ControllerManager, Kube Scheduler, Kube Proxy), etc.;
- components, accomplishing internetwork communication roles of Slave-servers and ip-address assignments: Flannel and Kiwi;
- components that provide system with container execution and storing mechanism: Docker Private Registry, lxc-docker.

Conducted tests with this cluster discovered some disadvantages of the current system:

a) absence of built-in geographical distribution for containers;
b) absence of pre-built system for log aggregation;
c) bad service discovery with shared namespaces;
d) no mechanisms for automated per-container parameter injection;
e) absence of configurational db for storing predefined configurations;
f) no per-deployment configurational db with init data;
g) Kubernetes by itself is a container provider of the low level, so we need one more level of abstraction.

Prototype of Advanced Container Management System

Well, based on the requirements for the microservices management system for control software and according to demonstrated disadvantages of the container management system on pure Kubernetes, we decided to make a prototype of the orchestration system that could partially cover some Kubernetes disadvantages.

Advanced Container Management System (AMS) is a high-level software that is written in golang and in fact it translates virtual topology definition from YAML file to the sequence of get/post REST requests to the underlying container-provider (in this case it is Kubernetes). The place of the AMS is shown in the Fig. 5. It has simple CLI, which is able to parse YAML configuration files and orchestrate containers over the Kubernetes.
AMS Operation Algorithm

When we call AMS, we pass filename with configuration as a parameter (1). This is a YAML file with structure, described in Fig. 6.

```
author: Roman Sorokin
name: AMS PoC Deployment
kubeApi: v1beta1
date: Dec 9 2015
version: 0.0.1
email: rosoroki@gmail.com
description: Simple example of multiservice deployment
topologies:
  - name: WebSocket Demo
description: WebSocket PoC in AMS
  sleep: 60
topology:
    api: replicationControllers
type: post
    payload: >
    
```

After that, AMS is parsing the file and performs set of get/post requests to the Kubernetes (2). Then, all required operations (3,4,5,6,7) are performed by Kubernetes and other components from cluster. AMS config has several topology layers, so it is capable to spin up several complex independent container topologies at a time.

Each container is prepared and uploaded to the Private Docker Registry before the start of the AMS. We use enhanced containers with structure, depicted on Fig. 7. Here we inject logstash-agent for log collection and consul for service discovery.
As a result of the AMS execution we have the topology of containers in the underlying container-provider (here it is Kubernetes) with additional ELK-container for log aggregation and visualization and Hashicorp's Consul container for per-topology service discovery and storing key-value init data for topology components.

**Evaluation of the Prototype**

In order to test designed AMS system, we developed simple use-case with WebSockets handling (Fig. 8.). Here we used external VM with Web Server and Queue Server for tasks coordination and Kubernetes Cluster with AMS described above.

Here we have a set of functional containers (Kafka, ZK, Redis) and two system containers, described earlier (Consul and ELK). Demo is launched by AMS system so the full process of starting all containers is automated. Once one topology is ready, a set of requests (1, 2, 3, 4, 5, 6, 7) can be performed for particular tenant. Inter-container communications (a, b, c, d, e, f, g, h) are performed inside of the Kubernetes. Furthermore, we can spin up more demos like this simultaneously, so each
demo will have its own independent set of containers and its own set of configurations from Consul. Thus, each tenant receives its own distributed independent set of resources, with autonomous initial configuration, lifecycle and external endpoints. Independent namespace allowed us to design components as they would be executed on the regular VM. With log aggregation on the per-tenant basis we can analyze what is happening with deployment of particular user. However, one would have even more beneficial and impressive impact if such system was used for control software automation, where each tenant should be provided with set of flexible control software.

Nonetheless, we would like to present obvious disadvantages of the AMS:

- no geographical distribution (no regional redundancy);
- dependency on the Kubernetes (provider-dependency);
- absence of prebuilt scenarious for root-cause and proactive analysis as well as lack of universal IO for container hookup;
- no configurational DB and UI for storing and modifying the possible container topologies.

**Conclusion, Lessons Learned and Future Work**

This paper has discovered potential ability of using docker containers as building blocks for control software. On-demand technique regarding microservices usage was proposed. Reviewing prerequisite for modern tenant-oriented cloud of control software we have highlighted main requirements for advanced microservices orchestration system. Constructed test cluster showed some disadvantages of pure Kubernetes, based on which we have introduced new PoC Advanced Container Management System. Conducted Demo emphasized reasonable advantages and practical significance of proposed approach. The primary consumers of this system would be cloud providers of control software, willing to:

- move existing VM-based infrastructures to container-based solutions;
- introduce per-tenant deployment architecture of software.

As a future work in this direction we would like to:

- propose concrete design of the Advanced Container Management System;
- develop and test it;
- get real feedback from customers and iterate according to agile philosophy.

**References**


