BalticLSC Platform Architectural Vision

Initial considerations regarding architecture of the BalticLSC Platform

Version 1.00

Priority 1: Innovation

Warsaw University of Technology, Poland
RISE Research Institutes of Sweden AB, Sweden
Institute of Mathematics and Computer Science, University of Latvia, Latvia
EurA AG, Germany
Municipality of Vejle, Denmark
Lithuanian Innovation Center, Lithuania
Machine Technology Center Turku Ltd., Finland
Tartu Science Park Foundation, Estonia

BalticLSC Platform Architectural Vision

Initial considerations regarding architecture of the BalticLSC Platform

Work package	WP4
Task id	A4.1
Document number	O4.1
Document type	Vision document
Title	BalticLSC Platform Architectural Vision
Subtitle	Initial considerations regarding architecture of the BalticLSC Platform
Author(s)	Daniel Olsson (RISE), Magnus Nilsson-Mäki (RISE), Henrik Niska (RISE)
Reviewer(s)	Agris Šostaks (IMCS)
Accepting	Michał Śmiałek
Version	1.00
Status	**Final version**
Distribution	Public

Interreg Baltic Sea Region
EUROPEAN UNION
EUROPEAN REGIONAL DEVELOPMENT FUND

History of changes

Date	Ver.	Author(s)	Review	Change description
19.06.2019	0.01	Daniel Olsson (RISE)		Document creation and initial contents
20.06.2019	0.02	Daniel Olsson (RISE)		Added additional content from Henrik Niska
20.06.2019	0.03		A-C Eriksson (RISE)	Removed n/a chapters, added footnote ref.
21.06.2019	0.04	Henri Hanson (TSP)		Input to Executive Summary and Introduction
27.06.2019	0.05	Daniel Olsson (RISE)	Agris Šostaks (IMCS)	Made changes according to review comments from Agris. Fixed typos. Added comments.
27.06.2019	1.00	Daniel Olsson (RISE)		Document approved during consortium meeting in Luleå

Executive summary

The overall aim for the Baltic LSC activities is developing and providing a platform for truly affordable and easy to access LSC Environment for end-users to significantly increase capacities to create new innovative data-intense and computation-intense products and services by a vast array of smaller actors in the region.

Current Platform Architectural Vision Document is an important output of the Baltic LSC project, based on technical workshops held during Q1-Q2 2019 with the participation of external experts and led mainly by the project's technology partners from:

- Warsaw University of Technology (WUT)
- RISE Research Institutes of Sweden AB (RISE)
- IMCS University of Latvia (IMCS)

The Software Architectural Vision Document includes also input obtained during Baltic LSC project's international and local workshops as well as individual meetings with experts, potential cooperation partners and end-users from March – June 2019.

It is supporting the Environment Vision Document (Baltic LSC Output 3.1), which is providing an overall vision of the Baltic LSC Environment and is complemented by Baltic LSC Software Architectural Vision (Baltic LSC Output 5.1) describing in detail the hardware visions for the platform.

This document contains an initial architectural model of the BalticLSC Platform. It contains decisions on which technologies (computing and networking hardware, operating system software, computing languages) should be used to develop the final platform. Also, it will propose optimal solutions for combining selected hardware units into coherent computation grids and networking solutions that would allow combining such small grids into large computation networks on a transnational level.

Table of Contents

1. Introduction

1.1 Objectives and scope

The BalticLSC Platform is where computation tasks compiled by the BalticLSC Software are to be executed. The scope of this document is to specify the most important requirements, as well as providing a high-level architecture that supports these requirements.

1.2 Relations to Other Documents

The current BalticLSC Platform Architectural Vision Document is supporting the Environment Vision (BalticLSC Output 3.1), which is providing an overall vision of the BalticLSC Environment and is complemented by BalticLSC Software Architectural Vision (BalticLSC Output 5.1) describing in detail the main concepts, components and software development technologies of the system.

1.3 Intended Audience and Usage Guidelines

This Architectural Vision Document is the Output 4.1 as described in the BalticLSC Application Form and intended for internal use within BalticLSC consortium as the basis for future software development activities as well as for reporting purposes for local First Level Control and Baltic Sea Region Program.

2. Platform requirements

2.1 REST API

There must exist a way to programmatically use the compute resources of the Platform. A REST API must therefore be provided which will be used by the users of the Platform. The most obvious user is the BalticLSC Software.

2.2 Docker container support

During the workshops we decided that Docker[1] container should be used as the execution platform.

A container is a standard unit of software that packages up the code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, run-time, system tools, system libraries, and settings.

Container images become containers at run-time and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker is the de facto standard when it comes to containers and Kubernetes comes with out of the box support for it. These are the reasons why BalticLSC should run Docker containers.

2.3 Resource usage tracking

To enable business models, hardware resource usage tracking is mandatory. This means that the amount of resources (CPU, GPU, RAM, etc.) that a docker container consumes during its lifetime is tracked and stored.

2.4 User isolation and quota

Users of the platform must be isolated from each other. This means that one user shall not be able to affect another user in any way. The Platform must also support resource quotas and limits. This means that it must be possible to specify how much resources (see above) a user is allowed to allocate, as well as controlling that the user does not use more than requested.

[1] https://docker.com

3. Architecture vision

This BalticLSC Platform consists of several different components which are described below.

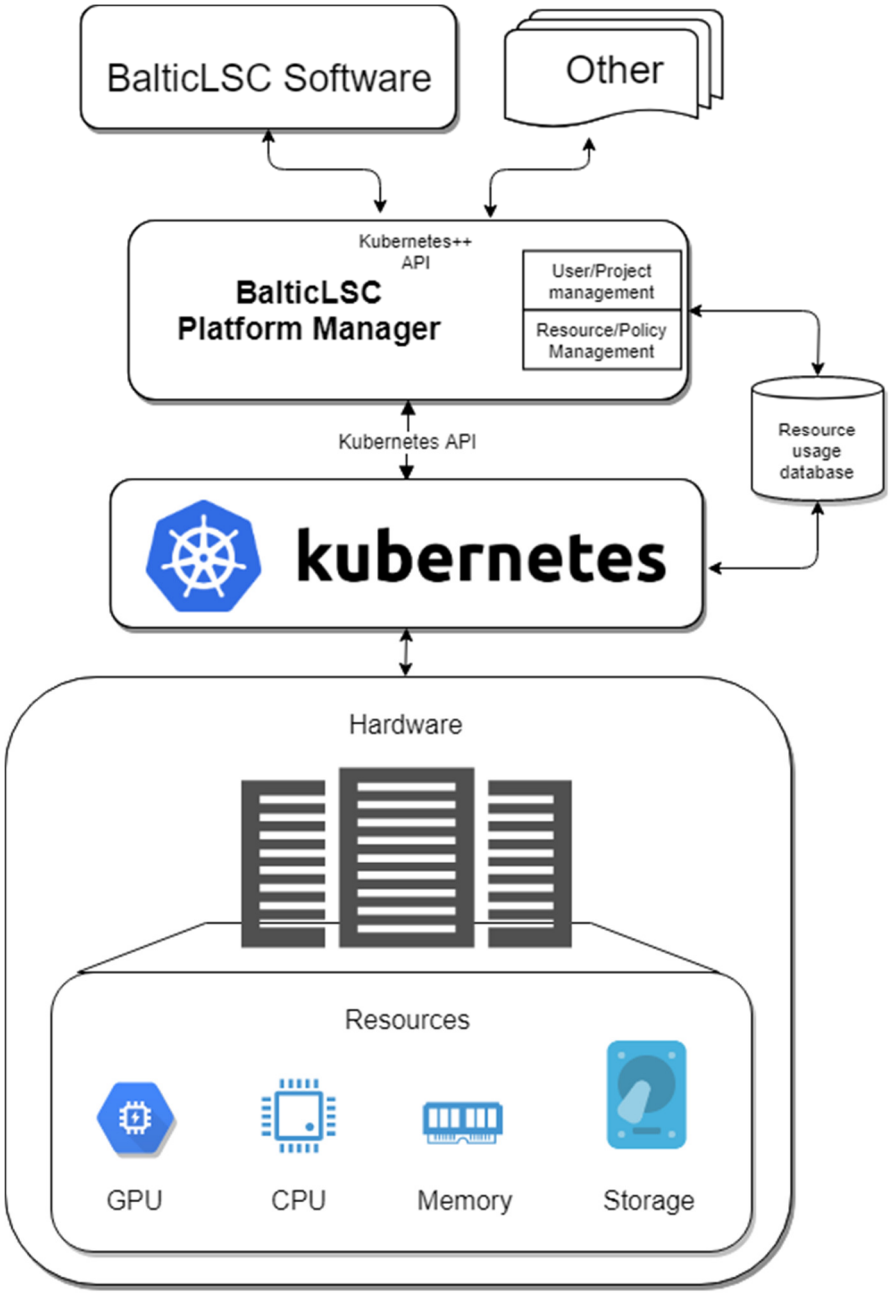

Figure 1: BalticLSC Platform Architecture

3.1 Kubernetes

Kubernetes (K8s)[2] is an open-source system for automating deployment, scaling, and management of containerized applications. Several different container orchestration platforms were evaluated before Kubernetes was chosen. Following platforms was evaluated:

- Kubernetes
- Docker Swarm
- Apache Mesos
- OpenShift (RedHat)
- DC/OS
- Amazon ECS
- Nomad
- Cattle

This evaluation was done a year ago and following google trends graph shows that Kubernetes popularity is only growing.

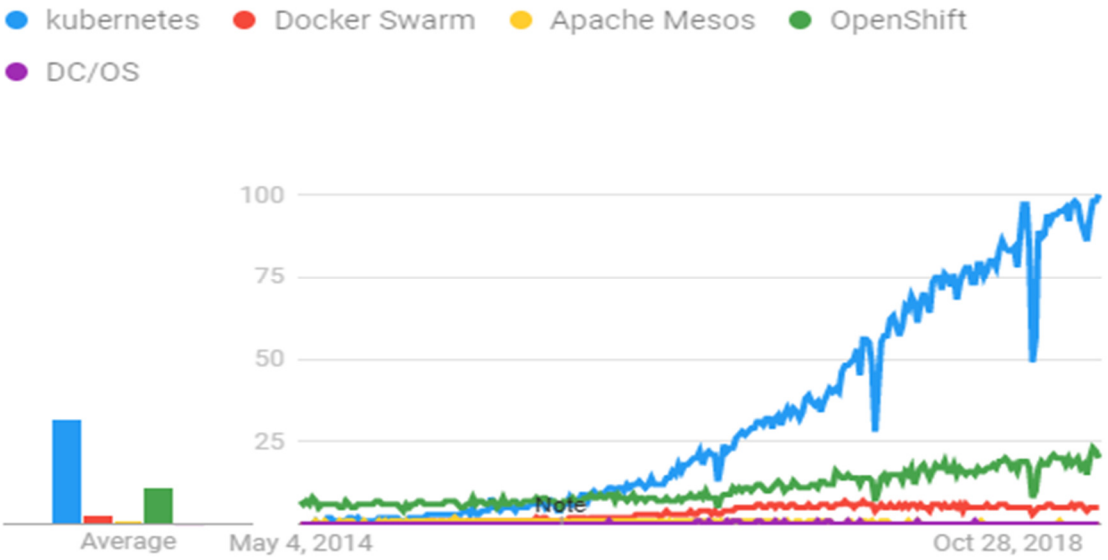

Figure 2: Google trends

A Kubernetes cluster is composed of one or more Nodes. The Nodes are the worker machines, they can be virtual machines or physical servers. Each node run the services necessary to run the workloads. Workloads consist of Pods which are groups of one or more containers with shared storage and network and a specification for how to run the containers.

Kubernetes fulfills following requirements:

- 2.1 REST API – the Kubernetes REST API
- 2.2 Docker Container Support – It is the native execution environment
- 2.4 User isolation and quota – supported by Kubernetes

[2] https://kubernetes.io

3.2 Hardware and resources

The hardware consists of computation servers, hereafter called nodes, interconnected using network switches.

There are multiple types of hardware resources that can be configured differently on different Nodes. All of the resources can be priced differently and also the same type of resource can be priced differently depending on the model and version. A factor that also can make the price vary is the geographical location of the resource because the cost of electricity can vary. Electricity cost variations and network switch resource usage costs are not taken into consideration by the platform.

3.2.1 Central Processing Unit - CPU

One thing to consider in the cluster regarding CPUs is the type of CPU running on the nodes. This information is needed both in terms of how much computation power is available and for the pricing of the resource. One CPU core is divided into millicores, 1/1000 of a core, in Kubernetes. This means that computation tasks don't need to claim full cores.

3.2.2 Memory

Much of what is being said about the CPU can be said about memory. Kubernetes provides information about the total amount of memory on one node. Detailed information about memory could be interesting for the business model.

3.2.3 Storage

Storage should be located as near as possible to the computation resources so that large amounts of data doesn't need to be moved long distances between computation cycles. The storage can be located on the Nodes where the computation should be done or on a storage cluster nearby. In the BalticLSC Platform storage clusters will be mandatory because the computation is never guaranteed to be run on a specific Node if more than one Node fulfill the requirements for the resources. There can be different classes of storage, priced differently, that have different performance. To prevent data loss the storage must replicate the data.

3.2.4 Network

Network traffic can be divided into two categories, inbound and outbound network traffic. But the inbound and outbound traffic can occur on different layers in the network, it can be inside the cluster but also from/to the cluster from/to Internet. There isn't any built-in functionality to distinguish between traffic going outside the cluster and internally. The majority of the traffic is most likely only traveling locally because the tasks are computation oriented on data already in the cluster. This, of course, assumes data has already been moved to the cluster.

3.2.5 Graphical Processing Unit - GPU

Servers usually don't contain a GPU by default except for dedicated GPU server in which case it can contain one or more GPUs. If the software that's being deployed needs GPUs it needs to be run on Nodes with one or more installed. There are some restrictions with regards to GPUs and software libraries, software development kits (SDKs) and the manufacturer of the GPU. The problem with libraries and SDKs is that they are designed for hardware from different vendors. There are tools that try to unify the hardware under common abstraction layers. Today these abstraction layers are not well supported because of the difficulty to port the different platforms.

The most established vendor in the field of using GPUs as computation power is NVIDIA, that is why our current recommendation is to only support NVIDIA GPUs.

3.3 BalticLSC Platform Manager

The BalticLSC Platform Manager fulfills requirement *2.3 Resource Usage Tracking* and makes fulfilling requirement *2.4 User isolation and quota* easier using a GUI.

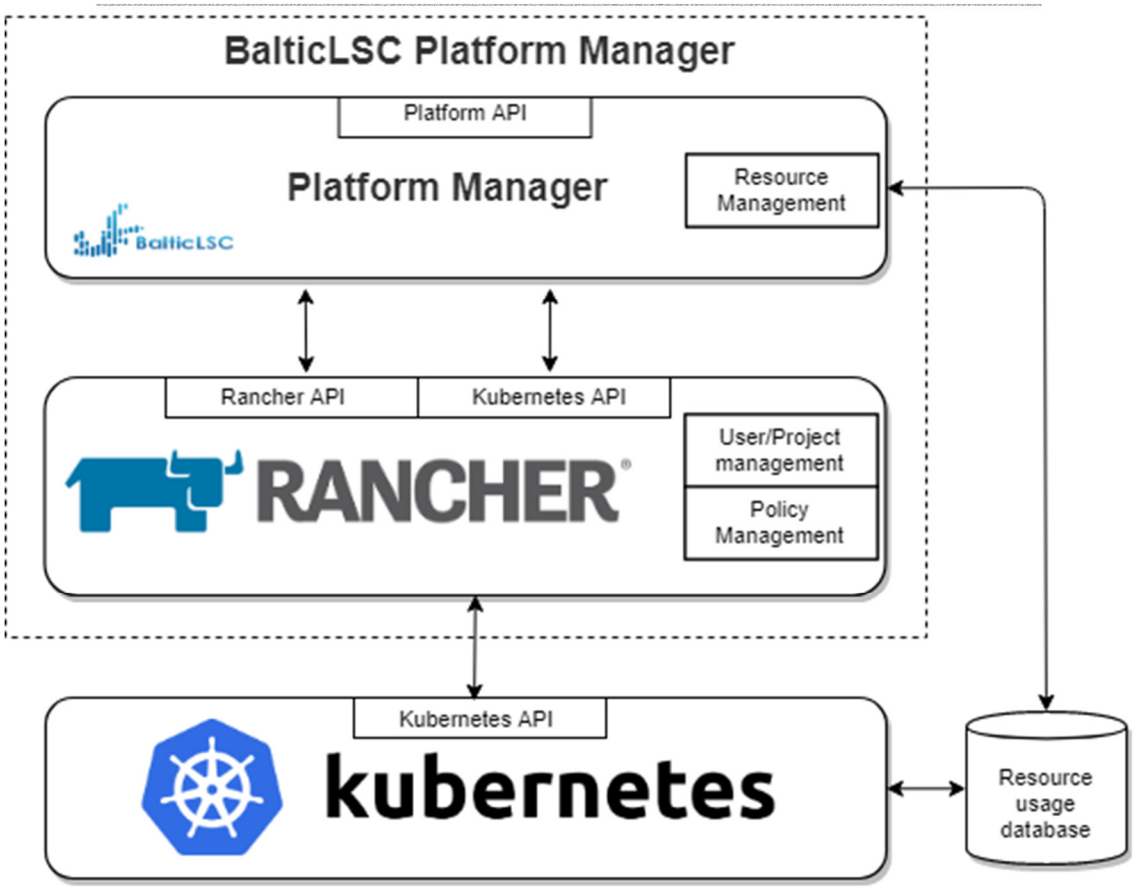

Figure 3: BalticLSC Platform Manager

3.3.1 Platform Manager

The BalticLSC Platform Manager consists of two components.

1. Platform Manager
2. Rancher (see section 3.3.2)

The Platform Manager is acting as a proxy between Rancher and users of the BalticLSC Platform. Its main purpose is to provide API access to the resource usage database. This is done via the Platform API which is the native Kubernetes API with additions for querying resource usage data. Other functions not directly supported by Rancher or Kubernetes will be the responsibility of the Platform Manager.

The Platform Manager is a new component that will be developed. It is still to be decided in which language it will be implemented. Current proof-of-concept is implemented in NodeJS. It could also be written as an addition to Rancher in the language Golang.

3.3.2 Rancher

Figure 4: Rancher

Rancher is a platform for Kubernetes management. Here is a short description from their website[3]:

Rancher is a complete software stack for teams adopting containers. It addresses the operational and security challenges of managing multiple Kubernetes clusters while providing DevOps teams with integrated tools for running containerized workloads.

Following list of features is what made us choose rancher:

- Easy Kubernetes deployment
- Easy Cluster Operation & Workload Management
- Very nice web GUI
- User and project management, authentication and policing
- Multi-Cluster support
- 100% Free and Open Source Software with Native Upstream Kubernetes

It comes with support for many of the requirements that BalticLSC needs. It has the project concept, where one project can include one or more namespace. It also comes shipped with security templates for network policies and more.

Rancher also uses annotations to add new namespaces to a project. So these need to be injected. This is discussed later in this document. It also comes with two templates for pod security policies, restricted and unrestricted. When deploying a new cluster with Rancher pod security policy support can be enabled under advanced options in the "Deploy Cluster" wizard.

[3] https://rancher.com

3.3.3 User isolation

For easier management of resources and resource quotas, the concept of *projects* is used. A project is a collection of namespaces. A resource quota can then be assigned on the project and a user assigned to the project. So, the BalticLSC Software user would be assigned a project with the appropriate resource quota and permissions.

The cluster owner set restrictions on how much resources BalticLSC Platform user can use. This is done on the project assigned to the user. Kubernetes implements resource quotas on namespace level so the user needs to assign part of the project quota to namespaces upon creation. If no quotas are assigned, defaults (set on a project) should be applied.

Permissions are controlled in Kubernetes with Role-based access control (RBAC). The recommended permissions for the BalticLSC user is

- Only allow namespaced scope, i.e. only allow control over namespaces created by the user
- The user needs a ClusterRole, with permissions to create namespaces

When creating a new namespace through the Kubernetes API there is no information available of who created the namespace. So the namespace cannot be assigned to a project with the appropriate permissions owned by the user. To be able to do this additional information must be added when the namespace is created. Here is an example how-to assign namespace *mynamespace* to project *myproject*.

```
An example
# Create Namespace for Feature
apiVersion: v1
kind: Namespace
metadata:
    name: mynamespace
annotations:
    field.cattle.io/projectId: myproject
```

One problem with this solution is that it is not transparent to the BalticLSC Platform user, it may not be a problem for BalticLSC Software to include this annotation but if the cluster would be shared with other users it could become a problem. To solve the problem with the mandatory project annotation, the Platform Manager could insert the correct project annotation for the user. This would be transparent for the user. The Platform Manager would intercept requests to the API resource for namespace creation and look up what project the user belongs to with a call to Kubernetes. What project a user is tied to can be stored in a custom resource. This same resource would be used by the controller acting on namespace creation events to know what permissions and roles should be bound to the namespace. The Platform Manager could also prevent users from creating namespaces in someone else projects. This would not pose a security risk because the namespace would be assigned with permissions only allowing the project owner to use the namespace. But it would create confusion.

3.3.4 Network isolation

All the BalticLSC user applications should be isolated from other user applications on a network level running in the cluster. The user applications should not be able to access other user network services. The user should only be able to access services running in the user project. Kubernetes can do this with network policies. To implement network policies, a network plugin compatible with the containers running must be installed. We have evaluated Canal/Calico which works well.