BalticLSC Platform Implementation Report

Version 1.0

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 Implementation Report

Work package	WP6
Task id	A6.1
Document number	O6.1
Document type	Technical document
Title	BalticLSC Platform Implementation Report
Subtitle	Implementation Report
Author(s)	Daniel Olsson (RISE), Marek Wdowiak (WUT)
Reviewer(s)	Krzysztof Marek (WUT)
Accepting	Michał Śmiałek (WUT)
Version	1.0
Status	**Final version**

Interreg Baltic Sea Region
EUROPEAN UNION
EUROPEAN REGIONAL DEVELOPMENT FUND

History of changes

Date	Ver.	Author(s)	Review	Change description
07.05.2021	0.1	Daniel Olsson (RISE)		Document creation and initial contents
11.05.2021	0.2	Daniel Olsson (RISE)		Added content
13.05.2021	0.21	Krzysztof Marek (WUT)	Marek Wdowiak (WUT)	Reviewed existing content
18.05.2021	0.3	Daniel Olsson (RISE)		Made changes according to review comments.
28.05.2021	0.4	Marek Wdowiak (WUT)	Krzysztof Marek (WUT)	Added Docker Swarm related content
10.06.2021	1.0	Krzysztof Marek (WUT)		Final version

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. The main technology partners are:

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

This work package, the implementation of BalticLSC Platform, is an important output of the BalticLSC project. It is based on the design that was created in work package 4.3 in the form of output document BalticLSC Platform Documentation (output 4.3) that contain:

- architectural model of the BalticLSC Platform
- decisions on which technologies (computing and networking hardware, operating system software, computing languages)

This document is a report describing the activities and results of the implementation of the prototype of the BalticLSC Platform. The report includes the description of the results of the activity: BalticLSC Platform code, the configuration and activities performed to set up the working BalticLSC Platform prototype at five partner sites.

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 give the reader a summary of what was done during the implementation phase. It consists mainly of two parts:

1. Implementation of the design
2. Support the installation and setup the finished implementation on five partner sites. Then share the site credentials with BalticLSC Software team to be able to bind the sites together into a distributed computing cluster.

1.2 Relations to Other Documents

The implementation is based on BalticLSC Platform Technical Documentation (BalticLSC Output 4.3), which in turn 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. The implementation is supporting all requirements in BalticLSC Platform User Requirements (BalticLSC Output 3.2).

The BalticLSC Platform is used by the BalticLSC Software to be able to perform computations. It is thus also related to BalticLSC Software Design (BalticLSC Output 5.2) and the corresponding implementation (BalticLSC Output 6.2).

1.3 Intended Audience and Usage Guidelines

This document is the Output 6.1 as described in the BalticLSC Application Form and is mainly intended for internal use within BalticLSC for reporting purposes for local First Level Control and Baltic Sea Region Program. The document is not intended as-is for external use, but parts of it can be used in the external document: BalticLSC Handbook which should contain all information needed for interested parties to use and further improve the BalticLSC Platform.

2. The code of the BalticLSC Platform

This section goes through the code structure and contents. The source code produced has been committed to the common project source code repository located at:

https://www.balticlsc.eu/gitlab/

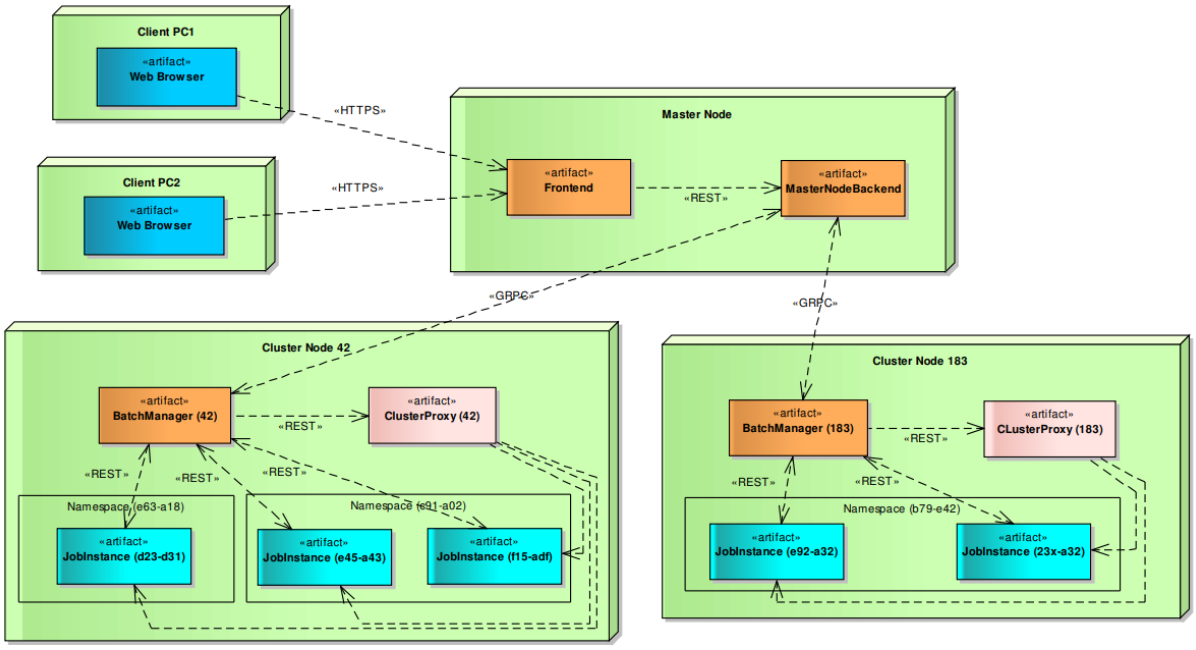

Figure 1 Deployment architecture

2.1 Code for Rancher clusters

All code related to implementation of feature that allows to connect Rancher cluster to Baltic network can be found in presented git repository. Following is the repository structure for that part:

```
.
├── cluster-proxy
│   ├── bin
│   ├── client
│   ├── clusterproxy
│   ├── proto
│   ├── rancher-go -> ../upstream/rancher-go
│   └── server
├── deployment
├── docs
│   └── img
├── ingress-rule-conflict-controller
│   ├── conf
│   └── img
├── namespace-controller
│   ├── deployment
```

```
│   ├── rancher-go -> ../upstream/rancher-go
│   ├── scripts
│   ├── server
│   └── test
├── nvidia-drivers
│   └── build
├── resource-monitoring
│   ├── operator-metering
│   └── prometheus
├── tests
│   └── boogeyman
└── upstream
    ├── operator-metering
    └── rancher-go
```

2.1.1 The code in numbers

Following table summarizes the repository contents. It was produced by an open-source tool called cloc[1] which counts blank lines, comment lines, and physical lines of source code in many programming languages. Note that the summary does not include all code of the BalticLSC Platform, only new code needed to complement the existing open-source components implementing the BalticLSC Platform. For instance, Rancher, Kubernetes and libraries.

Language	files	blank	comment	code
Go	26	811	216	5403
Markdown	13	497	0	2211
YAML	57	6	32	1675
Bourne Shell	26	95	47	329
Protocol Buffers	1	22	3	140
make	4	20	15	40
Dockerfile	5	14	13	33
SUM:	132	1465	326	9831

2.1.2 Implementation of Cluster Proxy for Rancher

ClusterProxy is a stateless proxy used to translate ClusterProxyAPI to BalticLSC Platform API. It manages the lifecycle of the computation containers executed on the BalticLSC Platform. The northbound interface called ClusterProxyAPI, used by the BalticLSC Software

[1] https://github.com/AlDanial/cloc

is implemented using gRPC. The southbound interfaces communicate with the BalticLSC Platform APIs:

- Rancher REST API
- Kubernetes REST API

Figure 1 shows the placement of the ClusterProxy in the deployment architecture. Note that **Cluster Node** refer to one instance of BalticLSC Platform (cluster of nodes). Also note that the protocol in figure between BatchManager and ClusterProxy is gRPC, not REST.

This component is not part of the design of the BalticLSC Platform. It is a BalticLSC Software component that RISE helped to implement.

Path: /cluster-proxy

2.1.3 Namespace controller

Namespaces are global; thus, conflicts can happen. If someone tries to create a namespace that already exist, he will get an error back. A suggestion would be to require users to add a prefix to all namespaces that are created. When creating a namespace via kubernetes API (kubectl), it is mandatory to add annotation (field.cattle.io/projectId: [clusterId]:[projectId]) to the spec which binds this new namespace to the users project. The problem is that it is possible to create a namespace without this annotation, but the user is then not allowed to access it. One solution that is to delete all namespaces that are created without any annotation. Another and better solution would be to add this annotation upon namespace creation. To solve the above, a software component called “Namespace Controller” was implemented. Its main tasks are:

1. To watch for namespace events and make sure that namespaces get associated with the user’s project www.balticlsc.eu
2. Remove annotation references to wrong projects, or just make sure an error is returned to user This controller must be installed in the “Rancher cluster” to be able to access all information needed. Note: When creating namespace via Rancher API or Rancher UI then it is put inside the correct project.

Path: /namespace-controller

2.1.4 Deployment scripts

Scripts to deploy instance of BalticLSC platform written in ansible.

Path: /deployment

2.1.5 Nvidia drivers

To be able to use Nvidia GPUs drivers needs to be installed on every worker node that has GPUs. The driver should be as new as possible to support the newest CUDA libraries. Note that the driver supports all older versions of CUDA.

Path: /nvidia-drivers

2.1.6 Ingress rule conflict controller

In Kubernetes it is possible to create two ingresses with the same host rule (ex: service.foo.com). What happens with nginx ingress controller is that the second (conflicting) rule is ignored. However, the second ingress object is created, and it is not possible to see that provisioning failed. And when you are not in control of the whole Kubernetes cluster (shared cluster), it is not possible to see the conflicting rule. This controller adds missing ingress rule conflict handling to Kubernetes API which return error upon conflict.

Path: /ingress-rule-conflict-controller

2.1.7 Boogeyman

Boogeyman is a test tool created to stress test the BalticLSC Platform. For instance, it can randomly kill pods at random times to test the stability and recovery of the cluster. It can also create and delete persistence volume claims to test storage performance under heavy load.

The BalticLSC platform was tested by configuring Boogeyman to spawn a large number of producer pods that wrote data to a number of persistent storage volumes. A corresponding set of consumer pods was spawned to read the data generated by the producers. During the test, Boogeyman acted as a chaos monkey and randomly kill producers and the consumers. As Kubernetes automatically re-spawned killed pods, it was still possible to process all generated data despite pods being randomly killed. The Boogeyman tool was implemented in Golang and used the official K8s client-go library to interact with the Kubernetes system.

Path: /tests/boogeyman

2.1.8 Reporting framework (Operator metering)

Operator Metering framework enables custom usage reporting derived from monitoring data that can be used for our purposes. It is suggested to generate reports in following intervals:

- Hourly
- Daily
- Monthly

The usage breakdown reports should be by namespace, which can then be used to summarize project/user utilization for a given period. In addition, it would be nice to summarize the namespace reports into user reports. The reports should be created on following metrics:

- CPU request – Amount of reserved CPU
- CPU usage – Real CPU usage (recorded)
- Memory request – Amount of reserved memory
- Memory usage – Real memory usage (recorded)
- Storage request – Amount of reserved storage
- Storage usage – Real storage usage
- GPU request – Number of reserved GPUs of different types

Path: /resource-monitoring/operator-metering

2.1.9 Prometheus metering custom metrics

For smaller clusters it can be overkill to use operator-metering which can handle clusters of thousands of nodes. In these cases, Prometheus monitoring is enough. However, to be able to deliver the reports per project that we want, some custom metrics are needed. This directory contains what is needed to run only Prometheus as reporting source.

Path: /resource-monitoring/prometheus

2.2 Code for Docker swarm clusters

Computation clusters with few machines can be managed with simpler software like Docker Swarm and Portainer. For those clusters, separated implementation of Cluster Proy API was prepared.

Code that implements functionality that allows connecting Docker Swarm clusters to BalticLSC networks implements the same architecture, that was shown in Figure 1. All code that was implemented can be seen at given git repository. Repository structure consists with .NET web application and looks as follows:

```
.
├── appsettings.json
├── baltic_clusterproxy_build_push.ps1
├── Baltic.Node.ClusterProxy.Swarm.csproj
├── Baltic.Node.ClusterProxy.Swarm.Production.dockerfile
├── Baltic.Node.ClusterProxy.Swarm.Production.dockerfile.dockerignore
├── Controllers
│   └── DebugController.cs
├── Program.cs
├── Properties
│   └── launchSettings.json
├── Services
│   ├── DockerApiWrapper.config.cs
│   ├── DockerApiWrapper.container.cs
│   ├── DockerApiWrapper.main.cs
│   ├── DockerApiWrapper.network.cs
│   ├── DockerApiWrapper.service.cs
│   ├── DockerApiWrapper.volume.cs
│   ├── DockerSwarmHelper.cs
│   ├── IPortainerApi.cs
│   ├── ISwarmProxy.cs
│   ├── PortainerApiWrapper.cs
│   └── SwarmClusterProxyService.cs
└── Startup.cs
```

2.2.1 The code in numbers

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C#                              14            205             31           1964
MSBuild script                   1              7              0             23
JSON                             2              0              0             20
PowerShell                       1              0              0              2
-------------------------------------------------------------------------------
SUM:                            18            212             31           2009
-------------------------------------------------------------------------------
```

2.2.2 Implementation of Cluster Proxy for Docker Swarm

ClusterProxy is a stateless proxy used to translate BalticLSC Platform API to specific cluster API. Docker Swarm implementation of *ClusterProxy* manages the lifecycle of the computation containers executed on the BalticLSC Platform that was started on Docker Swarm cluster. The northbound interface called ClusterProxyAPI, used by the BalticLSC Software is implemented using gRPC. The southbound interfaces communicate with the:

- Docker REST API
- Portainer REST API

This component is part of the design of the BalticLSC and serve as reference to another implementations of Cluster Proxy API. This component was implemented by WUT.

2.2.3 Services

The git repository folder services contain *.cs files with C# implementation of Docker REST API, Portainer REST API, and ClusterProxy gRPC API. Files that start with "DockerApiWraper" are responsible for translating gRPC requests to REST requests of Docker API which allows to create, delete, and manage jobs on computation cluster. File PortainerApiWrapper.cs contains an implementation of Portainer REST API which allows checking the state of a given job that was started on a cluster.

2.2.4 Debug controller.

Debug REST controller is not available in a normal production environment. It becomes available when ClusterProxy was started in developer mode. That allows testing ClusterProxy features without gRPC requests.

3. BalticLSC Network

The second part of the work package was to install and setup the BalticLSC Platform on all 5 partner sites:

1. Warsaw, Poland (WUT)
2. Riga, Latvia (IMCS)
3. Luleå, Sweden (RISE)
4. Turku, Finland (MTC)
5. Tallin, Estonia (TSP)

3.1 Hardware procurement

Each partner got a budget of around 30k Euro to procure hardware. Document O4.2 BalticLSC Platform Component Selection Report contain following suggestions:

Master node / Low performance compute node

- 1U server
- 2x CPU
- 32GB RAM
- SSD
- 10Gb Ethernet

Medium performance compute node

- 2U server
- 2 x CPU
- 128 GB RAM
- 1 x Nvidia GPU
- SSD
- 10Gbps Ethernet

High-performance GPU compute node

- Supermicro 4029GP-TRT3 server chassis
- Single root PCIe architecture
- 2 x Intel Xeon Gold 6130
- 256 GB RAM
- 4TB SSD
- 8 x Nvidia GTX 2080ti
- Infiniband 56 Gbps

The document also contains examples of cluster configurations which gave the partners advice of how many servers they need to procure. Following is an example of a full production cluster setup:

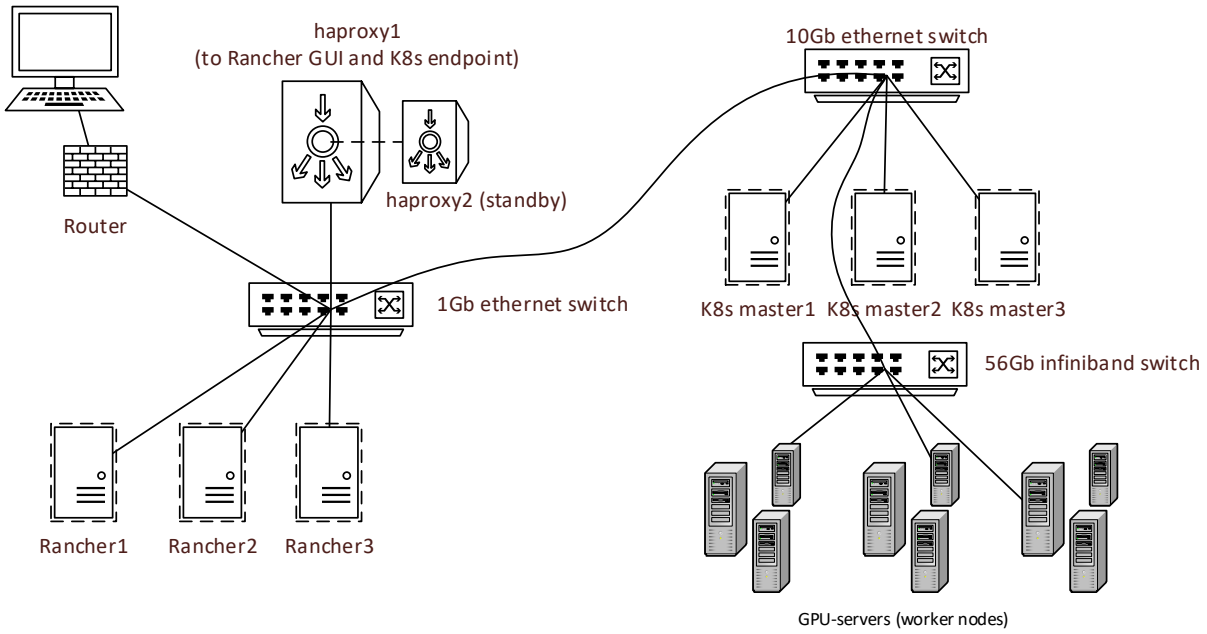

Figure 2 Production cluster

3.2 Installation

Installation was done by each partner using the installation instructions documented in BalticLSC Platform Technical Documentation (O4.2). With support from RISE all sites were successfully installed. The instructions are included in this chapter.

3.2.1 Installation of Rancher and Kubernetes

Because Rancher is an external component, we recommend using the official Rancher installation documentation that can be found here https://rancher.com/docs/rancher/v2.x/en/installation/.

But we have created a step-by-step video that goes through installation and setup that can be used when installing BalticLSC Platform in an early stage. The installation will be improved after the Platform is fully implemented and new instructions will be provided in the BalticLSC Handbook.

Video link: https://youtu.be/e__ss4SA4hY
Content:

1. Deployment of nodes

2. Preparing nodes
3. Installing Rancher
4. Installing Kubernetes using rancher
5. Configuring monitoring
6. Configuration and setup of a restricted user with custom security profile and resource quotas
7. Installation and setup of kubectl. Access via API for admin and restricted user

3.2.2 BalticLSC settings in Rancher

This section describes how to manually setup Rancher to support BalticLSC. It is also documenting how Rancher and Kubernetes is setup to support the requireemnts that BalticLSC Platform has. Not all requirements are fulfilled by just Kubernets and Rancher, therefore there are some new software components that needs to be implemented. These will be covered later in this document. The setup involves creating users, projects, security profiles and their associations, as well as how to setup resource quotas and limits.

3.2.2.1 Custom security policy

To restrict users from being able to do harm, a custom **pod security policy** is created which is bound to the user's project (shown in 3.2.2.2). The policy should only deny access to critical capabilities and resources. Thus, the user should not be able to do harm to the cluster and should not be restricting in normal operation. It is not an easy task to define an unrestricting policy that is still protecting the cluster. Following are the recommended policy settings:

- Name: rise-custom
- Basic Polices: No on all
- Capability Policies:
    - Allowed Capabilities: CHOWN , NET_BIND_SERVICE , NET_ADMIN, NET_BROADCAST , NET_RAW , SETGID , SETUID
    - Default Add Capabilites: CHOWN , NET_BIND_SERVICE , NET_BROADCAST , NET_RAW , SETGID , SETUID
- Required Drop Capabilities: None
- Volume Policy: emptyDir, secret, persistentVolumeClaim, downwardAPI, configMap, projected
- Allowed Host Paths Policy: None
- FS Group Policy: RunAsAny
- Host Ports Policy: None
- Run As User Policy: RunAsAny
- SELinux Policy: RunAsAny
- Supplemental Groups Policy: RunAsAny

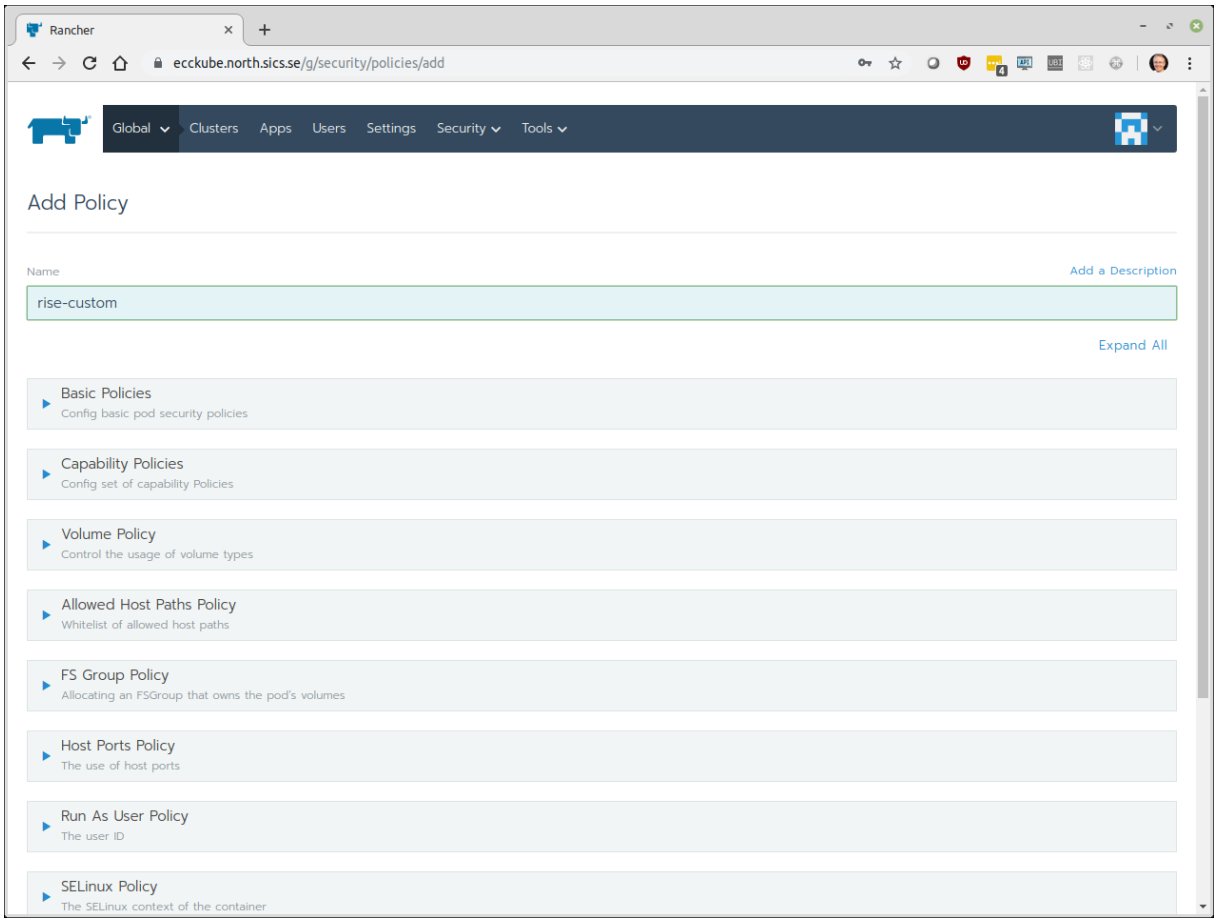

Figure 3: Adding custom security policy

3.2.2.2 Create user with associated project and resource quota

In this example we add an account for user *bob@mail.com* in Rancher. First, we create the user and fill in following:

Username: balticlsc

Password: **********

Display Name: BalticLSC Software User

Global Permissions:

- Custom
  - Use Catalog Templates
  - Login Access

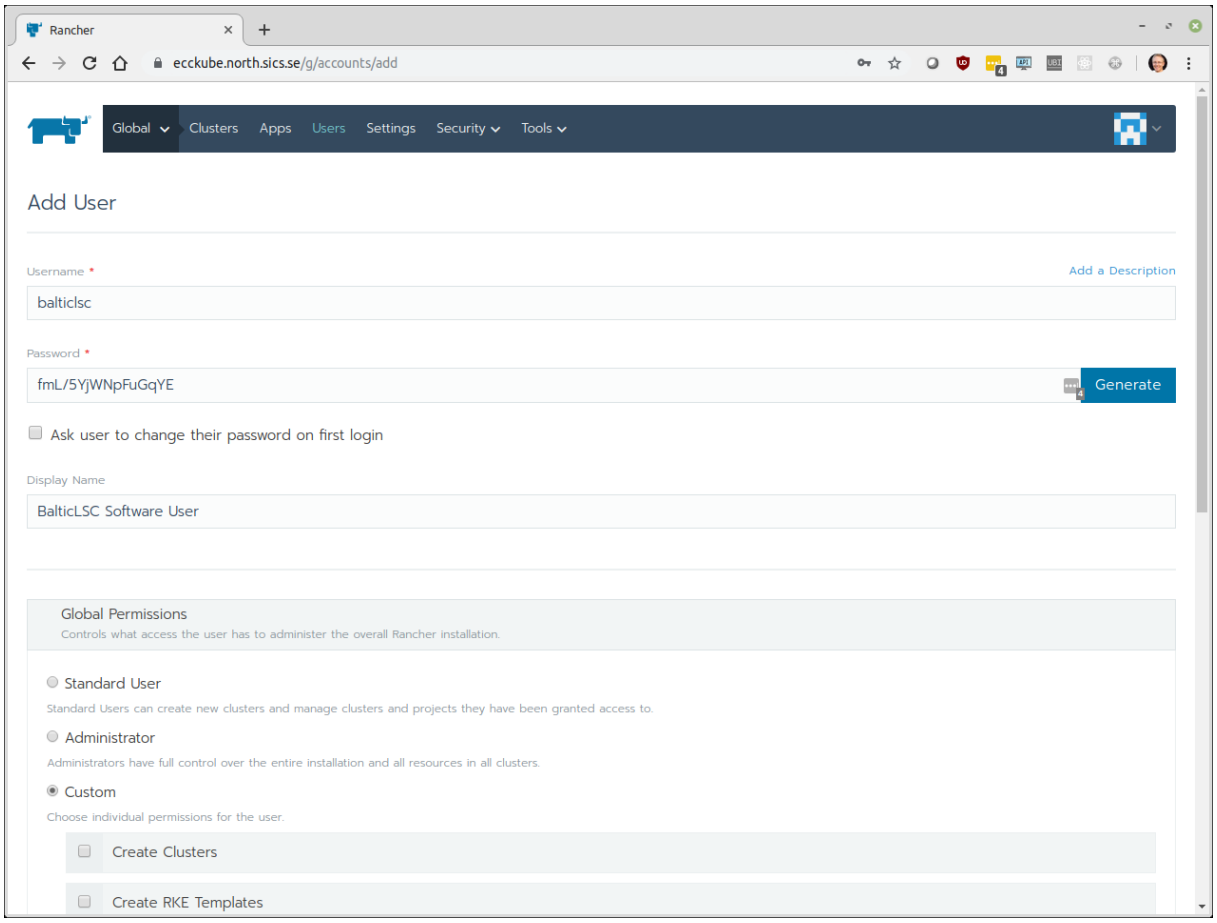

Figure 4: Screenshot adding user in Rancher

Then we add a new project with the user above as a project member:

- Project Name: balticlsc
- Pod Security Policy: rise-custom
- Add user balticlsc as project member
- Resource Quotas

Resource type	**Project**	**Default namespace**
CPU Limit	100000 mCPUs	10000 mCPUs
CPU Reservation	100000 mCPUs	10000 mCPUs
Memory Limit	256000 MiB	25600 MiB
Memory Reservation	256000 MiB	25600 MiB
Persistent Volume Claims	50	5
Storage Reservation	10000 GB	1000 GB

- Container Default Resource Limit

Resource type	
CPU Limit	2000 mCPUs
CPU Reservation	2000 mCPUs
Memory Limit	256 MiB
Memory Reservation	256 MiB

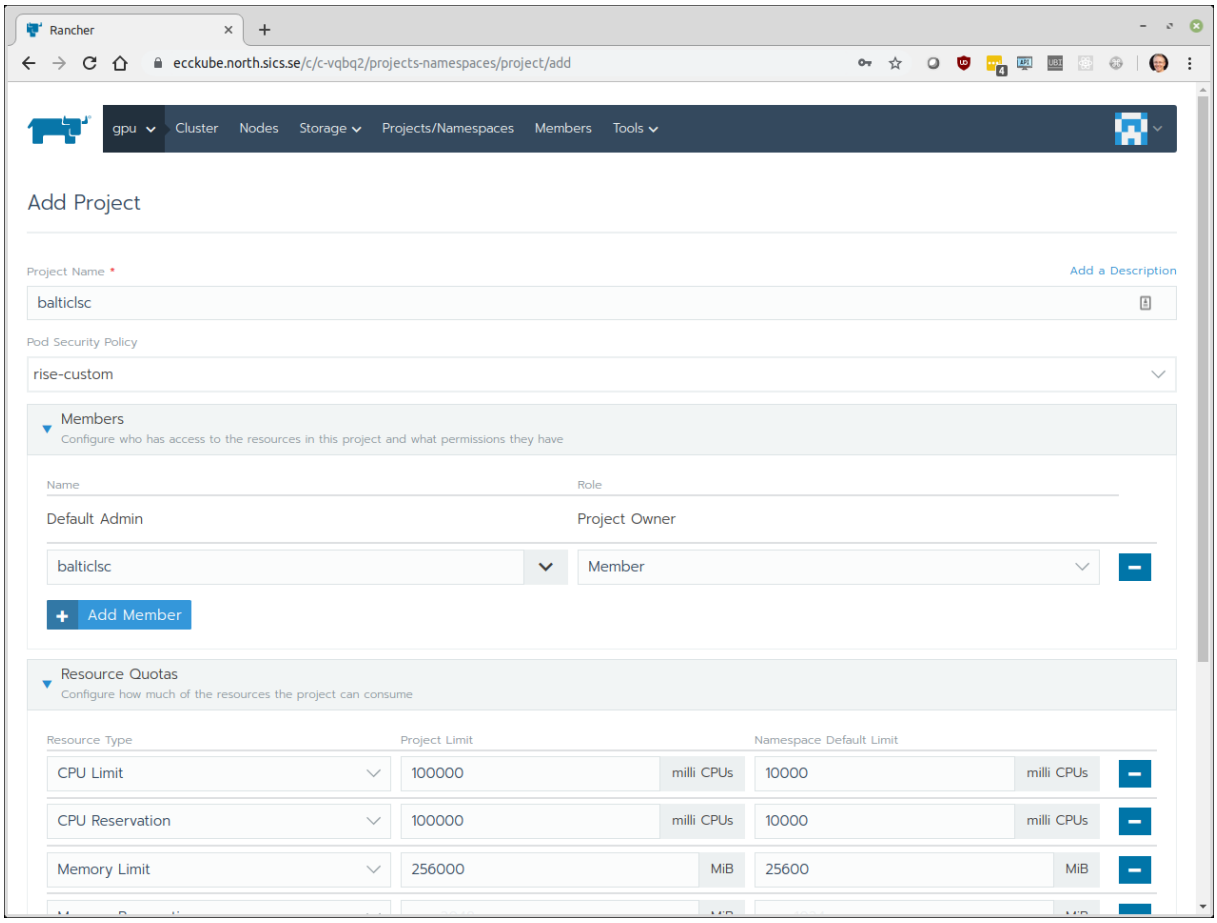

Figure 5: Adding project for baliclsc user

3.3 Docker Swarm and Portainer installation (alternative to Rancher)

Depending on the operating system on cluster machines proper Docker drivers need to be installed according to official Docker documentation https://docs.docker.com/.
After installing Docker drivers Docker Swarm cluster need to be set up.
One machine will be designated as the cluster manager. On this machine following command needs to be entered in the command line.

```
docker swarm init
```

As a result token and command that allows other machines to join cluster will be displayed:

```
Swarm initialized: current node (bvz81updecsj6wjz393c09vti) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-3pu6hszjas19xyp7ghgosyx9k8atbfcr8p2is99znpy26u2lkl-
1awxwuwd3z9j1z3puu7rcgdbx \
    172.17.0.2:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.
```

Join command needs to be run on all machines in the cluster using command line.

After successfully creating Docker Swarm Cluster Portainer software need to be installed on that cluster. Steps presented on official Portainer documentation https://documentation.portainer.io/ need to be performed on the machine which was selected as the cluster manager. Which contains running to commands in the command line:

```
curl -L https://downloads.portainer.io/portainer-agent-stack.yml -o portainer-agent-stack.yml

docker stack deploy -c portainer-agent-stack.yml portainer
```

After installing Portainer software a new Portainer user needs to be created. The last step is to start ClusterProxy in that cluster by deploing following yml file:

```
version: '3.7'

services:
  balticlsc-node:
    hostname: balticlsc-node
    image: balticlsc/balticlsc-node:local
    ports:
      - 7000:7000
      - 7001:7001
    networks:
      balticlsc-node:
        aliases:
          - balticlsc-node
          - balticlsc-node.balticlsc-node
    environment:
      masterHost: balticlsc.iem.pw.edu.pl
      masterPort: 5001
      nodePort: 7001
      clusterProxyUrl: https://cluster-proxy:6001
      clusterProjectName: balticlsc
      batchManagerUrl: https://balticlsc-node
      NodePublicHost: https://public-url-for-your-node:7001
    volumes:
      - type: bind
        source: ./logs/balticlsc-node
        target: /app/logs
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints: [node.role == manager]
    logging:
      driver: fluentd
      options:
        tag: balticlsc-node

  cluster-proxy-swarm:
    hostname: cluster-proxy
    image: balticlsc/balticlsc-cluster-proxy-swarm:local
    ports:
      - 6000:6000
```

```
      - 6001:6001
    networks:
      balticlsc-node:
        aliases:
         - cluster-proxy
         - cluster-proxy.balticlsc-node
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /var/lib/docker/volumes:/var/lib/docker/volumes
      - type: bind
        source: ./logs/cluster-proxy-swarm
        target: /app/logs
    environment:
      projectPrefix: balticlsc
      batchManagerNetworkName: batch-manager
      portainerUsername: portainer_username
      portainerPassword: portainer_passowrd
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints: [node.role == manager]
    logging:
      driver: fluentd
      options:
        tag: cluster-proxy-swarm


networks:
  balticlsc-server:
    external: true
  balticlsc-node:
    external: true
```

Remember to change `portainerUsername` and `portainerPassword` environment variable to match the username and password that was created for Portainer.

3.4 Sharing access credentials

When the installation was done the credentials needed to be shared with RISE in a secure way. This step will not be needed when BalticLSC Software support adding new provider clusters using the GUI. At the time it was needed for RISE to be able to inspect the installations and share the credentials with software team.

3.4.1 Sharing credentials with RISE

It was decided to use mail as a transport mechanism and GPG as encryption of the sensitive information to share credentials with RISE. The encrypted content was:

- Publicly reachable URL to Rancher GUI
- Username and password

3.4.2 Site credentials shared with WUT

WUT needed access to all sites to be added to BalticLSC Software (central management) as resource providers. RISE got SSH access to a virtual machine at WUT site. Credentials was uploaded using SCP (Secure Copy Protocol) to:

- Server: balticlsc-gateway.iem.pw.edu.pl
- Directory: /home/daniel/BalticLSC

3.5 Site configuration and status

This section summarizes the status of the sites as of 2021-06-10

3.5.1 WUT: Warsaw, Poland

Number of nodes in Cluster	6
Total number of CPU cores	80
Total RAM memory	768 GB
Number of GPUs	8

3.5.2 IMCS: Riga, Latvia

Number of nodes in Cluster	2
Total number of CPU cores	8
Total RAM memory	16 GB
Number of GPUs	0

3.5.3 RISE: Luleå, Sweden

Number of nodes in Cluster	30
Total number of CPU cores	1740
Total RAM memory	6936 GB
Number of GPUs (Nvidia GTX 1080ti / 2080ti)	216

3.5.4 MTC: Oulu, Finland

Number of nodes in Cluster	5
Total number of CPU cores	96
Total RAM memory	288 GB
Number of GPUs (Nvidia GTX 2080ti)	4

3.5.5 TSP: Tallin, Estonia

Number of nodes in Cluster	5
Total number of CPU cores	36
Total RAM memory	88 GB
Number of GPUs (Nvidia Tesla P100)	1