BalticLSC Handbook

The Handbook on using the BalticLSC Environment
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 Handbook

The Handbook on using the BalticLSC Environment

Work package	WP6
Task id	6.3
Document number	O6.3
Document type	Main Output
Title	BalticLSC Handbook
Subtitle	The Handbook on using the BalticLSC Environment
Author(s)	Michał Śmiałek, Kamil Rybiński, Krzysztof Marek, Marek Wdowiak, Daniel Olsson
Reviewer(s)	Radosław Roszczyk, Agris Sostaks
Accepting	Michał Śmiałek
Version	1.00
Status	Final version

Interreg
Baltic Sea Region
EUROPEAN UNION
EUROPEAN REGIONAL DEVELOPMENT FUND

History of changes

Date	Ver.	Author(s)	Change description
04.03.2020	0.1	Krzysztof Marek (WUT)	Document creation and initial contents
12.05.2020	0.2	Michał Śmiałek and Kamil Rybiński (WUT)	First version of Computation Module and CAL guides
05.06.2022	0.3	Daniel Olsson (RISE)	Platform and K8s instructions
03.07.2020	0.4	Krzysztof Marek (WUT)	First Video Tutorial
08.09.2020	0.5	Marek Wdowiak (WUT)	Docker Swarm instructions
09.03.2021	0.6	Krzysztof Marek (WUT)	Updated Video Tutorial
14.04.2021	0.7	Michał Śmiałek and Kamil Rybiński (WUT)	Update and extension to BalticLSC Software guides
07.06.2021	0.8	Michał Śmiałek (WUT)	BalticLSC Environment Introduction Video
08.12.2021	0.9	Krzysztof Marek (WUT)	Added missing content from other documents
09.12.2021	0.91	Agris Sostaks (IMCS)	Review
12.12.2021	0.92	Radosław Roszczyk (WUT)	Review
29.12.2021	1.0	Krzysztof Marek (WUT)	Final version

Executive summary

This Handbook is a collection of instructions, tutorials, step-by-step guides and practical examples on how to use the BalticLSC Environment. It covers areas of BalticLSC Environment use such as:

- Tips and guidelines on hardware selection for the BalticLSC Hardware
- Installation and configuration of the BalticLSC Platform
- Connection and configuration of BalticLSC Platform and Software
- Administration of the BalticLSC Environment
- Development of new Computation Modules
- Development of Computation Applications in CAL with the use of existing Computation Modules (including the specifically created ones)
- Performing computations on the BalticLSC Network using a Computations Application

Parts of this Handbook have been extended with YouTube tutorial videos. Is that cases links to the specific videos have been provided.

Table of contents

List of figures

List of tables

1. Introduction

1.1 Objectives and scope

The objective of this document is to gather all instruction materials created during the BalticLSC project on how to set up, operate and use the BalticLSC Platform and Software to perform advanced computations. It describes possible configurations of Computation Nodes, gives instruction on how to install required software, develop your own Computation Modules and teaches how to use the BalticLSC Software to perform computations.

1.2 Relations to Other Documents

This document remains in a very close relation with the joined outputs O5.2 – BalticLSC Admin Tool Technical Documentation, O5.3 – Computation Application Language Manual, and O5.4 – BalticLSC Computation Tool Technical Documentation which are the design artifacts of different components of the BalticLSC Software.

Another very important document is the joined outputs O4.3 – BalticLSC Platform Technical Documentation and O4.4 – BalticLSC Operating System Technical Documentation which are the design artifacts of different components of the BalticLSC Platform.

Another important documents are O6.1 – The BalticLSC Platform Implementation Report and O6.2 – BalticLSC Software Prototype. This documents describe the main outputs: the BalticLSC Platform and Software responsible for performing computations.

This document also derives from O4.2 – BalticLSC Platform Component Selection Report.

The last document related to O6.3 is O6.4 – BalticLSC Knowledge Transfer Guides that describes how to use the materials from O6.3 to transfer knowledge about BalticLSC to outside parties.

1.3 Intended Audience and Usage Guidelines

This document is intended for everyone interested in finding out how to use the BalticLSC Environment, either as a provider, administrator or an end user. The user should read selected parts of this documents, as the document covers multiple areas of BalticLSC and most of its users will be only interested in a part of it.

2. Installation, Setup and Administration – Commercial-grade Kubernetes based BalticLSC Platform Computation Node

2.1 Production cluster configuration

The production cluster consists of two Kubernetes clusters. One running Rancher and BalticLSC Platform components, called the Management Cluster. The other cluster is where all workloads are executed, called the Compute Cluster.

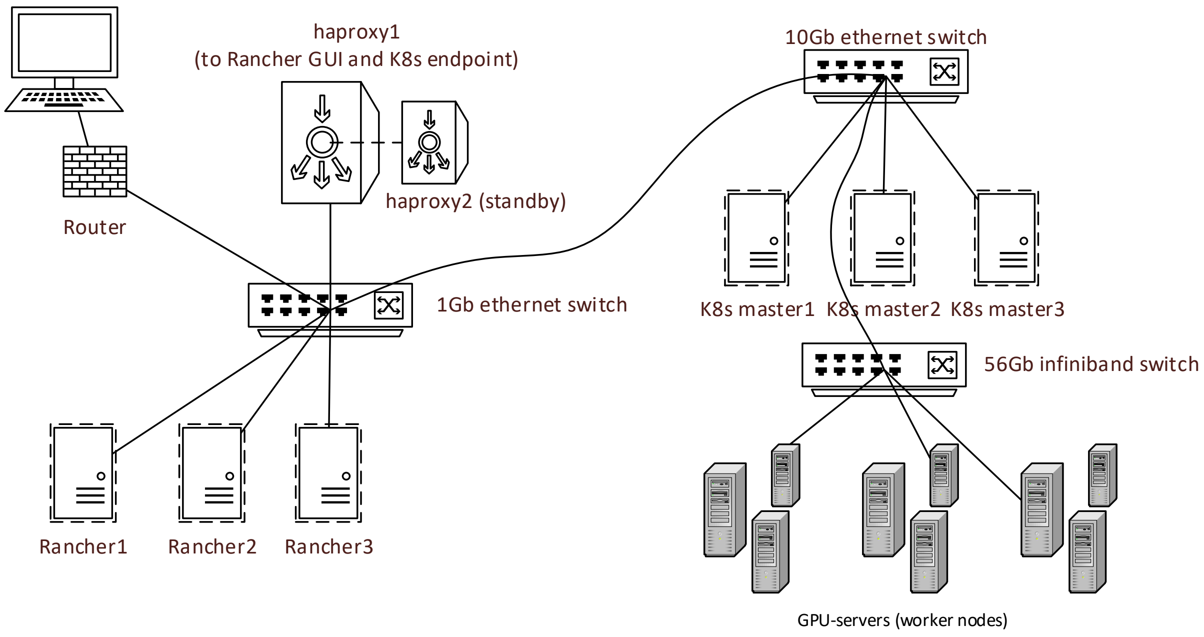


Figure 1: Production cluster topology

Imagine that the figure above is splitted vertically in the middle. Then the left side represents the Management Cluster with Rancher and BalticLSC Platform components. Everything to the right is then the Compute Cluster.

2.1.1 Management cluster

Rancher is running in its own Kubernetes cluster of three nodes. To provide load-balancing to the rancher nodes, a HA proxy can be used. This should also be configured in a highly available configuration. Details of this is omitted in this document. BalticLSC Platform specific components is also running in this cluster.

Hardware requirements on management nodes scale based on the total size of the clusters managed by Rancher. Provision each individual node according to the requirements in following table (taken from Rancher website[1]). If nodes are VMs, they need to run on separate physical hosts.

Table 1 Recomended Kubernates commercial-grade cluster configurations

Deployment size	Clusters	Nodes	vCPUs	RAM
Small	Up to 5	Up to 50	2	8 GB
Medium	Up to 15	Up to 200	4	16 GB
Large	Up to 50	Up to 500	8	32 GB
X-Large	Up to 100	Up to 1000	32	128 GB

[1] https://rancher.com/docs/rancher/v2.x/en/installation/requirements/

2.1.2 Compute cluster

It is recommended to use Rancher to install the compute cluster. The compute cluster consists of at least three master nodes and one or more worker nodes. Installation instructions is found later in this document.

Following server specifications can be used as inspiration when writing procurements.

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

1.1.1 Operating system requirements

Rancher is tested on the following operating systems and their subsequent non-major releases with a supported version of Docker[2].

- Ubuntu 16.04 (64-bit x86)
- Docker 17.03.x, 18.06.x, 18.09.x
- Ubuntu 18.04 (64-bit x86)
- Docker 18.06.x, 18.09.x
- Red Hat Enterprise Linux (RHEL)/CentOS 7.6 (64-bit x86)
- RHEL Docker 1.13
- Docker 17.03.x, 18.06.x, 18.09.x
- RancherOS 1.5.1 (64-bit x86)
- Docker 17.03.x, 18.06.x, 18.09.x

The recommendation is to run Ubuntu or CentOS because that is what we are experienced with. This concerns both Rancher and Compute nodes.

[2] https://docker.com

2.1.3 Development/minimal cluster configuration

For development and test there are almost no requirements on the hardware. It is possible to run everything on a standard consumer laptop. When running on laptop, and a cluster of several nodes is wanted, then each node should run in its own virtual machine. Choose your favorite hypervisor for running VMs, like KVM, VirtualBox or VMWare.

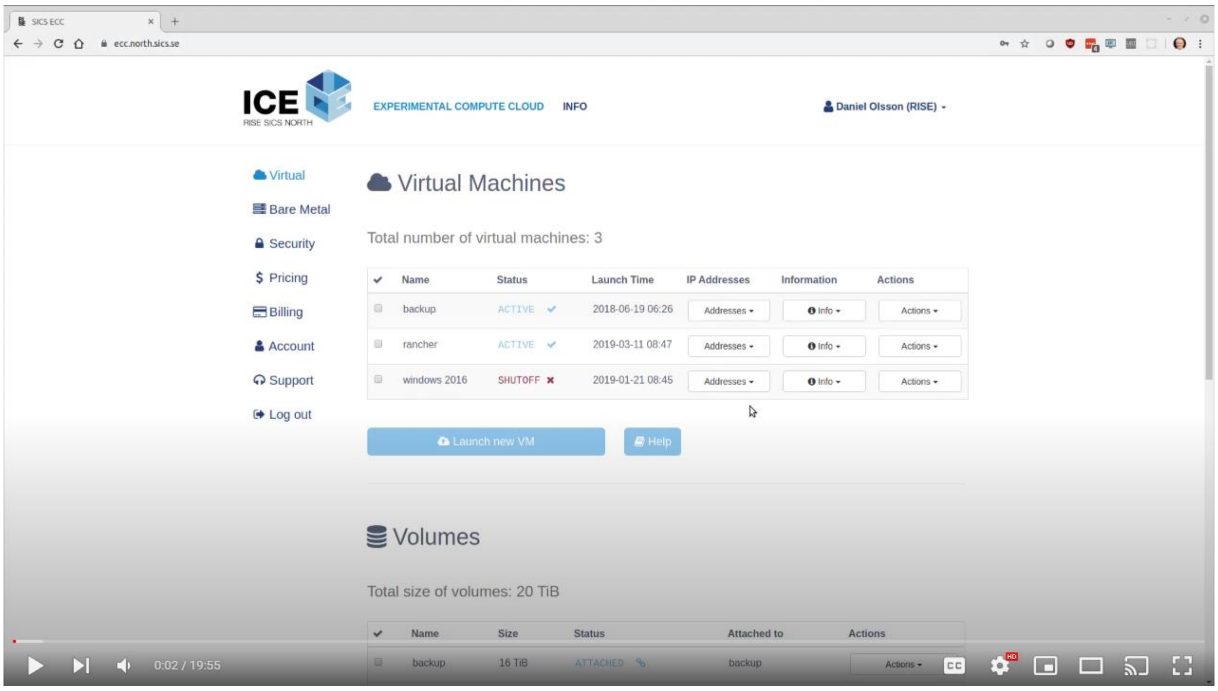
There are other more lightweight Kubernetes alternatives like Lightweight Kubernetes[3] which has very low system requirements. It has following minimum system requirements which means that it can even run on a Raspberry Pi.

- 512 MB of RAM per server
- 75 MB of RAM per node
- 200 MB of disk space

2.2 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/.

We have created a step-by-step video that goes through installation and setup that can be used when installing BalticLSC Platform.



Virtual Machines

Total number of virtual machines: 3

✓	Name	Status	Launch Time	IP Addresses	Information	Actions
	backup	ACTIVE ✓	2018-06-19 06:26	Addresses	Info	Actions
	rancher	ACTIVE ✓	2019-03-11 08:47	Addresses	Info	Actions
	windows 2016	SHUTOFF ✗	2019-01-21 08:45	Addresses	Info	Actions

Volumes

Total size of volumes: 20 TiB

✓	Name	Size	Status	Attached to	Actions
	backup	16 TiB	ATTACHED	backup	Actions

Figure 2 Rancher and Kubernetes installation video

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] https://k3s.io/

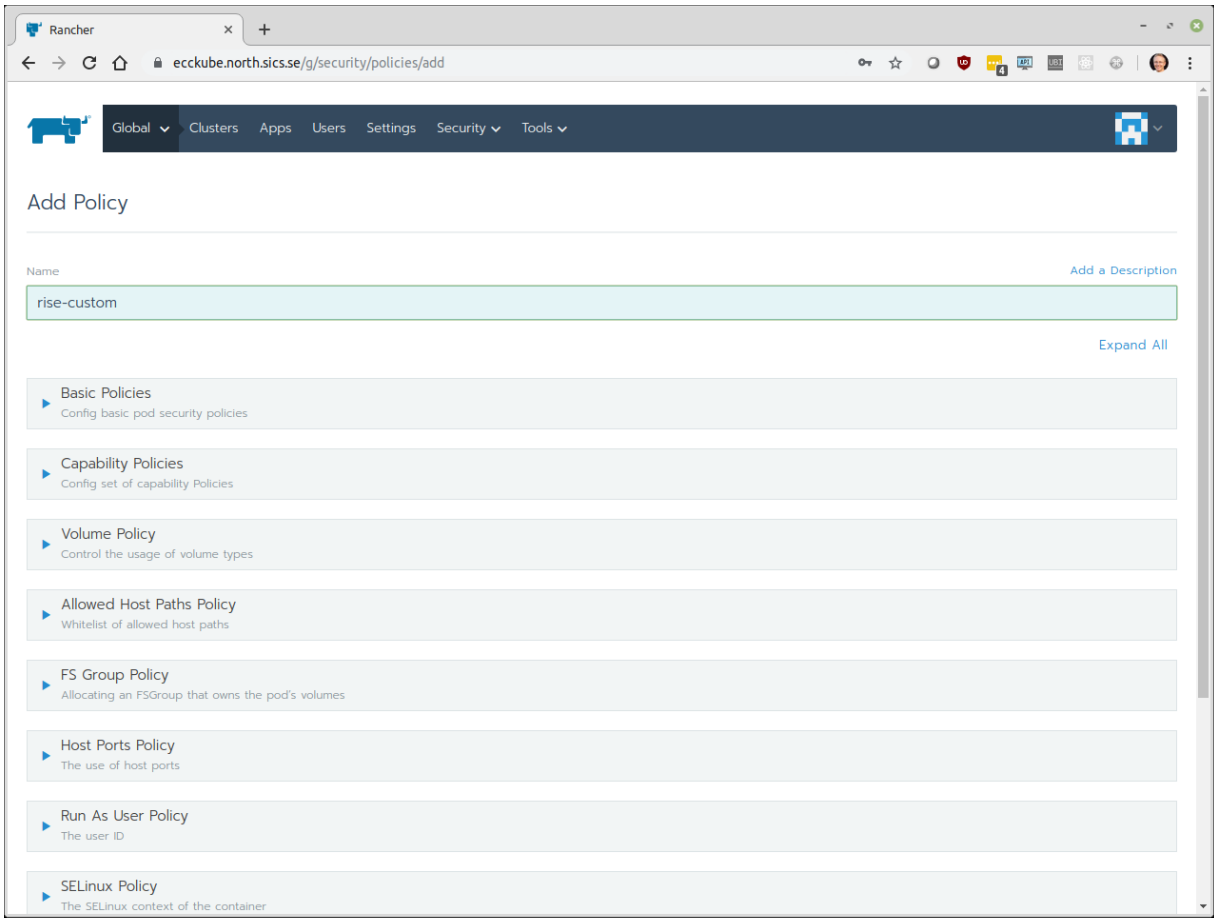
2.3 BalticLSC settings in Rancher

This section describes how to manually setup Rancher to support BalticLSC. It is also documenting how Rancher and Kubernetes is setup to support the requirements that BalticLSC Platform has. Not all requirements are fulfilled by just Kubernetes 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.

2.3.1 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. 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 is 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 , S ETUID
- 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


Rancher

ecckube.north.sics.se/g/security/policies/add

Global Clusters Apps Users Settings Security Tools

Add Policy

Name: rise-custom

Add a Description

Expand All

Basic Policies
Config basic pod security policies

Capability Policies
Config set of capability Policies

Volume Policy
Control the usage of volume types

Allowed Host Paths Policy
Whitelist of allowed host paths

FS Group Policy
Allocating an FSGroup that owns the pod's volumes

Host Ports Policy
The use of host ports

Run As User Policy
The user ID

SELinux Policy
The SELinux context of the container

Figure 3: Adding custom security policy

2.3.2 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:

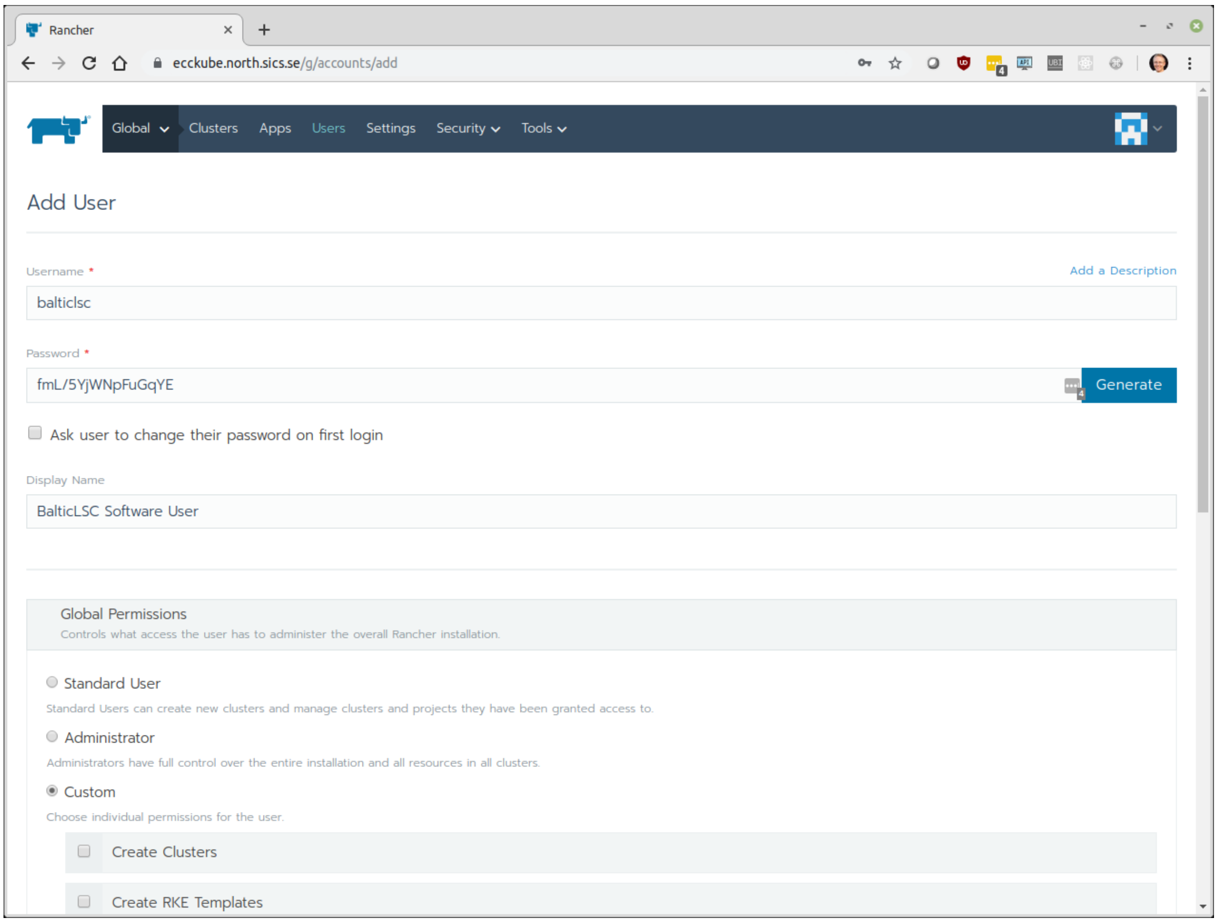
Username: balticlsc

Password: *********

Display Name: BalticLSC Software User

Global Permissions:

- Custom
  - Use Catalog Templates
  - Login Access


Rancher

ecckube.north.sics.se/g/accounts/add

Add User

Username * Add a Description

balticlsc

Password *

fmL/5YjWNpFuGqYE Generate

☐ Ask user to change their password on first login

Display Name

BalticLSC Software User

Global Permissions

Controls what access the user has to administer the overall Rancher installation.

○ Standard User

Standard Users can create new clusters and manage clusters and projects they have been granted access to.

○ Administrator

Administrators have full control over the entire installation and all resources in all clusters.

● Custom

Choose individual permissions for the user.

☐ Create Clusters

☐ Create RKE Templates

Figure 4: Screenshot adding user in Rancher

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

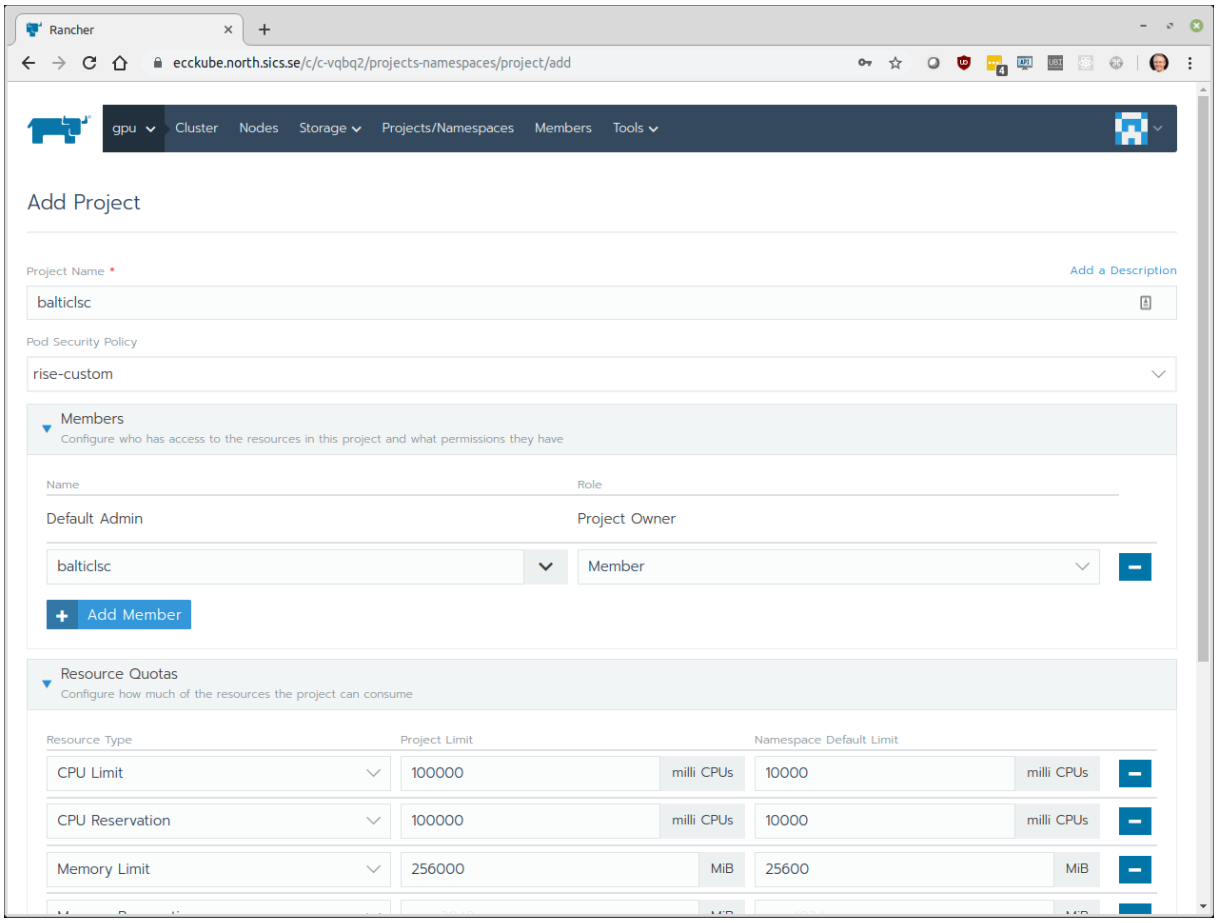
- Project Name: balticlsc
- Pod Security Policy: rise-custom
- Add user balticlsc as project member
- Resource Quotas

Table 2 Rancher resource setting

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

Table 3 Container Default Resource Limit

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



Add Project

Project Name * : balticlsc

Pod Security Policy: rise-custom

Members
Configure who has access to the resources in this project and what permissions they have

Name	Role
Default Admin	Project Owner
balticlsc	Member

Resource Quotas
Configure how much of the resources the project can consume

Resource Type	Project Limit	Namespace Default Limit
CPU Limit	100000 milli CPUs	10000 milli CPUs
CPU Reservation	100000 milli CPUs	10000 milli CPUs
Memory Limit	256000 MiB	25600 MiB

Figure 5: Adding project for baliclsc user

2.3.3 Cluster monitoring settings

Enable resource monitoring in Rancher with following recommended settings:

Table 4 Recommended Cluster monitoring settings in Rancher

Name	**Value**
Data retention	1080 hours
Enable Node Exporter	True
Enable Persistent Storage for Prometheus	True
Persistent Storage Size	500Gi
Grafana Persistent Storage	10Gi
Prometheus CPU Limit	2000 mCPUs
Prometheus Memory Limit	2000 MiB
Node Exporter CPU Limit	200 mCPUs
Node Exporter Memory Limit	200 MiB

GPU monitoring can be enabled using following Prometheus exporter. However, it is not necessary for the function of the system.

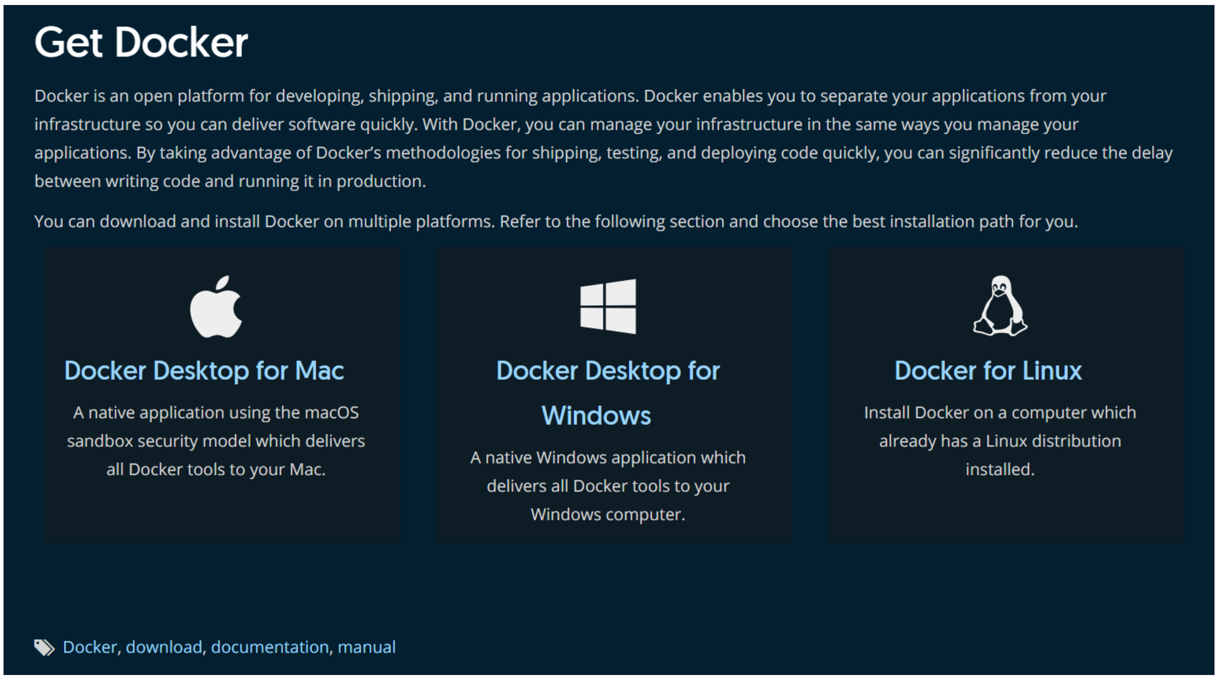
https://github.com/NVIDIA/gpu-monitoring-tools/tree/master/exporters/prometheus-dcgm

3. Installation, Setup and Administration – Lightweight Docker Swarm based BalticLSC Platform Computation Node

As Docker Swarm configuration is intended for small and experimental clusters, there is no specific configuration it was designed for. This procedure is the same for every hardware configuration.

3.1 Docker installation

The first step is to install Docker. To do this visit https://docs.docker.com/get-docker/ and follow the instructions for your platform (Windows, Mac and Linux are supported). Despite the fact that BalticLSC is currently supporting only Linux-based containers they can be run on every platform.


Get Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

You can download and install Docker on multiple platforms. Refer to the following section and choose the best installation path for you.

Docker Desktop for Mac
A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.

Docker Desktop for Windows
A native Windows application which delivers all Docker tools to your Windows computer.

Docker for Linux
Install Docker on a computer which already has a Linux distribution installed.

Docker, download, documentation, manual

Figure 6 Docker installation website

3.2 Docker Swarm Cluster Proxy Setup

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.

4. Computation Applications and Computation Modules

4.1 Introduction to Computation Modules

A ***Computation Module*** is a self-contained piece of software that performs a well-defined computation algorithm. It operates on ***Data Sets***, like files, folders, database collections, database tables etc. Access to these Data Sets is communicated using ***Data Tokens*** passed through so-called ***Data Pins***. Data Pins of type "input" accept Data Tokens that point to Data Sets provided by other modules or directly by the user. Data Pins of type "output" deliver Data Tokens that point to Data Sets provided by the current module.

Computation Modules should have many (at least one) input pins and can have many (maybe none) output pins. The case when there are no output pins is reserved for special modules that can communicate directly with data stores outside of the BalticLSC system. The module starts its operations when it receives a Data Token on every of its input pins. When the module finishes its computations (all or part), it sends a Data Token on a specific output pin.

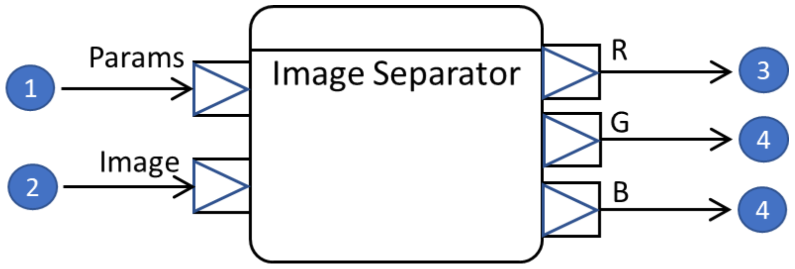


Figure 7 Example Computation Module with "single" Data Pins

This behaviour is illustrated in Figure 7. The module ("Image Separator") receives two tokens (denoted as "1" and "2") on its input pins (named "Params" and "Image"). When it finishes processing (executing a computation algorithm), it sends three tokens (denoted as "3", "4" and "5") to its output pins (named "R", "G" and "B").

The module in Figure 7 contains only "single" Data Pins. Such pins assume that only one token is passed through them for each execution of a computation algorithm. For instance, the example "Image Separator" accepts one "Image" token with an associated "Params" token. Then it processes this single image and produces one resulting token for each of the three image components ("R", "G" and "B").

Data Pins can be also defined as "multiple". This can pertain to "data multiplicity" or "token multiplicity". Pins with multiple data, accept or produce tokens that refer to data collections (e.g. folders vs. single files). Pins with multiple tokens accept or produce sequences of tokens within a single computation execution. A computation module with pins of the "multiple" kind is illustrated in Figure 8. The "Image Tokenizer" accepts a single token (denoted as "1") that refers to a collection (a folder with image files, cf. data multiplicity). Then, it produces multiple tokens (denoted as "2-1" and "2-2", cf. token multiplicity) that refer to individual image files. Note that pins can be "multiple" both for data and for tokens.

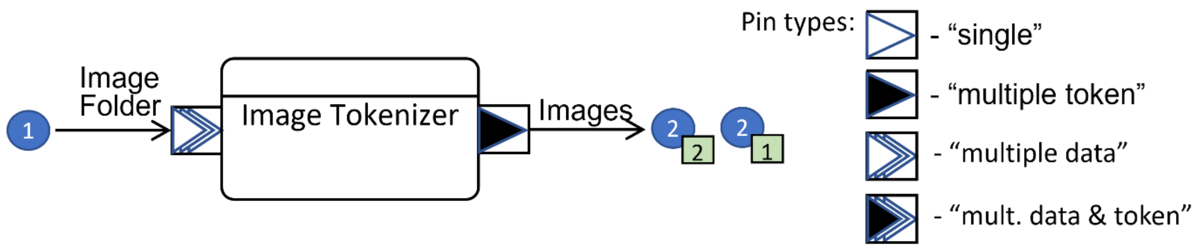


Figure 8 Example Computation Module with "multiple" Data Pins

Considering configuration and multiplicity of input and output pins, we can distinguish several reference module types. These types can be combined into hybrid types (e.g. splitter-joiner).

- Simple processor (one single input, one single output)
- Data separator (one single input, many single outputs)
- Data splitter (one single input, at least one token-multiple output)
- Data joiner (many single inputs, one single output)
- Data merger (at least one token-multiple input, one single output)

4.2 Programming in the Computation Application Language

In order to run a Computation Module, one needs to develop a Computation Application (CA). CAs are written in the Computation Application Language (CAL). The language is visual, and uses notation introduced in the previous section. We will explain CAL programming using two simple examples.

The first example is a very simple program, shown in Figure 9. The program contains a single ModuleCall, denoted by the rectangle with rounded corners. The rectangle has two compartments. The first compartment contains the call name (here: "detector", it can be any name that identifies this particular call). The second compartment contains the name of a particular Computation Module, selected from the list of available modules, that is called by this call (here: Face Detector).

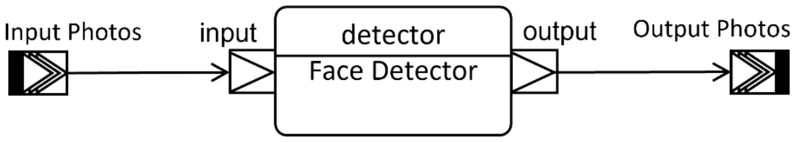


Figure 9 Simple CAL application with one Module Call and two External Data Pins

The Module Call in Figure 9 contains two "single" Data Pins ("input" and "output"), where their meaning was explained in the previous chapter. Apart from this, the program contains two External Data Pins ("Input Photos" and "Output Photos"). These pins refer to some data sources external to the execution environment. When the application is executed, the user can select specific credentials and location (e.g URL) where these data sources are situated.

Pins are connected through Data Flows denoted with arrows. Note, that in our first example, a "multiple data" pin ("Input Photos") is connected with a "single" pin ("input"). This means that the execution engine will process several data items (e.g. files) stored in a collection (e.g. a folder). For each such data item, a separate instance of the Face Detection module will be started. Each such instance will receive a token with a reference to a single data item (file) in the collection (folder). This means that the engine will possibly execute several instances of the Face Detector module in parallel, each processing a single data item (here: a photo file).

When any of the Face Detector module instances finishes processing its input data item, it sends a token to its "output" pin. This token will then be passed to the "Output Photos" pin. Note that the "Output Photos" is a "multiple data" pin, just like the "Input Photos" pin. This means that the data item will be sent to a collection (folder). In summary, the program in Figure 9 takes a single folder placed on some server managed by the user. It then sends photo files from that folder to several instances of the Face Detector module that run in parallel. While each of the module instances finish, the results of their work are placed in some (perhaps other) folder provided by the user.

Another, more complex CAL program is shown in Figure 10. The files in the "Input Images" folder are sent to several "Image channel Separator" module instances that run in parallel. Each such instance takes an "image" on its input and produces three images, sending three tokens – one on each of its output pins ("channel 1-3"). Each of these three images are in turn processed by a separate instance of the "Image Edger" module. When three "edger" instances finish their work, their "output" tokens are passes to an instance of the "Image Channel Joiner" module. This module takes the three "channel" images (produced by the "edgers"), combines them into a single image and sends a token on its output (the "image" pin). All the images produced by the instances of the "Image channel Joiner" module are then stored in the folder determined by the "Output Image" pin.

It should be noted that the program in Figure 10 can cause execution of many module instances in parallel. For example, let's consider running the application with 10 images stored in the "Input Images" folder. This will result in parallel execution of up to 10 instances of "Image Channel Separator", 30 instances of "Image Edger" and 10 instances of "Image Channel Joiner". Depending on the computation complexity and processing time for individual photos, it might happen that some instances of "Image channel Joiner" might start (and even finish) their tasks, while some of he instances of "Image Channel Separator" will still work. Also, it should be noted that the CAL execution engine will properly handle all the tokens flowing within the system. This means for instance, that the "channel" tokens for a single image will be separated and then joined appropriately, preventing from "mixing" tokens that "belong" to the same initial image.

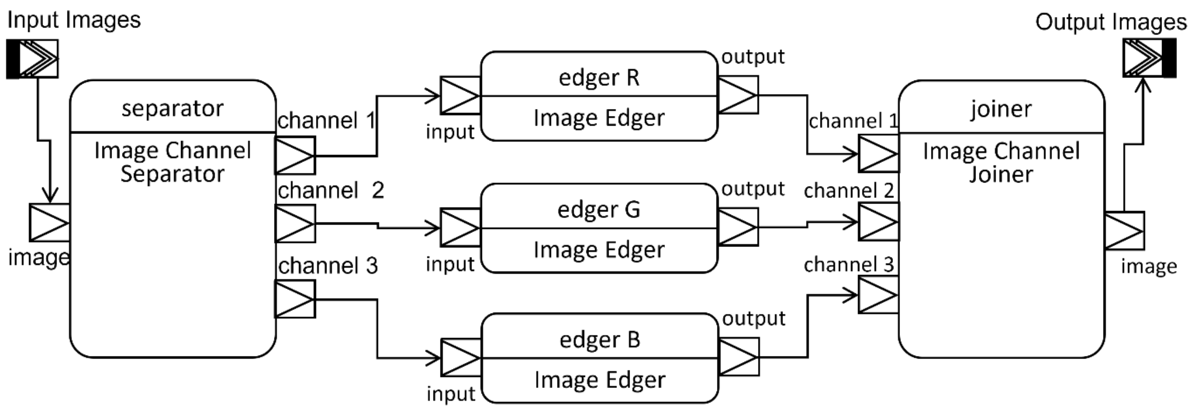


Figure 10 More complex CAL application with several Module Calls

4.3 Computation Modules within the BalticLSC system

In order for a Computation Module to operate within the BalticLSC system, it needs to be made compliant with specific deployment and communication rules. The most basic requirement is that it needs to be compiled and deployed as a Linux Docker container. It also needs to communicate with the BalticLSC Engine through specific REST interfaces (APIs). This allows to operate within the BalticLSC module execution environment that is illustrated in Figure 11.

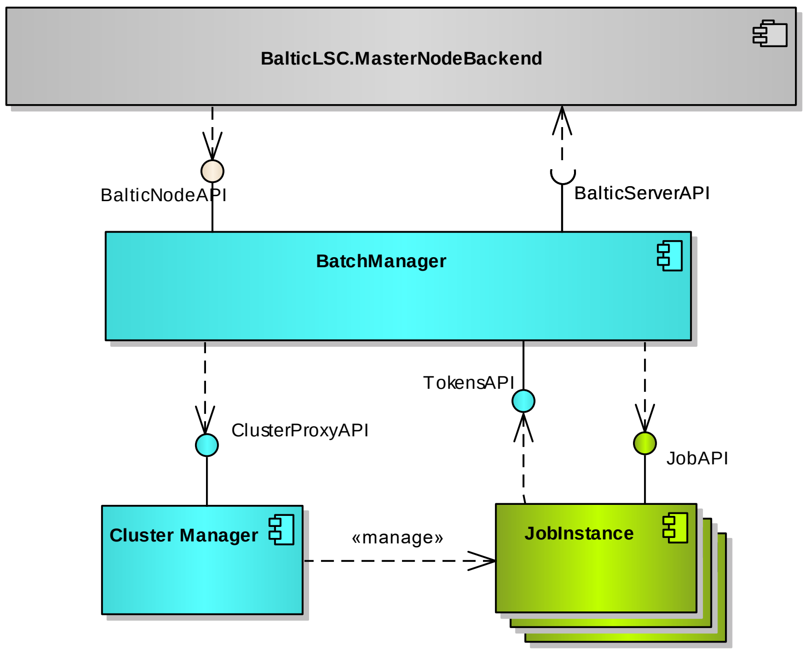


Figure 11 Component structure of the modules' execution environment

Computation Modules that work within the execution environment are called Job Instances. Each instance is executed on a specific Cluster Node and its instantiation and termination is managed through a specific Cluster Manager (using e.g. Kubernetes or Cluster Swarm technology). The instance lifecycle and token passing is managed by a separate component called the Batch Manager. The Batch Manager is an intermediary between the Job Instances and the central Master Node. It also instructs the Cluster Manager to start or finish the instances.

To properly communicate with the Batch Manager, the particular Computation Module should implement a simple REST API (“JobAPI”) consisting of two endpoints. It should also use a similarly simple API provided by the Batch Manager (“TokensAPI”). It should also read appropriate configuration data and access appropriate data stores.

In short, the code of your Computation Module should comply with the following lifecycle.

1. Read appropriate configuration data and set-up connections with the infrastructure (data stores, API endpoints, etc.)
2. Receive one or more Data Tokens on the JobAPI.
3. Access and start processing Data Sets based on the received tokens and input pin configuration.
4. Update existing or create new Data Sets based on output pin configuration.
5. When finished updating/creating some Data Set – send a Data Token to the TokensAPI.
6. When completed the computation execution – send an acknowledgement message for all the received tokens.
7. Reset all the internal states and prepare for potential processing of a next computation execution.

To comply with this lifecycle, you can use a template for C# and Python, presented in Section 5. This template hides all the technical details related to communication through the REST API and storing data in remote storages. Alternatively, Section 6 provides detailed information required to organize this communication without the use of the template.

5. Programming Computation Modules with a template

5.1 Available Sample Computation Modules

We have provided sample code of computation modules in C#, Java and Python. These sample are accessible through the public Git repository of the BalticLSC project. [Baltic Large Scale Computing (https://github.com/orgs/balticlsc/repositories)](https://github.com/orgs/balticlsc/repositories)

5.2 Template structure

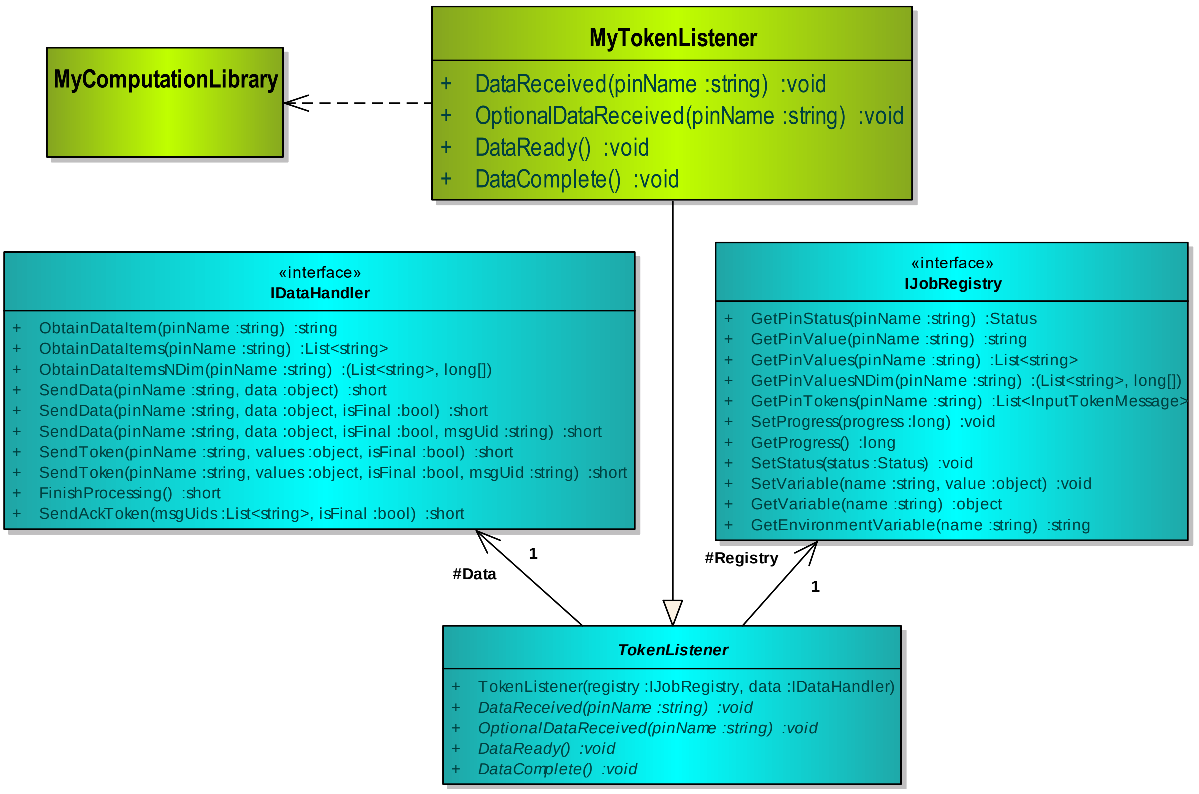

MyComputationLibrary
MyTokenListener
+ DataReceived(pinName :string) :void
+ OptionalDataReceived(pinName :string) :void
+ DataReady() :void
+ DataComplete() :void
«interface»
IDataHandler
+ ObtainDataItem(pinName :string) :string
+ ObtainDataItems(pinName :string) :List<string>
+ ObtainDataItemsNDim(pinName :string) :(List<string>, long[])
+ SendData(pinName :string, data :object) :short
+ SendData(pinName :string, data :object, isFinal :bool) :short
+ SendData(pinName :string, data :object, isFinal :bool, msgUid :string) :short
+ SendToken(pinName :string, values :object, isFinal :bool) :short
+ SendToken(pinName :string, values :object, isFinal :bool, msgUid :string) :short
+ FinishProcessing() :short
+ SendAckToken(msgUids :List<string>, isFinal :bool) :short
«interface»
IJobRegistry
+ GetPinStatus(pinName :string) :Status
+ GetPinValue(pinName :string) :string
+ GetPinValues(pinName :string) :List<string>
+ GetPinValuesNDim(pinName :string) :(List<string>, long[])
+ GetPinTokens(pinName :string) :List<InputTokenMessage>
+ SetProgress(progress :long) :void
+ GetProgress() :long
+ SetStatus(status :Status) :void
+ SetVariable(name :string, value :object) :void
+ GetVariable(name :string) :object
+ GetEnvironmentVariable(name :string) :string
#Data
1
#Registry
1
TokenListener
+ TokenListener(registry :IJobRegistry, data :IDataHandler)
+ DataReceived(pinName :string) :void
+ OptionalDataReceived(pinName :string) :void
+ DataReady() :void
+ DataComplete() :void

Figure 12 Classes and interfaces of the computation module template

The template structure is shown in Figure 12. The template is used through the three lower elements (IDataHandler, IJobRegistry and TokenListener). The two «interface» elements constitute the template's API. To implement a module using the template, you need to write your own class that specialises (inherits) the abstract TokenListener class. Obviously, this new class can refer to an appropriate computation library of your choice.

Your token listener class (e.g. MyTokenListener in Figure 12) should implement full contents of at least one of the four abstract methods declared by the TokenListener class. The choice of methods to be implemented depends on how we plan the module to start its operations. Each of the methods is called by the template code on a specific event related to receiving data on one or more input pins.

- DataReceived(pinName: string) – called when any single data item (token) has arrived at the pin with the provided name.
- OptionalDataReceived(pinName: string) – called additionally when the particular pin is optional.
- DataReady() – called additionally when all the non-optional input pins have received at least one data item (token); can be used for modules with many input pins to determine when the computations can be started.

- DataComplete() – called additionally when all the non-optional input pins have received all the expected data items (tokens); no additional data items should since be delivered to the non-optional input pins; to be used mainly in case when multiple token input pins exist in the module.

When implementing the above methods you need to use the template API through referring to Registry: IJobRegistry and Data: IDataHandler. The first one is used to store the computation state, including the arrived data items (tokens). The second one is used to access data related to the arriving data items (tokens). Example usages of these two interfaces are provided in Section 5.3 and their reference – in Section 5.4.

Finally, your token listener class should be registered with (injected into) the template code in order for the above methods to be invoked on the particular events. The template contains an example MyTokenListener class that is already registered. If you would like to use your own class, you need to use the registration code as presented in the code examples in Section 5.3.

5.3 Template usage

We will present the template usage through a simple example, providing code in C# and Python. This example implements the module presented in Figure 8. The appropriate code is shown in Listing 1 and Listing 2.

Listing 1 Example token listener code in C#

```
public class MyTokenListener : TokenListener
{

    // (...)


    public override void DataComplete()
    {
       Registry.SetStatus(Status.Working);

       string folder = Data.ObtainDataItem("Image Folder");
       string[] files = Directory.GetFiles(folder);

       Log.Debug($"Read folder: {folder}");

       for (int i=0; i<files.Length; i++)

       {

          Log.Debug(files[i]);

          Data.SendDataItem("Images", files[i], files.Length - 1 == i);
          Registry.SetProgress((i+1)/files.Length*100);

       }

       Data.FinishProcessing();

    }
}
```

Listing 2 Example token listener code in Python

```python
class MyTokenListener(TokenListener):



    # (...)


    def data_complete(self):
        self._registry.set_status(Status.WORKING)
        folder = self._data.obtain_data_item("Image Folder")

        files = list(f for f in listdir(folder) if isfile(join(folder, f)))

        logger.debug("Read folder:" + folder)

        for i in range(len(files)-1):

            logger.debug(files[i])

            self._data.send_data_item("Images", files[i],
                                      len(files)-1 == i)
            self.registry.set_progress((i+1)//len(files)*100)

        self._data.finish_processing()
```

In the listings, we implement just the DataComplete method. This is the default method to implement – when it is called, we are sure that all the data is ready for processing. Here, the module has just one input pin with single token multiplicity. Thus, we could alternatively implement the DataReceived or the DataReady method and it would be equivalent.

At the start we switch the job's status to "Working" by using the appropriate operation of the IJobRegistry interface. Further in code we report progress in computations by using the SetProgress (set_progress for Python) operation of the same interface.

The first step of the computations is to access the data. Here, we use the ObtainDataItem (obtain_data_item for Python) operation of the IDataHandler interface. This operation can be used for single token multiplicity pins only. It downloads the data to the local file store and returns the path to it (file or folder). For multiple token pins, you should use the ObtainDataItems or ObtainDataItemsNDim operations (see Section 5.4).

Note that the above operations should be used when the tokens contain references to files or folders. In case the data is simple and is wholly contained in the tokens, you should use operations from the IJobRegistry interface: GetPinValue, GetPinValues, GetPinValuesNDim. These operations could also be used if you would like to pass the access data directly and organize your data handling (e.g. downloading from an FTP server) manually.

After obtaining access to your data, you can start processing them. In our simple example, processing is limited to getting files from a folder (by using appropriate system libraries) and logging their names. This fragment also shows how to provide debug information through the logging mechanisms configured as part of the template.

Whenever some data item is ready, we send it to an output pin. In our example, we call the SendDataItem operation in a loop for each of the files in the input data folder. This operation is used for files or folders only. In case of data wholly contained in the tokens (or when you handle sending data manually), you should use the SendToken operation.

To mark the finalization of computations, we call the FinishProcessing operation. This operation cleans-up the job environment and also changes the job's status to "Completed" thus there is no need to change it manually. If you would like the job to clean-up after finishing processing of individual data items (tokens), you can use the SendAckToken operation. This can be used to optimize distributed processing (can start further jobs earlier). In our example we are processing single data items on input, so there was no point to do that.

Important: All the unhandled exceptions in your code will be reported to the BalticLSC execution environment and shown in the computation cockpit. Depending on the application or task settings this may cause the whole current task to be aborted automatically. To report failure in computations you can raise an exception and specify the exception message – this message will be shown to the user in the computation cockpit. This differs from setting the job status to Status.Failed. Just setting the status cannot cause aborting computations for the whole task and does not report the failure message in the computation cockpit.

The presented MyTokenListener class handles all the events related to incoming data. Obviously, you can extend your code with additional classes that would contain your computation code (algorithms, libraries, etc.). In order for your code to be integrated with the event handling mechanism, the MyTokenListener class has to be properly registered (injected). This is done by default within the example code. However, if you would like to change the name of the token listener class, you need to make sure to update the registration code.

In C#, you need to modify one line within the ConfigureServices method of the Startup class. This should involve just changing the name of the token listener class (change MyTokenListener to the name of your choice).

```
    services.AddScoped<TokenListener,MyTokenListener>();
```

In Python, you need to use the following line to initialize the event handling mechanism. Use the name of your class instead of MyTokenListener. In the example code, this line is already attached to the MyTokenListener class code.

```
    app = init_job_controller(MyTokenListener)
```

5.4 Template API reference

This reference describes all the operations of the template API, including those used less frequently. The list of operations uses the UML notation and C# naming convention. The descriptions for Python are identical but are provided in the template using appropriate Python naming conventions.

5.4.1 IJobRegistry

- **GetPinStatus(pinName: string): Status** – returns information about the tokens received for the given pin; uses the following status values: Status.Idle (no tokens received), Status.Working (at least one token received but more expected), Status.Completed (all tokens received).
- **GetPinValue(pinName: string): string** – returns the "value" field of a token for a specified pin; this is provided as a JSON-formatted string, containing either direct token data or reference information for the file or folder in a remote storage.
- **GetPinValues(pinName: string): List<string> -** as above but returns several "value" field strings for pins with multiple tokens.
- **GetPinValuesNDim(pinName: string): (List<string>, long[])** – as above but returns a multidimensional matrix of "value" fields, stored as an array (list), together with the matrix dimensions.
- **GetPinTokens(pinName: string): List<InputTokenMessage> -** returns a current list of received token messages in the JSON format for the specified pin.
- **SetProgress(progress: long): void** – sets the value of the current progress; this value will be reported to the BalticLSC system and displayed in the computation cockpit; the meaning of the values depend on the module (completion percentage information is suggested).
- **GetProgress(): long** – returns the previously set progress value.
- **SetStatus(status: Status): void** – sets the current status of computations; this value will be reported to the BalticLSC system and displayed in the computation cockpit; possible values are: Status.Idle (computations not yet started or is paused, e.g. waiting for further data), Status.Working (computations in progress), Status.Completed (computations are finished),

Status.Failed (computations have failed, e.g. the iteration limit has been reached without finding the result).

- **SetVariable(name: string, value: object): void** – safely stores a value of a global variable; can be used in case of multi-thread processing (e.g. communication between threads).
- **GetVariable(name: string): object** – retrieves a global variable value (see above).
- **GetEnvironmentVariable(name: string): string** – retrieves a system variable supplied by the external execution environment; can be used to parameterise the module's behaviour; such user-defined variables can be set when defining computation modules in the BalticLSC web interface.

5.4.2 IDataHandler

- **ObtainDataItem(pinName: string): string** – downloads a data item (file or folder) from a remote storage and returns the full path to it in the local file system, for the specified pin.
- **ObtainDataItems(pinName: string): List<string> -** as above but returns several path strings for pins with multiple tokens.
- **ObtainDataItemsNDim(pinName: string): (List<string>, long[]) -** as above but returns a multidimensional matrix of path strings, stored as an array (list), together with the matrix dimensions.
- **SendDataItem(pinName: string, data: string, isFinal: bool, msgUid: string = null): short** – uploads a data item (file or folder) to a remote storage for the specified pin; the isFinal parameter is used to mark the last element in the sequence (for multiple token pins); the optional msgUid parameter allows to specify a trace to the input data item for which the current data item (token) is produced.
- **SendToken(pinName: string, values: string, isFinal: bool, msgUid: string = null): short** – as above but used for direct data (no need to upload data); the data should be provided through the "values" parameter in the JSON format (see GetPinValue in IJobRegistry).
- **FinishProcessing(): short** – cleans-up the environment for the given job and sends the status to Status.Completed; should be called only after completing all the computations.
- **SendAckToken(msgUids: List<string>, isFinal: bool): short -** cleans-up after finishing processing of individual data items (tokens); sends a token notifying the execution environment about that some input data item (token) has been fully processed; this can be used to optimize distributed processing (can start further jobs earlier); setting the "isFinal" parameter to "true" indicates that we do not expect any jobs related to the given data item sequence to be run in the overall computation task – allows to optimize distributed processing.

6. Programming Computation Modules from scratch

In certain situations you might like to have better control over the exchange of tokens or might need to write a module in a language other than C# or Python. Thus, here we provide information on how to write a module from scratch which includes organising communication with the BalticLSC REST APIs and external data stores.

6.1 Reference Computation Module structure

An example structure of a Computation Module is presented in Figure 13. The module should introduce a REST controller (cf. JobController) that implements the JobAPI consisting of two operations (REST endpoints).

- **`ProcessTokenMessage`** – accepts a message containing an input Data Token and responds with a simple integer denoting the initial status of token validation;
- **`GetStatus`** – responds with an appropriate Job Status object denoting the current status of data processing.

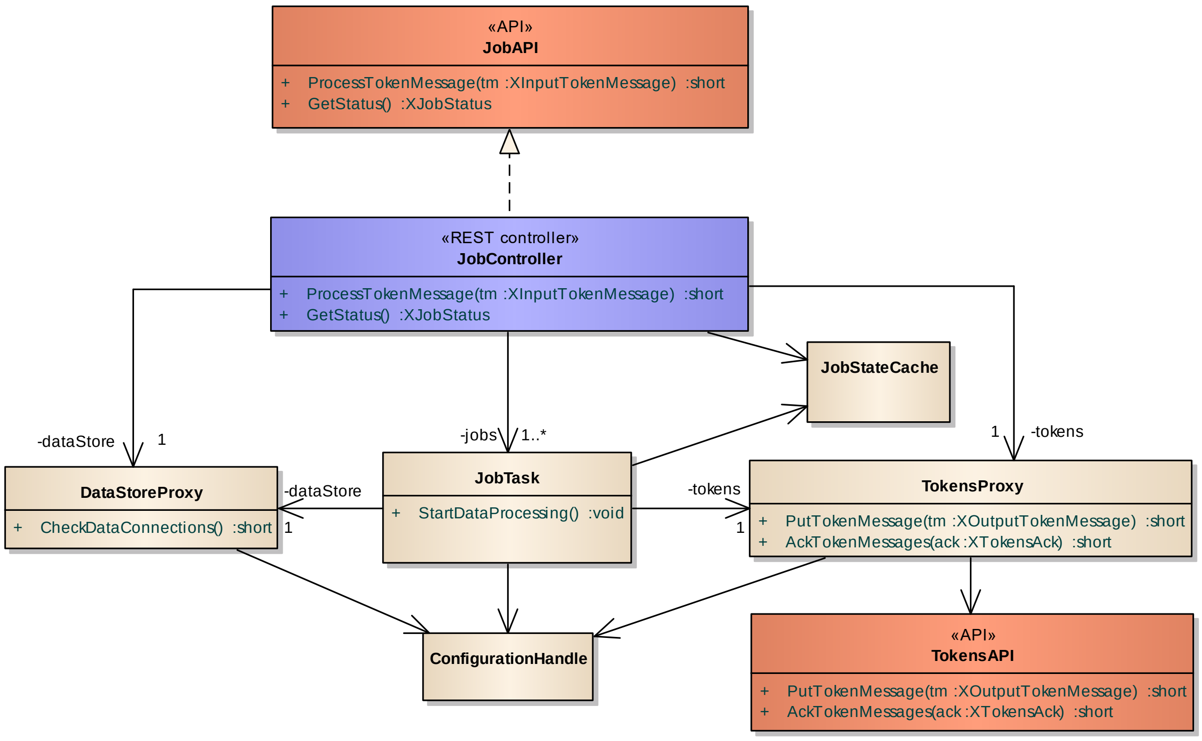

«API»
JobAPI
+ ProcessTokenMessage(tm :XInputTokenMessage) :short
+ GetStatus() :XJobStatus
«REST controller»
JobController
+ ProcessTokenMessage(tm :XInputTokenMessage) :short
+ GetStatus() :XJobStatus
JobStateCache
-dataStore
1
-jobs
1..*
1
-tokens
DataStoreProxy
+ CheckDataConnections() :short
-dataStore
1
JobTask
+ StartDataProcessing() :void
-tokens
1
TokensProxy
+ PutTokenMessage(tm :XOutputTokenMessage) :short
+ AckTokenMessages(ack :XTokensAck) :short
ConfigurationHandle
«API»
TokensAPI
+ PutTokenMessage(tm :XOutputTokenMessage) :short
+ AckTokenMessages(ack :XTokensAck) :short

Figure 13 Class model of the reference module structure

When the module gets initiated, it should read appropriate configuration data and make it accessible to other parts of code through a dedicated ConfigurationHandle. Through this configuration data, the module will be able to determine:

- connection with the data stores (cf. DataStoreProxy);
- optional constants of the data processing (computation) algorithm (cf. JobTask);
- access to the TokensAPI for sending output tokens (cf. TokensProxy).

To access the TokensAPI, the module should preferably introduce a dedicated REST proxy (cf. TokensProxy) that impelements access to two operations (REST endpoints).

- **`PutTokenMessage`** – sends a message containing an output Data Token and receives a simple integer response;
- **`AckTokenMessages`** – sends a special Tokens Ack message that acknowledges finishing of a single complete computation execution.

In further sections we present detailed information on how to handle configuration and how to perform token processing.

6.2 Configuration handling

The configuration of a Job Instance is determined through environment variables and configuration files. The module is obliged to read several standard environment variables and process the standard pins configuration file. Below we present detailed information on these standard configuration elements. The module developer can also define own configuration elements, depending on the needs of the particular computation.

6.2.1 Standard environment variables

The following environment variables should be read by the module initiation code.

- `SYS_MODULE_INSTANCE_UID` – the identifier of the module (Job Instance) granted by the Batch Manager that should be set in all the output tokens;
- `SYS_BATCH_MANAGER_TOKEN_ENDPOINT` – the address of the PutTokenMessage endpoint;
- `SYS_BATCH_MANAGER_ACK_ENDPOINT` – the address of the AckTokenMessages endpoint;
- `SYS_PIN_CONFIG_FILE_PATH` – the path to the pin configuration file, that is generated and provided by the Batch Manager.

6.2.2 Standard pins configuration file

The module should access the pins configuration file using the `SYS_PIN_CONFIG_FILE_PATH` variable. The file contains a JSON object. The object is an array of pin definitions. Each pin definition consists of several attributes.

- `PinName` – a string with the pin name, as specified in the module definition (see Section 7);
- `PinType` – either “input” or “output”, depending on the pin type;
- `AccessType` – a string defining the type of data storage and thus determining the access method (e.g. “MongoDB”, “FTP”, “Direct”);
- `DataMultiplicity` – either “single” or “multiple”, depending on the data multiplicity value;
- `TokenMultiplicity` - either “single” or “multiple”, depending on the token multiplicity value;
- `AccessCredential` – an object containing several attributes the provide credentials to access a particular data storage; its contents depends on the AccessType;
- `AccessPath` – an object containing one or more attributes that define the path to a particular data set or collection; ***Important note*****:** this part is normally supplied to or delivered by the module in data tokens and should be handled during token processing (see Section 6.3).

Below we provide an example JSON file containing two pin configurations with the above attributes.

```
[{ "PinName": "Image Folder",
   "PinType": "input",
   "AccessType": "MongoDB",
   "DataMultiplicity": "multiple",
   "TokenMultiplicity": "single",
   "AccessCredential": {
      "User": "someuser",
      "Password": "somepass",
      "Port": "27017",
      "Host": "b-36a1a684-51a8"
   }
},
{  "PinName": "Images",
   "PinType": "output",
   "AccessType": "FTP",
   "DataMultiplicity": "single",
   "TokenMultiplicity": "multiple",
   "AccessCredential": {
        "Host": "ftp.somehost.com",
        "User": "someuser",
        "Password": "somepass"
    },
    "AccessPath": {
        "ResourcePath": "/files/out"
    }
}]
```
Present only in special cases (see above)

Several standard AccessTypes are currently supported by the BalticLSC system. For each AccessType, the following credential and path parameters are provided in the configuration file or tokens.

- NoSQL_DB: AccessCredential = {Host, Port, User, Password}, AccessPath = {ResourcePath}
- RelationalDB: as above
- MySQL: as above
- FTP: as above
- MongoDB: AccessCredential as above, AccessPath = {Database, Collection, ObjectId}
- AzureLake: AccessCredential = {AccountName, ClientId, ClientSecret, TenantId, FileSystemName}, AccessPath = {ResourcePath}
- AWS3: AccessCredential = {AccessKey, SecretKey, BucketRegion, BucketName}, AccessPath = {ResourcePath}
- FileUpload: AccessCredential empty, AccessPath = {LocalPath}
- Direct: AccessCredential empty, AccessPath empty

Note that the “Direct” AccessType is used when the data should be passed directly in the Data Token, instead of passing access credentials to some data store and a path to some data set.

6.2.3 Developer-defined environment variables and configuration files

A particular module being developed might necessitate certain additional configuration elements. Thus, the BalticLSC system allows module developers to define dedicated environment variables and configuration files. These elements allow for parameterisation of Computation Modules (or generally: Computation Units) and thus are called Unit Parameters. For each parameter we need to define several attributes.

- `NameOrPath` - defines the name of the environment variable or the path of the configuration file that is to be used inside the module’s code;
- `DefaultValue` – specifies a string containing the default value of the particular environment variable or the default contents of the particular configuration file;
- `Type` – determines whether the particular parameter is a “Variable”, or a “Config”;
- `IsMandatory` – defines that the particular parameter value is mandatory and cannot be overridden in applications that use the module (see below).

The Unit Parameters can be provided during the definition of the particular Computation Module – see Section 7. The default values can be modified when defining an application that uses the module. The appropriate call to the module can be supplied with overriding values for each of the parameters. This does not pertain to parameters with “IsMandatory” switch being set.

6.3 Token processing

The computation lifecycle that involves token processing is illustrated in Figure 14. In the following we go through all the detailed steps necessary to process a token within a well-behaving Computation Module. The details of the endpoint parameters and responses are given in the next section.

- The process is started when an input Data Token message is received at the ProcessTokenMessage endpoint of the JobAPI and handled by the JobController.
- Following this, the module should check for correctness of the token’s structure and contents (“CheckToken”). If the token is incorrect, it should immediately send a response message “corrupted-token”.
    - Further on, the module should check connections to data stores (“CheckDataConnections”). In case when the data store does not respond and time-out

occurs, the module should immediately send a response message “no-response”. In case when the data store responds by indicating that the authorization or access to the data path has failed, the module should immediately send a response message “bad-credentials”.[4]

- If the token and data connections are correct, the module should start processing the data contained in the token (“StartDataProcessing”), set the computation status of the module to “Working” and immediately send a response message “ok”. Normally, the processing workload should be started asynchronously as a separate thread.
- When processing data, the module can connect to appropriate data stores through AccessCredentials taken from the pin configuration file (see Section 6.2.2). ***Important***: specific data items can be accessed through the AccessPath data taken from the input Data Token.

During and after finishing data processing, appropriate acknowledgement tokens and output token messages should be sent.

- When the module processes data (cf. “DoProcessData”), it creates some Data Set, usually by accessing (writing to) an appropriate data store, according to the specific Data Pin definition.
- When the output Data Set is ready, the module should send an output Data Token to the “PutTokenMessage” endpoint of the TokensAPI. ***Important***: the output token message should contain appropriate AccessPath data of the data item created by (output from) the module.
- When the module finishes a complete lifecycle for a single computation algorithm (processes all the necessary input tokens), it should send an “ack-ok” message to the AckTokenMessage endpoint of the TokensAPI.
- When the module encounters some data processing error (cause by corrupted data etc.), it should send a “failed-data-processing” message to the AckTokenMessages endpoint.

Note that the Batch Manager will transform the output tokens received from the current module, into input tokens. The tokens will start appropriate further instances of (other) computation modules if necessary, and as defined by the computation application.

[4] Note that when a “no-response” message is sent back, the Batch Manager will retry sending the token several times. In case when a “ok-bad-dataset” message is sent back, the Batch Manager will stop computations.

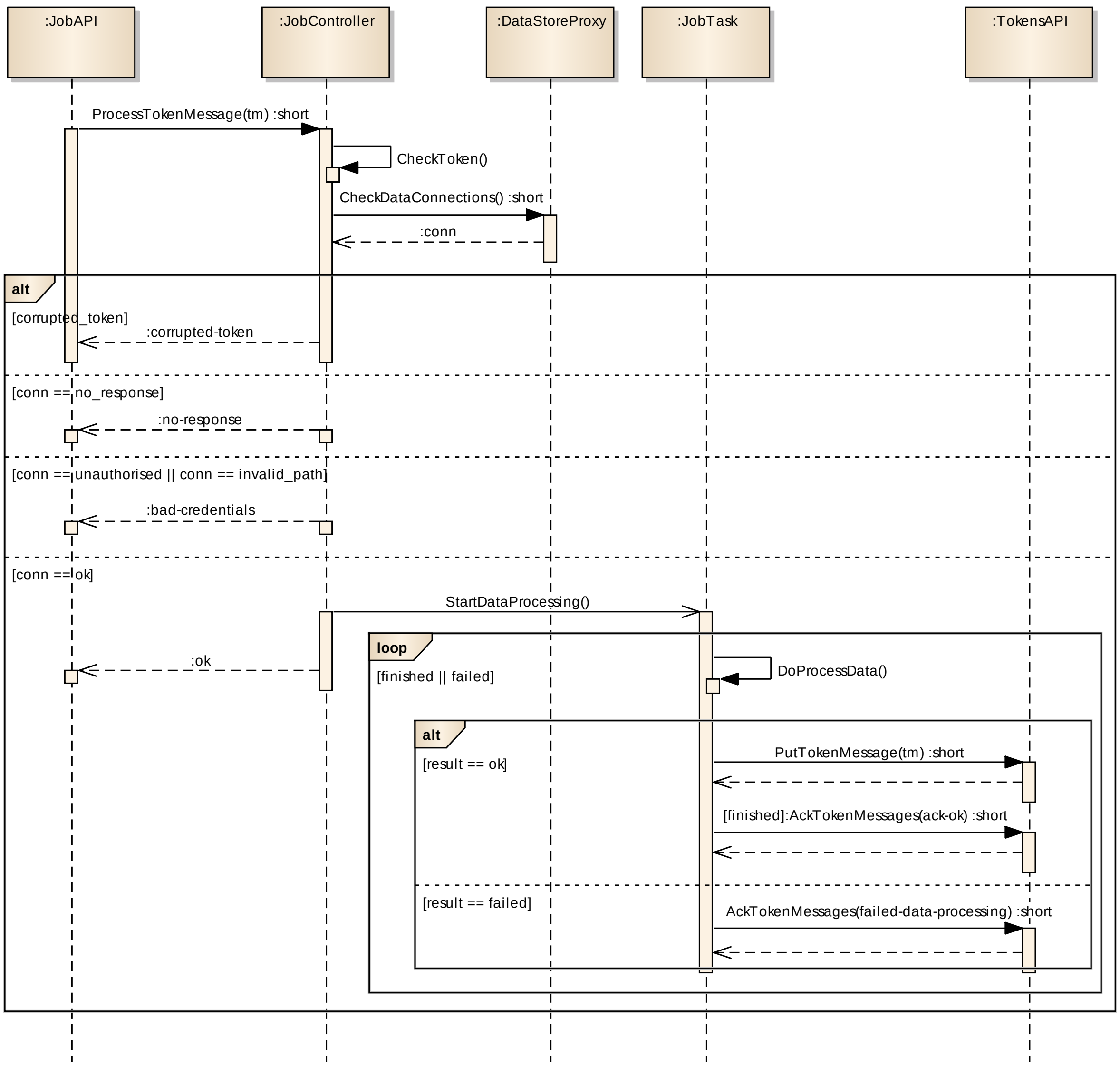


Figure 14 Sequence diagram illustrating token processing and error handling

6.4 API endpoints

In this section we provide a detailed specification of the two APIs and their endpoints.

6.4.1 JobAPI

This API should be implemented by the Computation Module. The following lists the two endpoints with the provided parameters (Data Transfer Objects, DTOs) and expected responses.

Note 1: all the DTOs are provided/supplied in JSON format.

Note 2: all the DTOs in POST endpoints are provided in request bodies.

1. `ProcessTokenMessage()`

method: `POST`, endpoint path: `/token`

`XInputTokenMessage` – token sent by the Batch Manager, containing information about data and parameters for the computation.

- MsgUid :string – Uid of the Token to be processed;
- PinName :string – name of the input Data Pin of this module to which the token is directed;
- Values :string – a JSON object with the actual token data (access data, parameters, etc.);
- TokenSeqStack :IEnumerable<XSeqToken> – contains message counters (for multiple token pins).

`XSeqToken` – represents the message number in the sequence

- SeqUid :string – Uid of the sequence
- No :long – token number in sequence
- IsFinal :bool – true if this message is final in the given sequence (for Multiple pins)

Example:

```
{
   "MsgUid": "b-fdeaccff-e219-401b-bb3d-aaa508b40116",
   "PinName": "ImageReader",
   "Values":
      "{\"Database\": \"Image_channel_separator\",
      \"Collection\": \"Image_channel_separator_b-8b0e36\",
      \"ObjectId\": \"60254938cf19c15519cdff71\"}",
   "TokenSeqStack": []

}
```

`Responses`

- "corrupted-token" - BadRequest (400) with some optional message indicating the error;
- "no-response" – NotFound (404);
- "bad-credentials" – Unauthorized (401) with some optional message indicating the source of the error;
- "ok" – OK (200).

2. `GetStatus()`

method: `GET`, endpoint path: `/status`

`Response`

- "ok" – OK (200) with the XJobStatus object

`XJobStatus` – current state of the computation

- Status :ComputationStatus – current module status
- JobProgress :long – current computation progress. Amount of completed work if possible given as a percentage, or just a number.

`ComputationStatus` – current state of the module.

- Idle = 0 – set when computation is not yet started (not all the tokens are available) or when the module is waiting for additional data,
- Working = 1 – set when performing computation,
- Completed = 2 – set when the computation for the given set of input tokens is finished,
- Failed = 3 – set when the computation has failed for any reason.

6.4.2 TokensAPI

This API should be used by the Computation Module. The following lists the two endpoints with the parameters to be provided and responses to be expected.

1. PutTokenMessage

method: `POST`, endpoint path from env. variable: `SYS_BATCH_MANAGER_TOKEN_ENDPOINT`

XOutputTokenMessage – token sent by module to the Batch Manager, containing information about some finished part of computation

- PinName :string – name of the Provided Pins of this module that is sending token out
- SenderUid :string – Uid of this module instance (from environment variables)
- Values :string – a JSON with the actual data (access data, parameters, etc.)
- BaseMsgUid :string – the Uid of one of the processed Tokens (from processed XInputTokenMessage)
- IsFinal :bool – true if this message is final in the given sequence (for Multiple pins)

Example:

```
{
   "PinName": "ImageWriter",
   "SenderUid": "b-da649fa7-df78-4dbe-81d0-b2fa63fe89d8",
   "Values":
      "{\"ResourcePath\": \"/images/out\"}",
   "BaseMsgUid": "b-fdeaccff-e219-401b-bb3d-aaa508b40116",
   "IsFinal": true
}
```

Responses

- “ok” – OK (200) with an optional message indicating an error.

2. FinalizeTokenMessageProcessing

method: `POST`, endpoint path from env. variable: `SYS_BATCH_MANAGER_ACK_ENDPOINT`

XTockensAck – token confirming completion of computations of the given input tokens

- MsgUids :List<string> – list of tokens (XInputTokenMessage) Uid confirmed as processed
- SenderUid :string – Uid of this module instance (from environment variables)
- IsFinal :bool – true if this message is final in the given sequence (for Multiple pins)
- IsFailed: bool – true if this computation has failed (for any reason)
- Note: string – a short message communicating the reason for failure

Example:

```
{
   "SenderUid": "b-da649fa7-df78-4dbe-81d0-b2fa63fe89d8",
   "MsgUids": [
      "b-fdeaccff-e219-401b-bb3d-aaa508b40116"
   ],
   "IsFinal": true,
   "IsFailed": true,
   "Note": "File not found"
}
```

Responses

- “ok” – OK (200) with an optional message indicating an error.

7. Registration of a computation module

To use your module, it must be registered in the BalticLSC system. This can be done through the project website[5]. To register the module you need to provide its Docker image address from the Docker Hub and appropriate BalticModuleBuild (appropriate for the docker file associated with the BalticLSC computation module) containing all additional information needed to run the docker image in the BalticLSC environment. The module build should also declare all the resources necessary for the module to run.

The following data has to be supplied before releasing the module to be used within BalticLSC.

- Name – the module's unique name through which it will be visible in the system (including other developers – module users);
- Image – the path in Docker Hub from which the module's image can be downloaded.
- Description (long + short) – comprehensive text that explains the purpose of the module.
- Keywords – a list of phrases that should facilitate determining suitability of the module for specific computation problems.
- Icon – an optional image that will be used to distinguish the module visually.
- Pins – an array of pin specifications, where for each pin the following data should be given.
    - Pin Name – short descriptive name unique in the context of the given module;
    - Pin Type – input or output;
    - Token Multiplicity – single or multiple;
    - Data Multiplicity – single or multiple;
    - Access Type – one of the types presented in Section 6.2.2;
    - Data Type – identification of the type of data that will be handled through the pin (e.g. Image, Video, Direct);
    - Data Structure – identifies a data schema that allows to send data directly – present when the Data Type and Access Type are "Direct"
- Required resources – specification of ranges for the following resources. Each range specifies minimum required value and maximum value.
    - CPU – number of milli-CPUs, e.g. 1500 means 1.5 CPU time slots;
    - GPU – number of GPUs, e.g. 2 means two full GPUs;
    - Memory – size of available memory in GB, e.g. 2 means 2 gigabytes;
    - Storage – size of available local storage (disk space) in GB, e.g. 2 means 2 gigabytes.
- Custom environment variables and configuration files – defined as in Section 6.2.3.

[5] Currently, modules can be added only through contacting the BalticLSC development team. Please specify what resources (memory, CPUs, GPUS, storage, external databases etc.) are needed for the module to run and the BalticLSC team will configure the system for you.

8. Performing large-scale computations on the BalticLSC Network using the BalticLSC Software

8.1 How the BalticLSC works – Introduction video

The BalticLSC introduction video has been recorded to explain the concept of the BalticLSC Environment (called BalticLSC Platform for the purpose of simplification). The video is available on project YouTube Channel and explains the basic concepts of BalticLSC, the BalticLSC Software Frontend User Interface, how to use the BalticLSC and how does it work.

The video is available under: https://www.youtube.com/watch?v=yaruuw6fCAg

8.2 Performing computations – BalticLSC Demo Tutorial video

The BalticLSC Software is very intuitive, but to help the end-users with their first steps on the website a demo tutorial video has been created. It explains step-by-step how to perform computations on the BalticLSC Network using an already available Computation Application with data provided by the end user.

The video is available under: https://www.youtube.com/watch?v=zA-VLDNm3go